

# **R: A Language and Environment for Statistical Computing**

## **Reference Index**

The R Development Core Team

Version 2.0.0 beta (2004-09-21)

Copyright (©) 1999–2003 R Foundation for Statistical Computing.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the R Development Core Team.

R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under the terms of the GNU General Public License. For more information about these matters, see <http://www.gnu.org/copyleft/gpl.html>.

ISBN 3-900051-07-0

# Contents

<b>1</b>	<b>The base package</b>	<b>1</b>
	.Device . . . . .	1
	.Machine . . . . .	1
	.Platform . . . . .	3
	.Script . . . . .	4
	abbreviate . . . . .	5
	abs . . . . .	6
	agrep . . . . .	7
	all . . . . .	8
	all.equal . . . . .	9
	all.names . . . . .	11
	any . . . . .	12
	aperm . . . . .	13
	append . . . . .	14
	apply . . . . .	15
	approxfun . . . . .	16
	args . . . . .	18
	Arithmetic . . . . .	19
	array . . . . .	20
	as.data.frame . . . . .	21
	as.environment . . . . .	23
	as.function . . . . .	23
	as.POSIX* . . . . .	24
	AsIs . . . . .	25
	assign . . . . .	26
	assignOps . . . . .	28
	attach . . . . .	29
	attr . . . . .	30
	attributes . . . . .	31
	autoload . . . . .	32
	backsolve . . . . .	33
	base-deprecated . . . . .	34
	basename . . . . .	35
	Bessel . . . . .	36
	body . . . . .	38
	bquote . . . . .	39
	browser . . . . .	40

builtins	40
by	41
c	42
call	43
capabilities	44
cat	45
cbind	46
char.expand	48
character	49
charmatch	50
chartr	51
chol	52
chol2inv	54
class	55
col	56
colSums	57
commandArgs	58
comment	59
Comparison	60
complex	61
conditions	63
conflicts	66
connections	67
Constants	72
contributors	73
Control	74
copyright	75
count.fields	75
crossprod	76
cumsum	77
cut	78
cut.POSIXt	80
data.class	81
data.frame	82
data.matrix	83
date	84
Dates	85
DateTimeClasses	85
dcf	88
debug	89
Defunct	90
delay	90
deparse	91
deparseOpts	93
Deprecated	94
det	95
detach	96
diag	97

diff . . . . .	98
difftime . . . . .	99
dim . . . . .	101
dimnames . . . . .	102
do.call . . . . .	103
double . . . . .	103
dput . . . . .	105
drop . . . . .	106
dump . . . . .	107
duplicated . . . . .	108
dyn.load . . . . .	109
eapply . . . . .	112
eigen . . . . .	113
environment . . . . .	114
eval . . . . .	116
exists . . . . .	118
expand.grid . . . . .	120
expression . . . . .	120
Extract . . . . .	121
Extract.data.frame . . . . .	124
Extract.factor . . . . .	127
Extremes . . . . .	128
factor . . . . .	129
file.access . . . . .	131
file.choose . . . . .	133
file.info . . . . .	133
file.path . . . . .	135
file.show . . . . .	135
files . . . . .	136
findInterval . . . . .	138
force . . . . .	139
Foreign . . . . .	140
formals . . . . .	143
format . . . . .	144
format.Date . . . . .	146
format.info . . . . .	148
formatC . . . . .	149
formatDL . . . . .	151
function . . . . .	153
gc . . . . .	154
gc.time . . . . .	155
gctorture . . . . .	156
get . . . . .	156
getCallingDLL . . . . .	158
getDLLRegisteredRoutines . . . . .	159
getLoadedDLLs . . . . .	160
getNativeSymbolInfo . . . . .	161
getNumCConverters . . . . .	163

getpid . . . . .	164
getwd . . . . .	165
gl . . . . .	166
grep . . . . .	167
groupGeneric . . . . .	169
gzcon . . . . .	172
Hyperbolic . . . . .	173
identical . . . . .	174
ifelse . . . . .	175
integer . . . . .	176
interaction . . . . .	177
interactive . . . . .	178
Internal . . . . .	179
InternalMethods . . . . .	179
invisible . . . . .	180
is.finite . . . . .	181
is.function . . . . .	182
is.language . . . . .	183
is.object . . . . .	183
is.R . . . . .	184
is.recursive . . . . .	185
is.single . . . . .	186
jitter . . . . .	186
kappa . . . . .	187
kronecker . . . . .	189
labels . . . . .	190
lapply . . . . .	191
Last.value . . . . .	192
length . . . . .	193
levels . . . . .	194
libPaths . . . . .	195
library . . . . .	196
library.dynam . . . . .	199
license . . . . .	201
list . . . . .	202
list.files . . . . .	203
load . . . . .	204
localeconv . . . . .	206
locales . . . . .	207
log . . . . .	208
Logic . . . . .	209
logical . . . . .	211
lower.tri . . . . .	212
ls . . . . .	212
make.names . . . . .	214
make.unique . . . . .	215
manglePackageName . . . . .	216
mapply . . . . .	216

margin.table . . . . .	217
mat.or.vec . . . . .	218
match . . . . .	219
match.arg . . . . .	220
match.call . . . . .	221
match.fun . . . . .	222
matmult . . . . .	223
matrix . . . . .	224
maxCol . . . . .	225
mean . . . . .	226
Memory . . . . .	227
Memory-limits . . . . .	229
memory.profile . . . . .	230
merge . . . . .	230
missing . . . . .	232
mode . . . . .	233
NA . . . . .	234
name . . . . .	236
names . . . . .	237
nargs . . . . .	238
nchar . . . . .	239
nlevels . . . . .	240
noquote . . . . .	240
NotYet . . . . .	241
nrow . . . . .	242
ns-dblcolon . . . . .	243
ns-hooks . . . . .	244
ns-load . . . . .	245
ns-topenv . . . . .	246
NULL . . . . .	247
numeric . . . . .	248
octmode . . . . .	249
on.exit . . . . .	250
options . . . . .	250
order . . . . .	255
outer . . . . .	257
package-version . . . . .	258
Paren . . . . .	259
parse . . . . .	260
paste . . . . .	261
path.expand . . . . .	262
pmatch . . . . .	262
polyroot . . . . .	264
pos.to.env . . . . .	265
pretty . . . . .	265
Primitive . . . . .	267
print . . . . .	268
print.data.frame . . . . .	269

print.default	270
prmatrix	271
proc.time	272
prod	273
prop.table	274
pushBack	275
qr	276
QR.Auxiliaries	278
quit	279
R.home	281
R.Version	281
r2dtable	282
Random	283
Random.user	287
range	289
rank	290
raw	291
rawConversion	292
RdUtils	293
read.table	294
readBin	298
readline	301
readLines	302
real	303
Recall	304
reg.finalizer	304
regex	305
remove	309
rep	310
replace	312
rev	313
rle	313
Round	314
round.POSIXt	316
row	317
row.names	318
row/colnames	319
rowsum	320
sample	321
save	322
scale	324
scan	325
search	328
seek	329
seq	330
seq.Date	332
seq.POSIXt	333
sequence	334

sets . . . . .	335
showConnections . . . . .	336
shQuote . . . . .	337
sign . . . . .	338
Signals . . . . .	339
sink . . . . .	339
slice.index . . . . .	341
slotOp . . . . .	342
socketSelect . . . . .	342
solve . . . . .	343
sort . . . . .	344
source . . . . .	346
Special . . . . .	348
split . . . . .	350
sprintf . . . . .	351
sQuote . . . . .	353
stack . . . . .	354
Startup . . . . .	356
stop . . . . .	358
stopifnot . . . . .	360
strptime . . . . .	361
strsplit . . . . .	364
structure . . . . .	366
strwrap . . . . .	366
subset . . . . .	367
substitute . . . . .	369
substr . . . . .	370
sum . . . . .	372
summary . . . . .	372
svd . . . . .	374
sweep . . . . .	376
switch . . . . .	377
Syntax . . . . .	378
Sys.getenv . . . . .	379
Sys.info . . . . .	380
sys.parent . . . . .	381
Sys.putenv . . . . .	383
Sys.sleep . . . . .	384
sys.source . . . . .	385
Sys.time . . . . .	385
system . . . . .	386
system.file . . . . .	387
system.time . . . . .	388
t . . . . .	389
table . . . . .	390
tabulate . . . . .	392
tapply . . . . .	393
taskCallback . . . . .	395

taskCallbackManager . . . . .	397
taskCallbackNames . . . . .	398
tempfile . . . . .	399
textConnection . . . . .	400
tilde . . . . .	402
toString . . . . .	403
trace . . . . .	404
traceback . . . . .	407
transform . . . . .	408
Trig . . . . .	409
try . . . . .	410
type.convert . . . . .	411
typeof . . . . .	412
unique . . . . .	413
unlink . . . . .	414
unlist . . . . .	415
unname . . . . .	416
UseMethod . . . . .	417
UserHooks . . . . .	418
vector . . . . .	420
warning . . . . .	421
warnings . . . . .	422
weekdays . . . . .	423
which . . . . .	424
which.min . . . . .	426
with . . . . .	427
write . . . . .	428
write.table . . . . .	429
writeLines . . . . .	431
zip.file.extract . . . . .	431
zpackages . . . . .	432
<b>2 The datasets package . . . . .</b>	<b>435</b>
ability.cov . . . . .	435
airmiles . . . . .	436
AirPassengers . . . . .	437
airquality . . . . .	438
anscombe . . . . .	439
attenu . . . . .	440
attitude . . . . .	441
austres . . . . .	442
beavers . . . . .	442
BJsales . . . . .	444
BOD . . . . .	444
cars . . . . .	445
ChickWeight . . . . .	446
chickwts . . . . .	447
CO2 . . . . .	448
co2 . . . . .	449

discoveries	450
DNase	451
esoph	452
euro	453
eurodist	454
EuStockMarkets	454
faithful	455
Formaldehyde	456
freeny	457
HairEyeColor	458
Harman23.cor	459
Harman74.cor	459
Indometh	460
infert	461
InsectSprays	462
iris	463
islands	464
JohnsonJohnson	465
LakeHuron	465
lh	466
LifeCycleSavings	466
Loblolly	467
longley	468
lynx	469
morley	470
mtcars	471
nhtemp	471
Nile	472
nottem	473
Orange	474
OrchardSprays	475
PlantGrowth	476
precip	477
presidents	477
pressure	478
Puromycin	479
quakes	480
randu	481
rivers	482
rock	483
sleep	483
stackloss	484
state	485
sunspot.month	487
sunspot.year	488
sunspots	488
swiss	489
Theoph	490

Titanic . . . . .	491
ToothGrowth . . . . .	493
treering . . . . .	493
trees . . . . .	494
UCBAdmissions . . . . .	495
UKDriverDeaths . . . . .	496
UKgas . . . . .	497
UKLungDeaths . . . . .	498
USAccDeaths . . . . .	499
USArrests . . . . .	499
USJudgeRatings . . . . .	500
USPersonalExpenditure . . . . .	501
uspop . . . . .	501
VADeaths . . . . .	502
volcano . . . . .	503
warpbreaks . . . . .	504
women . . . . .	505
WorldPhones . . . . .	505
WWWusage . . . . .	506
<b>3 The grDevices package</b>	<b>509</b>
check.options . . . . .	509
col2rgb . . . . .	510
colors . . . . .	512
dev.interactive . . . . .	512
dev.xxx . . . . .	513
dev2 . . . . .	514
dev2bitmap . . . . .	516
Devices . . . . .	517
gray . . . . .	518
Hershey . . . . .	519
hsv . . . . .	522
Japanese . . . . .	523
palette . . . . .	524
Palettes . . . . .	525
pdf . . . . .	526
pictex . . . . .	528
plotmath . . . . .	530
png . . . . .	533
postscript . . . . .	535
postscriptFonts . . . . .	539
quartz . . . . .	541
quartzFonts . . . . .	542
recordPlot . . . . .	543
rgb . . . . .	543
rgb2hsv . . . . .	544
units . . . . .	546
x11 . . . . .	547
X11Fonts . . . . .	548

xfig . . . . .	549
<b>4 The graphics package</b>	<b>551</b>
abline . . . . .	551
arrows . . . . .	552
assocplot . . . . .	554
axis . . . . .	555
axis.POSIXct . . . . .	557
axTicks . . . . .	558
barplot . . . . .	559
box . . . . .	562
boxplot . . . . .	563
boxplot.stats . . . . .	566
bxp . . . . .	568
chull . . . . .	571
contour . . . . .	572
coplot . . . . .	575
curve . . . . .	578
dotchart . . . . .	579
filled.contour . . . . .	580
fourfoldplot . . . . .	583
frame . . . . .	585
grid . . . . .	585
hist . . . . .	586
hist.POSIXt . . . . .	589
identify . . . . .	591
image . . . . .	592
layout . . . . .	594
legend . . . . .	596
lines . . . . .	600
locator . . . . .	602
matplot . . . . .	603
mosaicplot . . . . .	605
mtext . . . . .	608
n2mfrow . . . . .	610
nclass . . . . .	611
pairs . . . . .	612
panel.smooth . . . . .	614
par . . . . .	615
persp . . . . .	622
pie . . . . .	625
plot . . . . .	626
plot.data.frame . . . . .	628
plot.default . . . . .	629
plot.design . . . . .	631
plot.factor . . . . .	633
plot.formula . . . . .	634
plot.histogram . . . . .	635
plot.table . . . . .	636

plot.window . . . . .	637
plot.xy . . . . .	638
points . . . . .	639
polygon . . . . .	641
rect . . . . .	643
rug . . . . .	645
screen . . . . .	646
segments . . . . .	648
stars . . . . .	649
stem . . . . .	652
stripchart . . . . .	653
strwidth . . . . .	654
sunflowerplot . . . . .	656
symbols . . . . .	658
text . . . . .	660
title . . . . .	662
xy.coords . . . . .	663
xyz.coords . . . . .	665
<b>5 The grid package</b> . . . . .	<b>667</b>
absolute.size . . . . .	667
convertNative . . . . .	668
dataViewport . . . . .	669
drawDetails . . . . .	670
editDetails . . . . .	671
gEdit . . . . .	672
gpar . . . . .	673
gPath . . . . .	675
Grid . . . . .	676
Grid Viewports . . . . .	676
grid.add . . . . .	680
grid.arrows . . . . .	681
grid.circle . . . . .	684
grid.collection . . . . .	685
grid.convert . . . . .	686
grid.copy . . . . .	688
grid.display.list . . . . .	689
grid.draw . . . . .	690
grid.edit . . . . .	691
grid.frame . . . . .	692
grid.get . . . . .	693
grid.grab . . . . .	694
grid.grill . . . . .	695
grid.grob . . . . .	696
grid.layout . . . . .	697
grid.lines . . . . .	699
grid.locator . . . . .	700
grid.move.to . . . . .	702
grid.newpage . . . . .	703

grid.pack . . . . .	704
grid.place . . . . .	705
grid.plot.and.legend . . . . .	706
grid.points . . . . .	707
grid.polygon . . . . .	708
grid.pretty . . . . .	709
grid.prompt . . . . .	710
grid.rect . . . . .	711
grid.remove . . . . .	712
grid.segments . . . . .	713
grid.set . . . . .	714
grid.show.layout . . . . .	715
grid.show.viewport . . . . .	716
grid.text . . . . .	717
grid.xaxis . . . . .	719
grid.yaxis . . . . .	720
grobWidth . . . . .	722
plotViewport . . . . .	722
pop.viewport . . . . .	723
push.viewport . . . . .	724
Querying the Viewport Tree . . . . .	724
stringWidth . . . . .	726
unit . . . . .	726
unit.c . . . . .	728
unit.length . . . . .	729
unit.pmin . . . . .	730
unit.rep . . . . .	730
validDetails . . . . .	731
vpPath . . . . .	732
widthDetails . . . . .	733
Working with Viewports . . . . .	734
<b>6 The methods package . . . . .</b>	<b>737</b>
.BasicFunsList . . . . .	737
as . . . . .	738
BasicClasses . . . . .	742
callNextMethod . . . . .	743
Classes . . . . .	745
classRepresentation-class . . . . .	746
Documentation . . . . .	747
environment-class . . . . .	749
fixPre1.8 . . . . .	750
genericFunction-class . . . . .	751
GenericFunctions . . . . .	752
getClass . . . . .	756
getMethod . . . . .	757
getPackageName . . . . .	760
hasArg . . . . .	761
initialize-methods . . . . .	762

is . . . . .	764
isSealedMethod . . . . .	767
language-class . . . . .	768
LinearMethodsList-class . . . . .	769
makeClassRepresentation . . . . .	770
MethodDefinition-class . . . . .	771
Methods . . . . .	772
MethodsList-class . . . . .	775
MethodWithNext-class . . . . .	776
new . . . . .	777
ObjectsWithPackage-class . . . . .	779
promptClass . . . . .	779
promptMethods . . . . .	781
representation . . . . .	782
SClassExtension-class . . . . .	784
setClass . . . . .	785
setClassUnion . . . . .	789
setGeneric . . . . .	791
setMethod . . . . .	795
setOldClass . . . . .	798
show . . . . .	800
showMethods . . . . .	802
signature-class . . . . .	803
slot . . . . .	804
StructureClasses . . . . .	806
TraceClasses . . . . .	806
validObject . . . . .	808
<b>7 The stats package</b> . . . . .	<b>811</b>
.checkMFClasses . . . . .	811
acf . . . . .	812
acf2AR . . . . .	814
add1 . . . . .	815
addmargins . . . . .	817
aggregate . . . . .	819
AIC . . . . .	821
alias . . . . .	822
anova . . . . .	824
anova.glm . . . . .	825
anova.lm . . . . .	826
ansari.test . . . . .	828
aov . . . . .	830
ar . . . . .	832
ar.ols . . . . .	835
arima . . . . .	837
arima.sim . . . . .	841
arima0 . . . . .	842
ARMAacf . . . . .	846
ARMAtoMA . . . . .	847

as.hclust . . . . .	848
asOneSidedFormula . . . . .	849
ave . . . . .	849
bandwidth . . . . .	850
bartlett.test . . . . .	852
Beta . . . . .	853
binom.test . . . . .	855
Binomial . . . . .	856
biplot . . . . .	858
biplot.princomp . . . . .	859
birthday . . . . .	860
Box.test . . . . .	862
C . . . . .	863
cancor . . . . .	864
case/variable.names . . . . .	865
Cauchy . . . . .	866
chisq.test . . . . .	868
Chisquare . . . . .	870
clearNames . . . . .	872
cmdscale . . . . .	872
coef . . . . .	874
complete.cases . . . . .	875
confint . . . . .	876
constrOptim . . . . .	877
contrast . . . . .	879
contrasts . . . . .	880
convolve . . . . .	881
cophenetic . . . . .	882
cor . . . . .	884
cor.test . . . . .	886
cov.wt . . . . .	889
cpgram . . . . .	890
cutree . . . . .	891
decompose . . . . .	892
delete.response . . . . .	893
dendrapply . . . . .	894
dendrogram . . . . .	895
density . . . . .	899
deriv . . . . .	902
deviance . . . . .	904
df.residual . . . . .	905
diffinv . . . . .	906
dist . . . . .	907
dummy.coef . . . . .	909
ecdf . . . . .	911
eff.aovlist . . . . .	913
effects . . . . .	914
embed . . . . .	915

expand.model.frame . . . . .	916
Exponential . . . . .	917
extractAIC . . . . .	918
factanal . . . . .	919
factor.scope . . . . .	923
family . . . . .	924
FDist . . . . .	926
fft . . . . .	927
filter . . . . .	928
fisher.test . . . . .	930
fitted . . . . .	932
fivenum . . . . .	933
fligner.test . . . . .	934
formula . . . . .	935
formula.nls . . . . .	937
friedman.test . . . . .	938
ftable . . . . .	940
ftable.formula . . . . .	942
GammaDist . . . . .	943
Geometric . . . . .	945
getInitial . . . . .	946
glm . . . . .	947
glm.control . . . . .	951
glm.summaries . . . . .	952
hclust . . . . .	953
heatmap . . . . .	956
HoltWinters . . . . .	959
Hypergeometric . . . . .	962
identify.hclust . . . . .	963
influence.measures . . . . .	965
integrate . . . . .	968
interaction.plot . . . . .	970
IQR . . . . .	972
is.empty.model . . . . .	973
isoreg . . . . .	973
KalmanLike . . . . .	975
kernapply . . . . .	977
kernel . . . . .	978
kmeans . . . . .	979
kruskal.test . . . . .	980
ks.test . . . . .	982
ksmooth . . . . .	984
lag . . . . .	985
lag.plot . . . . .	986
line . . . . .	987
lm . . . . .	988
lm.fit . . . . .	991
lm.influence . . . . .	992

lm.summaries . . . . .	994
loadings . . . . .	995
loess . . . . .	996
loess.control . . . . .	998
Logistic . . . . .	999
logLik . . . . .	1001
loglin . . . . .	1002
Lognormal . . . . .	1004
lowess . . . . .	1005
ls.diag . . . . .	1006
ls.print . . . . .	1007
lsfit . . . . .	1008
mad . . . . .	1009
mahalanobis . . . . .	1010
make.link . . . . .	1011
makepredictcall . . . . .	1012
manova . . . . .	1013
mantelhaen.test . . . . .	1014
mcnemar.test . . . . .	1017
median . . . . .	1018
medpolish . . . . .	1019
model.extract . . . . .	1020
model.frame . . . . .	1021
model.matrix . . . . .	1023
model.tables . . . . .	1025
monthplot . . . . .	1026
mood.test . . . . .	1028
Multinomial . . . . .	1029
na.action . . . . .	1031
na.contiguous . . . . .	1031
na.fail . . . . .	1032
naprint . . . . .	1033
naresid . . . . .	1034
NegBinomial . . . . .	1035
nextn . . . . .	1037
nlm . . . . .	1037
nls . . . . .	1040
nls.control . . . . .	1042
nlsModel . . . . .	1044
NLSstAsymptotic . . . . .	1045
NLSstClosestX . . . . .	1046
NLSstLfAsymptote . . . . .	1047
NLSstRtAsymptote . . . . .	1048
Normal . . . . .	1048
numericDeriv . . . . .	1050
offset . . . . .	1051
oneway.test . . . . .	1052
optim . . . . .	1053

optimize . . . . .	1058
order.dendrogram . . . . .	1060
p.adjust . . . . .	1061
pairwise.prop.test . . . . .	1063
pairwise.t.test . . . . .	1063
pairwise.table . . . . .	1064
pairwise.wilcox.test . . . . .	1065
plot.acf . . . . .	1066
plot.density . . . . .	1067
plot.HoltWinters . . . . .	1068
plot.isoreg . . . . .	1069
plot.lm . . . . .	1070
plot.ppr . . . . .	1072
plot.profile.nls . . . . .	1073
plot.spec . . . . .	1074
plot.stepfun . . . . .	1075
plot.ts . . . . .	1077
Poisson . . . . .	1078
poly . . . . .	1080
power . . . . .	1081
power.anova.test . . . . .	1082
power.prop.test . . . . .	1083
power.t.test . . . . .	1085
PP.test . . . . .	1086
ppoints . . . . .	1087
ppr . . . . .	1088
prcomp . . . . .	1091
predict . . . . .	1093
predict.Arima . . . . .	1094
predict.glm . . . . .	1096
predict.HoltWinters . . . . .	1097
predict.lm . . . . .	1098
predict.loess . . . . .	1100
predict.nls . . . . .	1102
predict.smooth.spline . . . . .	1103
preplot . . . . .	1104
princomp . . . . .	1105
print.power.htest . . . . .	1107
print.ts . . . . .	1108
printCoefmat . . . . .	1109
profile . . . . .	1110
profile.nls . . . . .	1111
profiler . . . . .	1112
profiler.nls . . . . .	1113
proj . . . . .	1115
prop.test . . . . .	1117
prop.trend.test . . . . .	1119
qqnorm . . . . .	1120

quade.test . . . . .	1121
quantile . . . . .	1123
read.ftable . . . . .	1125
rect.hclust . . . . .	1127
relevel . . . . .	1128
reorder . . . . .	1129
reorder.factor . . . . .	1130
replications . . . . .	1131
reshape . . . . .	1132
residuals . . . . .	1134
runmed . . . . .	1135
scatter.smooth . . . . .	1137
screeplot . . . . .	1139
sd . . . . .	1140
se.contrast . . . . .	1140
selfStart . . . . .	1142
setNames . . . . .	1143
shapiro.test . . . . .	1144
SignRank . . . . .	1145
smooth . . . . .	1146
smooth.spline . . . . .	1148
smoothEnds . . . . .	1152
sortedXyData . . . . .	1153
spec.ar . . . . .	1154
spec.pgram . . . . .	1155
spec.taper . . . . .	1157
spectrum . . . . .	1158
splinefun . . . . .	1160
SSasymp . . . . .	1162
SSasympOff . . . . .	1163
SSasympOrig . . . . .	1164
SSbiexp . . . . .	1165
SSfol . . . . .	1166
SSfpl . . . . .	1167
SSgompertz . . . . .	1168
SSlogis . . . . .	1169
SSmicmen . . . . .	1170
SSweibull . . . . .	1171
start . . . . .	1172
stat.anova . . . . .	1173
step . . . . .	1174
stepfun . . . . .	1176
stl . . . . .	1178
stlmethods . . . . .	1180
StructTS . . . . .	1181
summary.aov . . . . .	1184
summary.glm . . . . .	1185
summary.lm . . . . .	1187

summary.manova . . . . .	1189
summary.princomp . . . . .	1190
supsmu . . . . .	1191
symnum . . . . .	1192
t.test . . . . .	1195
TDist . . . . .	1197
termplot . . . . .	1198
terms . . . . .	1200
terms.formula . . . . .	1201
terms.object . . . . .	1202
time . . . . .	1203
toeplitz . . . . .	1204
ts . . . . .	1204
ts-methods . . . . .	1206
ts.plot . . . . .	1207
ts.union . . . . .	1208
tsdiag . . . . .	1209
tsp . . . . .	1210
tsSmooth . . . . .	1211
Tukey . . . . .	1212
TukeyHSD . . . . .	1213
Uniform . . . . .	1214
uniroot . . . . .	1216
update . . . . .	1217
update.formula . . . . .	1218
var.test . . . . .	1219
varimax . . . . .	1220
vcov . . . . .	1222
Weibull . . . . .	1222
weighted.mean . . . . .	1224
weighted.residuals . . . . .	1224
wilcox.test . . . . .	1225
Wilcoxon . . . . .	1228
window . . . . .	1230
xtabs . . . . .	1231
<b>8 The tools package</b> . . . . .	<b>1233</b>
buildVignettes . . . . .	1233
checkFF . . . . .	1234
checkMD5sums . . . . .	1235
checkTnF . . . . .	1235
checkVignettes . . . . .	1236
codoc . . . . .	1237
delimMatch . . . . .	1239
fileutils . . . . .	1240
getDepList . . . . .	1241
installFoundDepends . . . . .	1243
makeLazyLoading . . . . .	1244
md5sum . . . . .	1245

package.dependencies . . . . .	1245
QC . . . . .	1246
Rdindex . . . . .	1247
Rdutils . . . . .	1248
read.00Index . . . . .	1250
texi2dvi . . . . .	1251
tools-deprecated . . . . .	1251
undoc . . . . .	1252
vignetteDepends . . . . .	1253
<b>9 The <code>utils</code> package</b> . . . . .	<b>1255</b>
apropos . . . . .	1255
BATCH . . . . .	1256
browseEnv . . . . .	1257
browseURL . . . . .	1259
bug.report . . . . .	1259
capture.output . . . . .	1262
citation . . . . .	1263
citEntry . . . . .	1264
close.socket . . . . .	1266
compareVersion . . . . .	1266
COMPILE . . . . .	1267
data . . . . .	1268
dataentry . . . . .	1270
debugger . . . . .	1272
demo . . . . .	1274
download.file . . . . .	1275
edit . . . . .	1276
edit.data.frame . . . . .	1278
example . . . . .	1279
file.edit . . . . .	1281
fix . . . . .	1282
getAnywhere . . . . .	1283
getFromNamespace . . . . .	1284
getS3method . . . . .	1285
head . . . . .	1286
help . . . . .	1287
help.search . . . . .	1290
help.start . . . . .	1292
index.search . . . . .	1293
INSTALL . . . . .	1294
installed.packages . . . . .	1296
LINK . . . . .	1297
ls.str . . . . .	1297
make.packages.html . . . . .	1299
make.socket . . . . .	1299
menu . . . . .	1301
methods . . . . .	1301
nsl . . . . .	1303

object.size . . . . .	1303
package.skeleton . . . . .	1304
packageDescription . . . . .	1305
packageStatus . . . . .	1306
page . . . . .	1308
person . . . . .	1308
PkgUtils . . . . .	1309
prompt . . . . .	1310
promptData . . . . .	1312
read.fortran . . . . .	1313
read.fwf . . . . .	1314
read.socket . . . . .	1316
recover . . . . .	1317
REMOVE . . . . .	1319
remove.packages . . . . .	1319
RHOME . . . . .	1320
Rprof . . . . .	1320
Rtangle . . . . .	1321
RweaveLatex . . . . .	1322
savehistory . . . . .	1324
sessionInfo . . . . .	1325
SHLIB . . . . .	1325
str . . . . .	1326
summaryRprof . . . . .	1328
Sweave . . . . .	1329
SweaveSyntConv . . . . .	1331
toLatex . . . . .	1332
update.packages . . . . .	1333
url.show . . . . .	1335
utils-deprecated . . . . .	1336
vignette . . . . .	1336

# Chapter 1

## The base package

---

`.Device`

*Lists of Open Graphics Devices*

---

### Description

A list of the names of the open graphics devices is stored in `.Devices`. The name of the active device is stored in `.Device`.

---

`.Machine`

*Numerical Characteristics of the Machine*

---

### Description

`.Machine` is a variable holding information on the numerical characteristics of the machine R is running on, such as the largest double or integer and the machine's precision.

### Usage

`.Machine`

### Details

The algorithm is based on Cody's (1988) subroutine MACHAR.

**Value**

A list with components (for simplicity, the prefix “double” is omitted in the explanations)

<code>double.eps</code>	the smallest positive floating-point number $x$ such that $1 + x \neq 1$ . It equals $\text{base}^{\text{ulp.digits}}$ if either <code>base</code> is 2 or <code>rounding</code> is 0; otherwise, it is $(\text{base}^{\text{ulp.digits}}) / 2$ .
<code>double.neg.eps</code>	a small positive floating-point number $x$ such that $1 - x \neq 1$ . It equals $\text{base}^{\text{neg.ulp.digits}}$ if <code>base</code> is 2 or <code>round</code> is 0; otherwise, it is $(\text{base}^{\text{neg.ulp.digits}}) / 2$ . As <code>neg.ulp.digits</code> is bounded below by $-(\text{digits} + 3)$ , <code>neg.eps</code> may not be the smallest number that can alter 1 by subtraction.
<code>double.xmin</code>	the smallest non-vanishing normalized floating-point power of the radix, i.e., $\text{base}^{\text{min.exp}}$ .
<code>double.xmax</code>	the largest finite floating-point number. Typically, it is equal to $(1 - \text{neg.eps}) * \text{base}^{\text{max.exp}}$ , but on some machines it is only the second, or perhaps third, largest number, being too small by 1 or 2 units in the last digit of the significand.
<code>double.base</code>	the radix for the floating-point representation
<code>double.digits</code>	the number of base digits in the floating-point significand
<code>double.rounding</code>	the rounding action. 0 if floating-point addition chops; 1 if floating-point addition rounds, but not in the IEEE style; 2 if floating-point addition rounds in the IEEE style; 3 if floating-point addition chops, and there is partial underflow; 4 if floating-point addition rounds, but not in the IEEE style, and there is partial underflow; 5 if floating-point addition rounds in the IEEE style, and there is partial underflow
<code>double.guard</code>	the number of guard digits for multiplication with truncating arithmetic. It is 1 if floating-point arithmetic truncates and more than <code>digits</code> base base digits participate in the post-normalization shift of the floating-point significand in multiplication, and 0 otherwise.
<code>double.ulp.digits</code>	the largest negative integer $i$ such that $1 + \text{base}^i \neq 1$ , except that it is bounded below by $-(\text{digits} + 3)$ .
<code>double.neg.ulp.digits</code>	the largest negative integer $i$ such that $1 - \text{base}^i \neq 1$ , except that it is bounded below by $-(\text{digits} + 3)$ .
<code>double.exponent</code>	the number of bits (decimal places if <code>base</code> is 10) reserved for the representation of the exponent (including the bias or sign) of a floating-point number
<code>double.min.exp</code>	the largest in magnitude negative integer $i$ such that $\text{base}^i$ is positive and normalized.

`double.max.exp` the smallest positive power of base that overflows.  
`integer.max` the largest integer which can be represented.  
`sizeof.long` the number of bytes in a C long type.  
`sizeof.longlong` the number of bytes in a C long long type. Will be zero if there is no such type.  
`sizeof.longdouble` the number of bytes in a C long double type. Will be zero if there is no such type.  
`sizeof.pointer` the number of bytes in a C SEXP type.

### References

Cody, W. J. (1988) MACHAR: A subroutine to dynamically determine machine parameters. *Transactions on Mathematical Software*, **14**, 4, 303–311.

### See Also

[.Platform](#) for details of the platform.

### Examples

```
.Machine  
## or for a neat printout  
noquote(unlist(format(.Machine)))
```

---

.Platform                      *Platform Specific Variables*

---

### Description

.Platform is a list with some details of the platform under which R was built. This provides means to write OS portable R code.

### Usage

```
.Platform
```

### Value

A list with at least the following components:

`OS.type` character, giving the **O**perating **S**ystem (family) of the computer. One of "unix" or "windows".

<code>file.sep</code>	character, giving the <b>file separator</b> , used on your platform, e.g., "/" on Unix alike.
<code>dynlib.ext</code>	character, giving the file name <b>extension</b> of <b>dynamically loadable libraries</b> , e.g., ".dll" on Windows.
<code>GUI</code>	character, giving the type of GUI in use, or "unknown" if no GUI can be assumed.
<code>endian</code>	character, "big" or "little", giving the endianness of the processor in use.

**See Also**

[R.version](#) and [Sys.info](#) give more details about the OS. In particular, `R.version$platform` is the canonical name of the platform under which R was compiled.

[.Machine](#) for details of the arithmetic used, and [system](#) for invoking platform-specific system commands.

**Examples**

```
## Note: this can be done in a system-independent way by file.info()$isdir
if(.Platform$OS.type == "unix") {
  system.test <- function(...) { system(paste("test", ...)) == 0 }
  dir.exists <- function(dir) sapply(dir, function(d)system.test("-d", d))
  dir.exists(c(R.home(), "/tmp", "~", "/NO"))# > T T T F
}
```

---

.Script

*Scripting Language Interface*

---

**Description**

Run a script through its interpreter with given arguments.

**Usage**

```
.Script(interpreter, script, args, ...)
```

**Arguments**

<code>interpreter</code>	a character string naming the interpreter for the script.
<code>script</code>	a character string with the base file name of the script, which must be located in the 'interpreter' subdirectory of 'R_HOME/share'.
<code>args</code>	a character string giving the arguments to pass to the script.
<code>...</code>	further arguments to be passed to <a href="#">system</a> when invoking the interpreter on the script.

**Note**

This function is for R internal use only.

**Examples**

```
.Script("perl", "message-Examples.pl",  
       paste("tools", system.file("R-ex", package = "tools")))
```

---

abbreviate

*Abbreviate Strings*

---

**Description**

Abbreviate strings to at least `minlength` characters, such that they remain *unique* (if they were).

**Usage**

```
abbreviate(names.arg, minlength = 4, use.classes = TRUE,  
          dot = FALSE)
```

**Arguments**

<code>names.arg</code>	a vector of names to be abbreviated.
<code>minlength</code>	the minimum length of the abbreviations.
<code>use.classes</code>	logical (currently ignored by R).
<code>dot</code>	logical; should a dot (".") be appended?

**Details**

The algorithm used is similar to that of `S`. First spaces at the beginning of the word are stripped. Then any other spaces are stripped. Next lower case vowels are removed followed by lower case consonants. Finally if the abbreviation is still longer than `minlength` upper case letters are stripped.

Letters are always stripped from the end of the word first. If an element of `names.arg` contains more than one word (words are separated by space) then at least one letter from each word will be retained. If a single string is passed it is abbreviated in the same manner as a vector of strings.

Missing (NA) values are not abbreviated.

If `use.classes` is `FALSE` then the only distinction is to be between letters and space. This has NOT been implemented.

**Value**

A character vector containing abbreviations for the strings in its first argument. Duplicates in the original `names.arg` will be given identical abbreviations. If any non-duplicated elements have the same `minlength` abbreviations then `minlength` is incremented by one and new abbreviations are found for those elements only. This process is repeated until all unique elements of `names.arg` have unique abbreviations.

The character version of `names.arg` is attached to the returned value as a `names` argument.

**See Also**

[substr](#).

**Examples**

```
x <- c("abcd", "efgh", "abce")
abbreviate(x, 2)

(st.abb <- abbreviate(state.name, 2))
table(nchar(st.abb))# out of 50, 3 need 4 letters
```

---

abs

*Miscellaneous Mathematical Functions*

---

**Description**

These functions compute miscellaneous mathematical functions. The naming follows the standard for computer languages such as C or Fortran.

**Usage**

```
abs(x)
sqrt(x)
```

**Arguments**

`x` a numeric or [complex](#) vector or array.

**Details**

These are generic functions: methods can be defined for them individually or via the [Math](#) group generic. For complex arguments (and the default method), `z`, `abs(z) == Mod(z)` and `sqrt(z) == z^0.5`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[Arithmetic](#) for simple, [log](#) for logarithmic, [sin](#) for trigonometric, and [Special](#) for special mathematical functions.

**Examples**

```
require(stats) # for spline
xx <- -9:9
plot(xx, sqrt(abs(xx)), col = "red")
lines(spline(xx, sqrt(abs(xx)), n=101), col = "pink")
```

---

agrep

*Approximate String Matching (Fuzzy Matching)*


---

**Description**

Searches for approximate matches to `pattern` (the first argument) within the string `x` (the second argument) using the Levenshtein edit distance.

**Usage**

```
agrep(pattern, x, ignore.case = FALSE, value = FALSE, max.distance = 0.1)
```

**Arguments**

<code>pattern</code>	a non-empty character string to be matched ( <i>not</i> a regular expression!)
<code>x</code>	character vector where matches are sought.
<code>ignore.case</code>	if FALSE, the pattern matching is <i>case sensitive</i> and if TRUE, case is ignored during matching.
<code>value</code>	if FALSE, a vector containing the (integer) indices of the matches determined is returned and if TRUE, a vector containing the matching elements themselves is returned.
<code>max.distance</code>	Maximum distance allowed for a match. Expressed either as integer, or as a fraction of the pattern length (will be replaced by the smallest integer not less than the corresponding fraction), or a list with possible components <p><b>all:</b> maximal (overall) distance</p> <p><b>insertions:</b> maximum number/fraction of insertions</p> <p><b>deletions:</b> maximum number/fraction of deletions</p> <p><b>substitutions:</b> maximum number/fraction of substitutions</p> <p>If <code>all</code> is missing, it is set to 10%, the other components default to <code>all</code>. The component names can be abbreviated.</p>

**Details**

The Levenshtein edit distance is used as measure of approximateness: it is the total number of insertions, deletions and substitutions required to transform one string into another.

The function is a simple interface to the `apse` library developed by Jarkko Hietaniemi (also used in the Perl `String::Approx` module).

**Value**

Either a vector giving the indices of the elements that yielded a match, of, if `value` is `TRUE`, the matched elements.

**Author(s)**

David Meyer (David.Meyer@wu-wien.ac.at) (based on C code by Jarkko Hietaniemi); modifications by Kurt Hornik.

**See Also**

[grep](#)

**Examples**

```
agrep("lasy", "1 lazy 2")
agrep("lasy", "1 lazy 2", max = list(sub = 0))
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2)
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2, value = TRUE)
agrep("laysy", c("1 lazy", "1", "1 LAZY"), max = 2, ignore.case = TRUE)
```

---

all

*Are All Values True?*

---

**Description**

Given a set of logical vectors, are all of the values true?

**Usage**

```
all(..., na.rm = FALSE)
```

**Arguments**

`...` one or more logical vectors.  
`na.rm` logical. If true NA values are removed before the result is computed.

**Details**

This is a generic function: methods can be defined for it directly or via the [Summary](#) group generic.

**Value**

Given a sequence of logical arguments, a logical value indicating whether or not all of the elements of `x` are TRUE.

The value returned is TRUE if all the values in `x` are TRUE, and FALSE if any the values in `x` are FALSE.

If `x` consists of a mix of TRUE and NA values, then value is NA.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[any](#), the “complement” of all, and [stopifnot](#)(\*) which is an all(\*) “insurance”.

**Examples**

```
range(x <- sort(round(rnorm(10) - 1.2,1)))
if(all(x < 0)) cat("all x values are negative\n")
```

---

all.equal

*Test if Two Objects are (Nearly) Equal*


---

**Description**

`all.equal(x, y)` is a utility to compare R objects `x` and `y` testing “near equality”. If they are different, comparison is still made to some extent, and a report of the differences is returned. Don’t use `all.equal` directly in `if` expressions—either use [identical](#) or combine the two, as shown in the documentation for [identical](#).

**Usage**

```
all.equal(target, current, ...)

## S3 method for class 'numeric':
all.equal(target, current,
          tolerance = .Machine$double.eps ^ 0.5,
          scale = NULL, ...)

attr.all.equal(target, current, ...)
```

**Arguments**

target	R object.
current	other R object, to be compared with target.
...	Further arguments for different methods, notably the following two, for numerical comparison:
tolerance	numeric $\geq 0$ . Differences smaller than tolerance are not considered.
scale	numeric scalar $> 0$ (or NULL). See Details.

**Details**

There are several methods available, most of which are dispatched by the default method, see `methods("all.equal")`. `all.equal.list` and `all.equal.language` provide comparison of recursive objects.

Numerical comparisons for `scale = NULL` (the default) are done by first computing the mean absolute difference of the two numerical vectors. If this is smaller than `tolerance` or not finite, absolute differences are used, otherwise relative differences scaled by the mean absolute difference.

If `scale` is positive, absolute comparisons are after scaling (dividing) by `scale`.

For complex arguments, the modulus `Mod` of the difference is used: `all.numeric.numeric` is called so arguments `tolerance` and `scale` are available.

`attr.all.equal` is used for comparing `attributes`, returning NULL or character.

**Value**

Either TRUE or a vector of `mode` "character" describing the differences between `target` and `current`.

Numerical differences are reported by relative error

**References**

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer (for =).

**See Also**

`identical`, `==`, and `all` for exact equality testing.

**Examples**

```
all.equal(pi, 355/113) # not precise enough (default tol) > relative error

d45 <- pi*(1/4 + 1:10)
stopifnot(
  all.equal(tan(d45), rep(1,10)) # TRUE, but
  all      (tan(d45) == rep(1,10)) # FALSE, since not exactly
  all.equal(tan(d45), rep(1,10), tol=0) # to see difference

all.equal(options(), .Options)
all.equal(options(), as.list(.Options))# TRUE
```

```
.Options $ myopt <- TRUE
all.equal(options(), as.list(.Options))
rm(.Options)
```

---

`all.names`*Find All Names in an Expression*

---

### Description

Return a character vector containing all the names which occur in an expression or call.

### Usage

```
all.names(expr, functions = TRUE, max.names = 200, unique = FALSE)
```

```
all.vars(expr, functions = FALSE, max.names = 200, unique = TRUE)
```

### Arguments

<code>expr</code>	an expression or call from which the names are to be extracted.
<code>functions</code>	a logical value indicating whether function names should be included in the result.
<code>max.names</code>	the maximum number of names to be returned.
<code>unique</code>	a logical value which indicates whether duplicate names should be removed from the value.

### Details

These functions differ only in the default values for their arguments.

### Value

A character vector with the extracted names.

### Examples

```
all.names(expression(sin(x+y)))
all.vars(expression(sin(x+y)))
```

---

any

*Are Some Values True?*

---

### Description

Given a set of logical vectors, are any of the values true?

### Usage

```
any(..., na.rm = FALSE)
```

### Arguments

`...` one or more logical vectors.  
`na.rm` logical. If true NA values are removed before the result is computed.

### Details

This is a generic function: methods can be defined for it directly or via the [Summary](#) group generic.

### Value

Given a sequence of logical arguments, a logical value indicating whether or not any of the elements of `x` are TRUE.

The value returned is TRUE if any the values in `x` are TRUE, and FALSE if all the values in `x` are FALSE.

If `x` consists of a mix of FALSE and NA values, the value is NA.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[all](#), the “complement” of any.

### Examples

```
range(x <- sort(round(rnorm(10) - 1.2,1)))  
if(any(x < 0)) cat("x contains negative values\n")
```

---

aperm	<i>Array Transposition</i>
-------	----------------------------

---

**Description**

Transpose an array by permuting its dimensions and optionally resizing it.

**Usage**

```
aperm(a, perm, resize = TRUE)
```

**Arguments**

<code>a</code>	the array to be transposed.
<code>perm</code>	the subscript permutation vector, which must be a permutation of the integers $1:n$ , where $n$ is the number of dimensions of <code>a</code> . The default is to reverse the order of the dimensions.
<code>resize</code>	a flag indicating whether the vector should be resized as well as having its elements reordered (default <code>TRUE</code> ).

**Value**

A transposed version of array `a`, with subscripts permuted as indicated by the array `perm`. If `resize` is `TRUE`, the array is reshaped as well as having its elements permuted, the `dimnames` are also permuted; if `FALSE` then the returned object has the same dimensions as `a`, and the `dimnames` are dropped.

The function `t` provides a faster and more convenient way of transposing matrices.

**Author(s)**

Jonathan Rougier, [J.C.Rougier@durham.ac.uk](mailto:J.C.Rougier@durham.ac.uk) did the faster C implementation.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[t](#), to transpose matrices.

**Examples**

```
# interchange the first two subscripts on a 3-way array x
x <- array(1:24, 2:4)
xt <- aperm(x, c(2,1,3))
stopifnot(t(xt[, ,2]) == x[, ,2],
          t(xt[, ,3]) == x[, ,3],
          t(xt[, ,4]) == x[, ,4])
```

---

 append

*Vector Merging*


---

**Description**

Add elements to a vector.

**Usage**

```
append(x, values, after = length(x))
```

**Arguments**

`x` the vector to be modified.  
`values` to be included in the modified vector.  
`after` a subscript, after which the values are to be appended.

**Value**

A vector containing the values in `x` with the elements of `values` appended after the specified element of `x`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```
append(1:5, 0:1, after=3)
```

---

`apply`*Apply Functions Over Array Margins*

---

**Description**

Returns a vector or array or list of values obtained by applying a function to margins of an array.

**Usage**

```
apply(X, MARGIN, FUN, ...)
```

**Arguments**

<code>X</code>	the array to be used.
<code>MARGIN</code>	a vector giving the subscripts which the function will be applied over. 1 indicates rows, 2 indicates columns, <code>c(1, 2)</code> indicates rows and columns.
<code>FUN</code>	the function to be applied. In the case of functions like <code>+</code> , <code>%*%</code> , etc., the function name must be quoted.
<code>...</code>	optional arguments to <code>FUN</code> .

**Details**

If `X` is not an array but has a `dimension` attribute, `apply` attempts to coerce it to an array via `as.matrix` if it is two-dimensional (e.g., data frames) or via `as.array`.

**Value**

If each call to `FUN` returns a vector of length `n`, then `apply` returns an array of dimension `c(n, dim(X)[MARGIN])` if `n > 1`. If `n` equals 1, `apply` returns a vector if `MARGIN` has length 1 and an array of dimension `dim(X)[MARGIN]` otherwise. If `n` is 0, the result has length 0 but not necessarily the “correct” dimension.

If the calls to `FUN` return vectors of different lengths, `apply` returns a list of length `dim(X)[MARGIN]`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[lapply](#), [tapply](#), and convenience functions [sweep](#) and [aggregate](#).

**Examples**

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))

stopifnot( apply(x,2, is.vector)) # not ok in R <= 0.63.2

## Sort the columns of a matrix
apply(x, 2, sort)

##- function with extra args:
cave <- function(x, c1,c2) c(mean(x[c1]),mean(x[c2]))
apply(x,1, cave, c1="x1", c2=c("x1","x2"))

ma <- matrix(c(1:4, 1, 6:8), nr = 2)
ma
apply(ma, 1, table) #--> a list of length 2
apply(ma, 1, quantile)# 5 x n matrix with rownames

stopifnot(dim(ma) == dim(apply(ma, 1:2, sum)))## wasn't ok before R 0.63.1
```

---

approxfun

*Interpolation Functions*


---

**Description**

Return a list of points which linearly interpolate given data points, or a function performing the linear (or constant) interpolation.

**Usage**

```
approx (x, y = NULL, xout, method="linear", n=50,
        yleft, yright, rule = 1, f = 0, ties = mean)

approxfun(x, y = NULL, method="linear",
          yleft, yright, rule = 1, f = 0, ties = mean)
```

**Arguments**

**x, y** vectors giving the coordinates of the points to be interpolated. Alternatively a single plotting structure can be specified: see [xy.coords](#).

**xout** an optional set of values specifying where interpolation is to take place.

method	specifies the interpolation method to be used. Choices are "linear" or "constant".
n	If <code>xout</code> is not specified, interpolation takes place at <code>n</code> equally spaced points spanning the interval $[\min(x), \max(x)]$ .
yleft	the value to be returned when input <code>x</code> values are less than $\min(x)$ . The default is defined by the value of <code>rule</code> given below.
yright	the value to be returned when input <code>x</code> values are greater than $\max(x)$ . The default is defined by the value of <code>rule</code> given below.
rule	an integer describing how interpolation is to take place outside the interval $[\min(x), \max(x)]$ . If <code>rule</code> is 1 then NAs are returned for such points and if it is 2, the value at the closest data extreme is used.
f	For <code>method="constant"</code> a number between 0 and 1 inclusive, indicating a compromise between left- and right-continuous step functions. If <code>y0</code> and <code>y1</code> are the values to the left and right of the point then the value is $y0 * (1-f) + y1 * f$ so that <code>f=0</code> is right-continuous and <code>f=1</code> is left-continuous.
ties	Handling of tied <code>x</code> values. Either a function with a single vector argument returning a single number result or the string "ordered".

### Details

The inputs can contain missing values which are deleted, so at least two complete (`x`, `y`) pairs are required. If there are duplicated (tied) `x` values and `ties` is a function it is applied to the `y` values for each distinct `x` value. Useful functions in this context include [mean](#), [min](#), and [max](#). If `ties="ordered"` the `x` values are assumed to be already ordered. The first `y` value will be used for interpolation to the left and the last one for interpolation to the right.

### Value

`approx` returns a list with components `x` and `y`, containing `n` coordinates which interpolate the given data points according to the `method` (and `rule`) desired.

The function `approxfun` returns a function performing (linear or constant) interpolation of the given data points. For a given set of `x` values, this function will return the corresponding interpolated values. This is often more useful than `approx`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[spline](#) and [splinefun](#) for spline interpolation.

**Examples**

```

x <- 1:10
y <- rnorm(10)
par(mfrow = c(2,1))
plot(x, y, main = "approx(.) and approxfun(.)")
points(approx(x, y), col = 2, pch = "*")
points(approx(x, y, method = "constant"), col = 4, pch = "*")

f <- approxfun(x, y)
curve(f(x), 0, 10, col = "green")
points(x, y)
is.function(fc <- approxfun(x, y, method = "const")) # TRUE
curve(fc(x), 0, 10, col = "darkblue", add = TRUE)

## Show treatment of 'ties' :

x <- c(2,2:4,4,4,5,5,7,7,7)
y <- c(1:6, 5:4, 3:1)
approx(x,y, xout=x)$y # warning
(ay <- approx(x,y, xout=x, ties = "ordered"))$y
stopifnot(ay == c(2,2,3,6,6,6,4,4,1,1,1))
approx(x,y, xout=x, ties = min)$y
approx(x,y, xout=x, ties = max)$y

```

---

args

*Argument List of a Function*


---

**Description**

Displays the argument names and corresponding default values of a function.

**Usage**

```
args(name)
```

**Arguments**

name	an interpreted function. If name is a character string then the function with that name is found and used.
------	--

**Details**

This function is mainly used interactively. For programming, use `formals` instead.

**Value**

A function with identical formal argument list but an empty body if given an interpreted function; NULL in case of a variable or primitive (non-interpreted) function.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[formals](#), [help](#).

**Examples**

```
args(c)           # -> NULL (c is a 'primitive' function)
args(graphics::plot.default)
```

---

Arithmetic

*Arithmetic Operators*

---

**Description**

These binary operators perform arithmetic on numeric or complex vectors (or objects which can be coerced to them).

**Usage**

```
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
```

**Arguments**

`x`, `y` numeric or complex vectors or objects which can be coerced to such, or other objects for which methods have been written.

**Details**

The binary arithmetic operators are generic functions: methods can be written for them individually or via the [Ops](#) group generic function.

If applied to arrays the result will be an array if this is sensible (for example it will not if the recycling rule has been invoked).

Logical vectors will be coerced to numeric vectors, FALSE having value 0 and TRUE having value one.

$1 \wedge y$  and  $y \wedge 0$  are 1, *always*.  $x \wedge y$  should also give the proper “limit” result when either argument is infinite (i.e., +- [Inf](#)).

Objects such as arrays or time-series can be operated on this way provided they are conformable.

**Value**

They return numeric vectors containing the result of the element by element operations. The elements of shorter vectors are recycled as necessary (with a [warning](#) when they are recycled only *fractionally*). The operators are + for addition, - for subtraction, \* for multiplication, / for division and ^ for exponentiation.

%% indicates  $x \bmod y$  and %/% indicates integer division. It is guaranteed that  $x == (x \% \% y) + y * (x \% \% y)$  unless  $y == 0$  where the result is [NA](#) or [NaN](#) (depending on the [typeof](#) of the arguments).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[sqrt](#) for miscellaneous and [Special](#) for special mathematical functions.

[Syntax](#) for operator precedence.

**Examples**

```
x <- -1:12
x + 1
2 * x + 3
x %% 2 #-- is periodic
x %/% 5
```

---

array

*Multi-way Arrays*


---

**Description**

Creates or tests for arrays.

**Usage**

```
array(data = NA, dim = length(data), dimnames = NULL)
as.array(x)
is.array(x)
```

**Arguments**

**data** a vector (including a list) giving data to fill the array.

**dim** the dim attribute for the array to be created, that is a vector of length one or more giving the maximal indices in each dimension.

`dimnames` the names for the dimensions. This is a list with one component for each dimension, either NULL or a character vector of the length given by `dim` for that dimension. The list can be names, and the names will be used as names for the dimensions.

`x` an R object.

### Value

`array` returns an array with the extents specified in `dim` and naming information in `dimnames`. The values in `data` are taken to be those in the array with the leftmost subscript moving fastest. If there are too few elements in `data` to fill the array, then the elements in `data` are recycled. If `data` has length zero, NA of an appropriate type is used for atomic vectors (0 for raw vectors) and NULL for lists.

`as.array()` coerces its argument to be an array by attaching a `dim` attribute to it. It also attaches `dimnames` if `x` has `names`. The sole purpose of this is to make it possible to access the `dim[names]` attribute at a later time.

`is.array` returns TRUE or FALSE depending on whether its argument is an array (i.e., has a `dim` attribute) or not. It is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[aperm](#), [matrix](#), [dim](#), [dimnames](#).

### Examples

```
dim(as.array(letters))
array(1:3, c(2,4)) # recycle 1:3 "2 2/3 times"
#      [,1] [,2] [,3] [,4]
#[1,]  1   3   2   1
#[2,]  2   1   3   2
```

---

as.data.frame                    *Coerce to a Data Frame*

---

### Description

Functions to check if an object is a data frame, or coerce it if possible.

### Usage

```
as.data.frame(x, row.names = NULL, optional = FALSE)
is.data.frame(x)
```

**Arguments**

<code>x</code>	any R object.
<code>row.names</code>	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
<code>optional</code>	logical. If TRUE, setting row names and converting column names (to syntactic names) is optional.

**Details**

`as.data.frame` is a generic function with many methods, and users and packages can supply further methods.

If a list is supplied, each element is converted to a column in the data frame. Similarly, each column of a matrix is converted separately. This can be overridden if the object has a class which has a method for `as.data.frame`: two examples are matrices of class `"model.matrix"` (which are included as a single column) and list objects of class `"POSIXlt"` which are coerced to class `"POSIXct"`.

As from R 1.9.0 arrays can be converted. One-dimensional arrays are treated like vectors and two-dimensional arrays like matrices. Arrays with more than two dimensions are converted to matrices by ‘flattening’ all dimensions after the first and creating suitable column labels.

Character variables are converted to factor columns unless protected by [I](#).

If a data frame is supplied, all classes preceding `"data.frame"` are stripped, and the row names are changed if that argument is supplied.

If `row.names = NULL`, row names are constructed from the names or `dimnames` of `x`, otherwise are the integer sequence starting at one. Few of the methods check for duplicated row names. Names are removed from vector columns unless [I](#).

**Value**

`as.data.frame` returns a data frame, normally with all row names `" "` if `optional = TRUE`.

`is.data.frame` returns TRUE if its argument is a data frame (that is, has `"data.frame"` amongst its classes) and FALSE otherwise.

**Note**

In versions of R prior to 1.4.0 logical columns were converted to factors (as in S3 but not S4).

**References**

Chambers, J. M. (1992) *Data for models*. Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[data.frame](#)

---

as.environment      *Coerce to an Environment Object*

---

### Description

Converts a number or a character string to the corresponding environment on the search path.

### Usage

```
as.environment(object)
```

### Arguments

`object`      the object to convert. If it is already an environment, just return it. If it is a number, return the environment corresponding to that position on the search list. If it is a character string, match the string to the names on the search list.

### Value

The corresponding environment object.

### Author(s)

John Chambers

### See Also

[environment](#) for creation and manipulation, [search](#).

### Examples

```
as.environment(1) ## the global environment
identical(globalenv(), as.environment(1)) ## is TRUE
try(as.environment("package:stats"))      ## stats need not be loaded
```

---

as.function      *Convert Object to Function*

---

### Description

`as.function` is a generic function which is used to convert objects to functions.

`as.function.default` works on a list `x`, which should contain the concatenation of a formal argument list and an expression or an object of mode "[call](#)" which will become the function body. The function will be defined in a specified environment, by default that of the caller.

**Usage**

```
as.function(x, ...)

## Default S3 method:
as.function(x, envir = parent.frame(), ...)
```

**Arguments**

x	object to convert, a list for the default method.
...	additional arguments, depending on object
envir	environment in which the function should be defined

**Value**

The desired function.

**Author(s)**

Peter Dalgaard

**See Also**

[function](#); [alist](#) which is handy for the construction of argument lists, etc.

**Examples**

```
as.function(alist(a=,b=2,a+b))
as.function(alist(a=,b=2,a+b))(3)
```

---

as.POSIX\*

*Date-time Conversion Functions*


---

**Description**

Functions to manipulate objects of classes "POSIXlt" and "POSIXct" representing calendar dates and times (to the nearest second).

**Usage**

```
as.POSIXct(x, tz = "")
as.POSIXlt(x, tz = "")
```

**Arguments**

x	An object to be converted.
tz	A timezone specification to be used for the conversion, <i>if one is required</i> . System-specific, but "" is the current timezone, and "GMT" is UTC (Coordinated Universal Time, in French).

## Details

The `as.POSIX*` functions convert an object to one of the two classes used to represent date/times (calendar dates plus time to the nearest second). They can take convert a wide variety of objects, including objects of the other class and of classes "Date", "date" (from package **date** or **survival**), "chron" and "dates" (from package **chron**) to these classes. Dates are treated as being at midnight UTC.

They can also convert character strings of the formats "2001-02-03" and "2001/02/03" optionally followed by white space and a time in the format "14:52" or "14:52:03". (Formats such as "01/02/03" are ambiguous but can be converted via a format specification by [strptime](#).)

Logical NAs can be converted to either of the classes, but no other logical vectors can be.

## Value

`as.POSIXct` and `as.POSIXlt` return an object of the appropriate class. If `tz` was specified, `as.POSIXlt` will give an appropriate "tzone" attribute.

## Note

If you want to extract specific aspects of a time (such as the day of the week) just convert it to class "POSIXlt" and extract the relevant component(s) of the list, or if you want a character representation (such as a named day of the week) use `format.POSIXlt` or `format.POSIXct`.

If a timezone is needed and that specified is invalid on your system, what happens is system-specific but it will probably be ignored.

## See Also

[DateTimeClasses](#) for details of the classes; [strptime](#) for conversion to and from character representations.

## Examples

```
(z <- Sys.time())           # the current date, as class "POSIXct"
unclass(z)                  # a large integer
floor(unclass(z)/86400)     # the number of days since 1970-01-01
(z <- as.POSIXlt(Sys.time())) # the current date, as class "POSIXlt"
unlist(unclass(z))         # a list shown as a named vector

as.POSIXlt(Sys.time(), "GMT") # the current time in GMT
```

## Description

Change the class of an object to indicate that it should be treated "as is".

**Usage**

```
I(x)
```

**Arguments**

```
x          an object
```

**Details**

Function `I` has two main uses.

- In function `data.frame`. Protecting an object by enclosing it in `I()` in a call to `data.frame` inhibits the conversion of character vectors to factors, and the dropping of names. `I` can also be used to protect objects which are to be added to a data frame, or converted to a data frame *via* `as.data.frame`.

It achieves this by prepending the class "AsIs" to the object's classes. Class "AsIs" has a few of its own methods, including `[, as.data.frame, print` and `format`.

- In function `formula`. There it is used to inhibit the interpretation of operators such as "+", "-", "\*" and "^" as formula operators, so they are used as arithmetical operators. This is interpreted as a symbol by `terms.formula`.

**Value**

A copy of the object with class "AsIs" prepended to the class(es).

**References**

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

`data.frame`, `formula`

---

assign

*Assign a Value to a Name*

---

**Description**

Assign a value to a name in an environment.

**Usage**

```
assign(x, value, pos = -1, envir = as.environment(pos),
       inherits = FALSE, immediate = TRUE)
```

**Arguments**

<code>x</code>	a variable name (given as a quoted string in the function call).
<code>value</code>	a value to be assigned to <code>x</code> .
<code>pos</code>	where to do the assignment. By default, assigns into the current environment. See the details for other possibilities.
<code>envir</code>	the <a href="#">environment</a> to use. See the details section.
<code>inherits</code>	should the enclosing frames of the environment be inspected?
<code>immediate</code>	an ignored compatibility feature.

**Details**

The `pos` argument can specify the environment in which to assign the object in any of several ways: as an integer (the position in the [search](#) list); as the character string name of an element in the search list; or as an [environment](#) (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

`assign` does not dispatch assignment methods, so it cannot be used to set elements of vectors, names, attributes, etc.

Note that assignment to an attached list or data frame changes the attached copy and not the original object: see [attach](#).

**Value**

This function is invoked for its side effect, which is assigning `value` to the variable `x`. If no `envir` is specified, then the assignment takes place in the currently active environment.

If `inherits` is `TRUE`, enclosing environments of the supplied environment are searched until the variable `x` is encountered. The value is then assigned in the environment in which the variable is encountered. If the symbol is not encountered then assignment takes place in the user's workspace (the global environment).

If `inherits` is `FALSE`, assignment takes place in the initial frame of `envir`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[<-](#), [get](#), [exists](#), [environment](#).

**Examples**

```
for(i in 1:6) { #-- Create objects 'r1', 'r2', ... 'r6' --
  nam <- paste("r",i, sep=".")
  assign(nam, 1:i)
}
ls(pat="^r..$")
```

```
##-- Global assignment within a function:
myf <- function(x) {
  innerf <- function(x) assign("Global.res", x^2, env = .GlobalEnv)
  innerf(x+1)
}
myf(3)
Global.res # 16

a <- 1:4
assign("a[1]", 2)
a[1] == 2      #FALSE
get("a[1]") == 2 #TRUE
```

---

assignOps

*Assignment Operators*


---

## Description

Assign a value to a name.

## Usage

```
x <- value
x <<- value
value -> x
value ->> x

x = value
```

## Arguments

x	a variable name (possibly quoted).
value	a value to be assigned to x.

## Details

There are three different assignment operators: two of them have leftwards and rightwards forms.

The operators `<-` and `=` assign into the environment in which they are evaluated. The `<-` can be used anywhere, but the `=` is only allowed at the top level (that is, in the complete expression typed by the user) or as one of the subexpressions in a braced list of expressions.

The operators `<<-` and `->>` cause a search to be made through the environment for an existing definition of the variable being assigned. If such a variable is found then its value is redefined, otherwise assignment takes place globally. Note that their semantics differ from that in the S language, but are useful in conjunction with the scoping rules of R.

In all the assignment operator expressions, `x` can be a name or an expression defining a part of an object to be replaced (e.g., `z[[1]]`). The name does not need to be quoted, though it can be.

The leftwards forms of assignment `<- = <<-` group right to left, the other from left to right.

**Value**

value. Thus one can use `a <- b <- c <- 6`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chamber, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer (for =).

**See Also**

[assign](#), [environment](#).

---

attach

*Attach Set of R Objects to Search Path*

---

**Description**

The database is attached to the R search path. This means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names.

**Usage**

```
attach(what, pos = 2, name = deparse(substitute(what)))
```

**Arguments**

what	“database”. This may currently be a <code>data.frame</code> or <code>list</code> or a R data file created with <a href="#">save</a> .
pos	integer specifying position in <a href="#">search()</a> where to attach.
name	alternative way to specify the database to be attached.

**Details**

When evaluating a variable or function name R searches for that name in the databases listed by [search](#). The first name of the appropriate type is used.

By attaching a data frame to the search path it is possible to refer to the variables in the data frame by their names alone, rather than as components of the data frame (eg in the example below, `height` rather than `women$height`).

By default the database is attached in position 2 in the search path, immediately after the user’s workspace and before all previously loaded packages and previously attached databases. This can be altered to attach later in the search path with the `pos` option, but you cannot attach at `pos=1`.

Note that by default assignment is not performed in an attached database. Attempting to modify a variable or function in an attached database will actually create a modified version in the user’s workspace (the R global environment). If you use [assign](#) to assign to an attached list or data frame, you only alter the attached copy, not the original object. For this reason `attach` can lead to confusion.

**Value**

The `environment` is returned invisibly with a "name" attribute.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`library`, `detach`, `search`, `objects`, `environment`, `with`.

**Examples**

```
summary(women$height) ## refers to variable 'height' in the data frame
attach(women)
summary(height)      ## The same variable now available by name
height <- height*2.54 ## Don't do this. It creates a new variable
detach("women")
summary(height)      ## The new variable created by modifying 'height'
rm(height)
```

---

attr

*Object Attributes*

---

**Description**

Get or set specific attributes of an object.

**Usage**

```
attr(x, which)
attr(x, which) <- value
```

**Arguments**

x	an object whose attributes are to be accessed.
which	a character string specifying which attribute is to be accessed.
value	an object, the new value of the attribute.

**Value**

This function provides access to a single object attribute. The simple form above returns the value of the named attribute. The assignment form causes the named attribute to take the value on the right of the assignment symbol.

The first form first looks for an exact match to `code` amongst the attributed of `x`, then a partial match. If no exact match is found and more than one partial match is found, the result is `NULL`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[attributes](#)

## Examples

```
# create a 2 by 5 matrix
x <- 1:10
attr(x,"dim") <- c(2, 5)
```

---

attributes	<i>Object Attribute Lists</i>
------------	-------------------------------

---

## Description

These functions access an object's attribute list. The first form below returns the object's attribute list. The assignment forms make the list on the right-hand side of the assignment the object's attribute list (if appropriate).

## Usage

```
attributes(obj)
attributes(obj) <- value
mostattributes(obj) <- value
```

## Arguments

obj	an object
value	an appropriate attribute list, or NULL.

## Details

The `mostattributes` assignment takes special care for the `dim`, `names` and `dimnames` attributes, and assigns them only when that is valid whereas as `attributes` assignment would give an error in that case.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[attr.](#)

**Examples**

```
x <- cbind(a=1:3, pi=pi) # simple matrix w/ dimnames
attributes(x)

## strip an object's attributes:
attributes(x) <- NULL
x # now just a vector of length 6

mostattributes(x) <- list(mycomment = "really special", dim = 3:2,
  dimnames = list(LETTERS[1:3], letters[1:5]), names = paste(1:6))
x # dim(), but not {dim}names
```

---

 autoload

*On-demand Loading of Packages*


---

**Description**

autoload creates a promise-to-evaluate autoloader and stores it with name name in .AutoloadEnv environment. When R attempts to evaluate name, autoloader is run, the package is loaded and name is re-evaluated in the new package's environment. The result is that R behaves as if file was loaded but it does not occupy memory.

.Autoloaded contains the “base names” of the packages for which autoloading has been promised.

**Usage**

```
autoload(name, package, reset = FALSE, ...)
autoloader(name, package, ...)
```

```
.AutoloadEnv
.Autoloaded
```

**Arguments**

name	string giving the name of an object.
package	string giving the name of a package containing the object.
reset	logical: for internal use by autoloader.
...	other arguments to <a href="#">library</a> .

**Value**

This function is invoked for its side-effect. It has no return value as of R 1.7.0.

**See Also**

[delay](#), [library](#)

**Examples**

```

require(stats)
autoload("interpSpline", "splines")
search()
ls("Autoloads")
.Autoloaded

x <- sort(rnorm(12))
y <- x^2
is <- interpSpline(x,y)
search() ## now has splines
detach("package:splines")
search()
is2 <- interpSpline(x,y+x)
search() ## and again
detach("package:splines")

```

---

backsolve

*Solve an Upper or Lower Triangular System*


---

**Description**

Solves a system of linear equations where the coefficient matrix is upper or lower triangular.

**Usage**

```

backsolve(r, x, k= ncol(r), upper.tri = TRUE, transpose = FALSE)
forwardsolve(l, x, k= ncol(l), upper.tri = FALSE, transpose = FALSE)

```

**Arguments**

<code>r, l</code>	an upper (or lower) triangular matrix giving the coefficients for the system to be solved. Values below (above) the diagonal are ignored.
<code>x</code>	a matrix whose columns give “right-hand sides” for the equations.
<code>k</code>	The number of columns of <code>r</code> and rows of <code>x</code> to use.
<code>upper.tri</code>	logical; if TRUE (default), the <i>upper triangular</i> part of <code>r</code> is used. Otherwise, the lower one.
<code>transpose</code>	logical; if TRUE, solve $r' * y = x$ for $y$ , i.e., <code>t(r) %*% y == x</code> .

**Value**

The solution of the triangular system. The result will be a vector if `x` is a vector and a matrix if `x` is a matrix.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

## See Also

[chol](#), [qr](#), [solve](#).

## Examples

```
## upper triangular matrix 'r':
r <- rbind(c(1,2,3),
           c(0,1,1),
           c(0,0,2))
( y <- backsolve(r, x <- c(8,4,2)) ) # -1 3 1
r %*% y # == x = (8,4,2)
backsolve(r, x, transpose = TRUE) # 8 -12 -5
```

---

base-deprecated      *Deprecated Functions in Base package*

---

## Description

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as the next release.

## Usage

## Details

The original help page for these functions is often available at `help("oldName-deprecated")` (note the quotes). Functions in packages other than the base package are listed in `help("pkg-deprecated")`.

## See Also

[Deprecated](#)

---

basename	<i>Manipulate File Paths</i>
----------	------------------------------

---

### Description

basename removes all of the path up to the last path separator (if any).

dirname returns the part of the path up to (but excluding) the last path separator, or "." if there is no path separator.

### Usage

```
basename(path)
dirname(path)
```

### Arguments

path            character vector, containing path names.

### Details

For dirname tilde expansion is done: see the description of [path.expand](#).

Trailing file separators are removed before dissecting the path, and for dirname any trailing file separators are removed from the result.

### Value

A character vector of the same length as path. A zero-length input will give a zero-length output with no error (unlike R < 1.7.0).

### See Also

[file.path](#), [path.expand](#).

### Examples

```
basename(file.path("", "p1", "p2", "p3", c("file1", "file2")))
dirname(file.path("", "p1", "p2", "p3", "filename"))
```

Bessel

*Bessel Functions***Description**

Bessel Functions of integer and fractional order, of first and second kind,  $J_\nu$  and  $Y_\nu$ , and Modified Bessel functions (of first and third kind),  $I_\nu$  and  $K_\nu$ .

`gammaCody` is the ( $\Gamma$ ) function as from the `Specfun` package and originally used in the Bessel code.

**Usage**

```
besselI(x, nu, expon.scaled = FALSE)
besselK(x, nu, expon.scaled = FALSE)
besselJ(x, nu)
bessely(x, nu)
gammaCody(x)
```

**Arguments**

`x` numeric,  $\geq 0$ .

`nu` numeric; The *order* (maybe fractional!) of the corresponding Bessel function.

`expon.scaled` logical; if TRUE, the results are exponentially scaled in order to avoid overflow ( $I_\nu$ ) or underflow ( $K_\nu$ ), respectively.

**Details**

The underlying C code stems from *Netlib* ([http://www.netlib.org/specfun/r\[ijky\]besl](http://www.netlib.org/specfun/r[ijky]besl)).

If `expon.scaled = TRUE`,  $e^{-x}I_\nu(x)$ , or  $e^xK_\nu(x)$  are returned.

`gammaCody` may be somewhat faster but less precise and/or robust than R's standard `gamma`. It is here for experimental purpose mainly, and *may be defunct very soon*.

For  $\nu < 0$ , formulae 9.1.2 and 9.6.2 from the reference below are applied (which is probably suboptimal), unless for `besselK` which is symmetric in `nu`.

**Value**

Numeric vector of the same length of `x` with the (scaled, if `expon.scale=TRUE`) values of the corresponding Bessel function.

**Author(s)**

Original Fortran code: W. J. Cody, Argonne National Laboratory  
 Translation to C and adaption to R: Martin Maechler <maechler@stat.math.ethz.ch.>

## References

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. Dover, New York; Chapter 9: Bessel Functions of Integer Order.

## See Also

Other special mathematical functions, such as [gamma](#),  $\Gamma(x)$ , and [beta](#),  $B(x)$ .

## Examples

```
nus <- c(0:5, 10, 20)

x <- seq(0, 4, len = 501)
plot(x, x, ylim = c(0, 6), ylab = "", type = "n",
     main = "Bessel Functions I_nu(x)")
for(nu in nus) lines(x, besseli(x, nu=nu), col = nu+2)
legend(0, 6, legend = paste("nu=", nus), col = nus+2, lwd = 1)

x <- seq(0, 40, len=801); y1 <- c(-.8, .8)
plot(x, x, ylim = y1, ylab = "", type = "n",
     main = "Bessel Functions J_nu(x)")
for(nu in nus) lines(x, besselj(x, nu=nu), col = nu+2)
legend(32,-.18, legend = paste("nu=", nus), col = nus+2, lwd = 1)

## Negative nu's :
xx <- 2:7
nu <- seq(-10, 9, len = 2001)
op <- par(lab = c(16, 5, 7))
matplot(nu, t(outer(xx, nu, besseli)), type = "l", ylim = c(-50, 200),
        main = expression(paste("Bessel ", I[nu](x), " for fixed ", x,
                                ", as ", f(nu))),
        xlab = expression(nu))
abline(v=0, col = "light gray", lty = 3)
legend(5, 200, legend = paste("x=", xx), col=seq(xx), lty=seq(xx))
par(op)

x0 <- 2^(-20:10)
plot(x0, x0^-8, log="xy", ylab="", type="n",
     main = "Bessel Functions J_nu(x) near 0\n log - log scale")
for(nu in sort(c(nus, nus+.5)))
  lines(x0, besselj(x0, nu=nu), col = nu+2)
legend(3, 1e50, legend = paste("nu=", paste(nus, nus+.5, sep=",")),
      col = nus + 2, lwd = 1)

plot(x0, x0^-8, log="xy", ylab="", type="n",
     main = "Bessel Functions K_nu(x) near 0\n log - log scale")
for(nu in sort(c(nus, nus+.5)))
  lines(x0, besselk(x0, nu=nu), col = nu+2)
legend(3, 1e50, legend = paste("nu=", paste(nus, nus+.5, sep=",")),
      col = nus + 2, lwd = 1)

x <- x[x > 0]
```

```

plot(x, x, ylim=c(1e-18, 1e11), log = "y", ylab = "", type = "n",
     main = "Bessel Functions  K_nu(x)")
for(nu in nus) lines(x, besselK(x, nu=nu), col = nu+2)
legend(0, 1e-5, legend=paste("nu=", nus), col = nus+2, lwd = 1)

yl <- c(-1.6, .6)
plot(x, x, ylim = yl, ylab = "", type = "n",
     main = "Bessel Functions  Y_nu(x)")
for(nu in nus){
  xx <- x[x > .6*nu]
  lines(xx, bessely(xx, nu=nu), col = nu+2)
}
legend(25, -.5, legend = paste("nu=", nus), col = nus+2, lwd = 1)

```

---

body

*Access to and Manipulation of the Body of a Function*


---

## Description

Get or set the body of a function.

## Usage

```

body(fun = sys.function(sys.parent()))
body(fun, envir = parent.frame()) <- value

```

## Arguments

fun	a function object, or see Details.
envir	environment in which the function should be defined.
value	an expression or a list of R expressions.

## Details

For the first form, `fun` can be a character string naming the function to be manipulated, which is searched for from the parent environment. If it is not specified, the function calling `body` is used.

## Value

`body` returns the body of the function specified.

The assignment form sets the body of a function to the list on the right hand side.

## See Also

[alist](#), [args](#), [function](#).

**Examples**

```
body(body)
f <- function(x) x^5
body(f) <- expression(5^x)
## or equivalently body(f) <- list(quote(5^x))
f(3) # = 125
body(f)
```

---

**bquote***Partial substitution in expressions*

---

**Description**

An analogue of the LISP backquote macro. `bquote` quotes its argument except that terms wrapped in `. ( )` are evaluated in the specified `where` environment.

**Usage**

```
bquote(expr, where = parent.frame())
```

**Arguments**

<code>expr</code>	An expression
<code>where</code>	An environment

**Value**

An expression

**See Also**

[quote](#), [substitute](#)

**Examples**

```
a <- 2

bquote(a==a)
quote(a==a)

bquote(a==.(a))
substitute(a==A, list(A=a))

plot(1:10, a*(1:10), main = bquote(a==.(a)))
```

browser *Environment Browser*

---

### Description

Interrupt the execution of an expression and allow the inspection of the environment where `browser` was called from.

### Usage

```
browser()
```

### Details

A call to `browser` causes a pause in the execution of the current expression and runs a copy of the R interpreter which has access to variables local to the environment where the call took place.

Local variables can be listed with `ls`, and manipulated with R expressions typed to this sub-interpreter. The sub-interpreter can be exited by typing `c`. Execution then resumes at the statement following the call to `browser`.

Typing `n` causes the step-through-debugger, to start and it is possible to step through the remainder of the function one line at a time. In this mode `c` will continue to the end of the current context (to the next loop iteration if within a loop).

Typing `Q` quits the current execution and returns you to the top-level prompt.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

### See Also

[debug](#), and [traceback](#) for the stack on error.

---

builtins *Returns the names of all built-in objects*

---

### Description

Return the names of all the built-in objects. These are fetched directly from the symbol table of the R interpreter.

### Usage

```
builtins(internal = FALSE)
```

**Arguments**

`internal` a logical indicating whether only “internal” functions (which can be called via `.Internal`) should be returned.

---

by *Apply a Function to a Data Frame split by Factors*

---

**Description**

Function `by` is an object-oriented wrapper for `tapply` applied to data frames.

**Usage**

```
by(data, INDICES, FUN, ...)
```

**Arguments**

`data` an R object, normally a data frame, possibly a matrix.  
`INDICES` a factor or a list of factors, each of length `nrow(data)`.  
`FUN` a function to be applied to data frame subsets of `data`.  
`...` further arguments to `FUN`.

**Details**

A data frame is split by row into data frames subsetted by the values of one or more factors, and function `FUN` is applied to each subset in turn.

Object `data` will be coerced to a data frame by default.

**Value**

A list of class "by", giving the results for each subset.

**See Also**

[tapply](#)

**Examples**

```
require(stats)
attach(warpbreaks)
by(warpbreaks[, 1:2], tension, summary)
by(warpbreaks[, 1], list(wool=wool, tension=tension), summary)
by(warpbreaks, tension, function(x) lm(breaks ~ wool, data=x))

## now suppose we want to extract the coefficients by group
tmp <- by(warpbreaks, tension, function(x) lm(breaks ~ wool, data=x))
sapply(tmp, coef)

detach("warpbreaks")
```

**Description**

This is a generic function which combines its arguments.

The default method combines its arguments to form a vector. All arguments are coerced to a common type which is the type of the returned value.

**Usage**

```
c(..., recursive=FALSE)
```

**Arguments**

<code>...</code>	objects to be concatenated.
<code>recursive</code>	logical. If <code>recursive=TRUE</code> , the function recursively descends through lists combining all their elements into a vector.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[unlist](#) and [as.vector](#) to produce attribute-free vectors.

**Examples**

```
c(1,7:9)
c(1:5, 10.5, "next")

## append to a list:
ll <- list(A = 1, c="C")
## do *not* use
c(ll, d = 1:3) # which is == c(ll, as.list(c(d=1:3)))
## but rather
c(ll, d = list(1:3))# c() combining two lists

c(list(A=c(B=1)), recursive=TRUE)

c(options(), recursive=TRUE)
c(list(A=c(B=1,C=2), B=c(E=7)), recursive=TRUE)
```

---

`call`*Function Calls*

---

**Description**

Create or test for objects of mode "call".

**Usage**

```
call(name, ...)  
is.call(x)  
as.call(x)
```

**Arguments**

<code>name</code>	a character string naming the function to be called.
<code>...</code>	arguments to be part of the call.
<code>x</code>	an arbitrary R object.

**Details**

`call` returns an unevaluated function call, that is, an unevaluated expression which consists of the named function applied to the given arguments (`name` must be a quoted string which gives the name of a function to be called).

`is.call` is used to determine whether `x` is a call (i.e., of mode "call"). It is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

Objects of mode "list" can be coerced to mode "call". The first element of the list becomes the function part of the call, so should be a function or the name of one (as a symbol; a quoted string will not do).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[do.call](#) for calling a function by name and argument list; [Recall](#) for recursive calling of functions; further [is.language](#), [expression](#), [function](#).

**Examples**

```
is.call(call) #-> FALSE: Functions are NOT calls  
  
# set up a function call to round with argument 10.5  
cl <- call("round", 10.5)  
is.call(cl)# TRUE
```

```

c1
# such a call can also be evaluated.
eval(c1)# [1] 10

```

---

capabilities                      *Report Capabilities of this Build of R*

---

## Description

Report on the optional features which have been compiled into this build of R.

## Usage

```
capabilities(what = NULL)
```

## Arguments

`what`                      character vector or NULL, specifying required components. NULL implies that all are required.

## Value

A named logical vector. Current components are

<code>jpeg</code>	Is the <a href="#">jpeg</a> function operational?
<code>png</code>	Is the <a href="#">png</a> function operational?
<code>tcltk</code>	Is the <a href="#">tcltk</a> package operational?
<code>X11</code>	(Unix) Are X11 and the data editor available?
<code>GNOME</code>	(Unix) Is the GNOME GUI in use and are GTK and GNOME graphics devices available?
<code>libz</code>	Is <a href="#">gzfile</a> available? From R 1.5.0 this will always be true.
<code>http/ftp</code>	Are <a href="#">url</a> and the internal method for <a href="#">download.file</a> available?
<code>sockets</code>	Are <a href="#">make.socket</a> and related functions available?
<code>libxml</code>	Is there support for integrating <code>libxml</code> with the R event loop?
<code>cledit</code>	Is command-line editing available in the current R session? This is false in non-interactive sessions. It will be true if <code>readline</code> supported has been compiled in and <code>'--no-readline'</code> was <i>not</i> invoked.
<code>IEEE754</code>	Does this platform have IEEE 754 arithmetic? Note that this is more correctly known by the international standard IEC 60559, and will always be true from R version 2.0.0.
<code>bzip2</code>	Is <a href="#">bzfile</a> available?
<code>PCRE</code>	Is the Perl-Compatible Regular Expression library available? This is needed for the <code>perl = TRUE</code> option to <a href="#">grep</a> are related function.

**See Also**[.Platform](#)**Examples**

```
capabilities()

if(!capabilities("http/ftp"))
  warning("internal download.file() is not available")

## See also the examples for 'connections'.
```

---

  
cat*Concatenate and Print*

---

**Description**

Prints the arguments, coercing them if necessary to character mode first.

**Usage**

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL,
     append = FALSE)
```

**Arguments**

<code>...</code>	R objects which are coerced to character strings, concatenated, and printed, with the remaining arguments controlling the output.
<code>file</code>	A connection, or a character string naming the file to print to. If "" (the default), <code>cat</code> prints to the standard output connection, the console unless redirected by <a href="#">sink</a> . If it is " <code> cmd</code> ", the output is piped to the command given by ' <code>cmd</code> ', by opening a pipe connection.
<code>sep</code>	character string to insert between the objects to print.
<code>fill</code>	a logical or numeric controlling how the output is broken into successive lines. If <code>FALSE</code> (default), only newlines created explicitly by ' <code>\n</code> ' are printed. Otherwise, the output is broken into lines with print width equal to the option <code>width</code> if <code>fill</code> is <code>TRUE</code> , or the value of <code>fill</code> if this is numeric.
<code>labels</code>	character vector of labels for the lines printed. Ignored if <code>fill</code> is <code>FALSE</code> .
<code>append</code>	logical. Only used if the argument <code>file</code> is the name of file (and not a connection or " <code> cmd</code> "). If <code>TRUE</code> output will be appended to <code>file</code> ; otherwise, it will overwrite the contents of <code>file</code> .

**Details**

cat converts its arguments to character strings, concatenates them, separating them by the given sep= string, and then prints them.

No linefeeds are printed unless explicitly requested by '\n' or if generated by filling (if argument fill is TRUE or numeric.)

cat is useful for producing output in user-defined functions.

**Value**

None (invisible NULL).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[print](#), [format](#), and [paste](#) which concatenates into a string.

**Examples**

```
iter <- rpois(1, lambda=10)
## print an informative message
cat("iteration = ", iter <- iter + 1, "\n")

## 'fill' and label lines:
cat(paste(letters, 100* 1:26), fill = TRUE,
    labels = paste("{",1:10,"}:",sep=""))
```

---

cbind

*Combine R Objects by Rows or Columns*

---

**Description**

Take a sequence of vector, matrix or data frames arguments and combine by columns or rows, respectively. These are generic functions with methods for other R classes.

**Usage**

```
cbind(..., deparse.level = 1)
rbind(..., deparse.level = 1)
```

**Arguments**

... vectors or matrices. These can be given as named arguments.

deparse.level

integer controlling the construction of labels; currently, 1 is the only possible value.

## Details

The functions `cbind` and `rbind` are generic, with methods for data frames. The data frame method will be used if an argument is a data frame and the rest are vectors or matrices. There can be other methods; in particular, there is one for time series objects.

In the matrix case, all the vectors/matrices must be atomic or lists (e.g. not expressions).

Data frames can be `cbind`-ed with matrices, in which case each matrix forms a single column in the result, unlike calling `data.frame`.

The `rbind` data frame method takes the classes of the columns from the first data frame. Factors have their levels expanded as necessary (in the order of the levels of the levelsets of the factors encountered) and the result is an ordered factor if and only if all the components were ordered factors. (The last point differs from S-PLUS.)

If there are several matrix arguments, they must all have the same number of columns (or rows) and this will be the number of columns (or rows) of the result. If all the arguments are vectors, the number of columns (rows) in the result is equal to the length of the longest vector. Values in shorter arguments are recycled to achieve this length (with a `warning` if they are recycled only *fractionally*).

When the arguments consist of a mix of matrices and vectors the number of columns (rows) of the result is determined by the number of columns (rows) of the matrix arguments. Any vectors have their values recycled or subsetted to achieve this length.

For `cbind` (`rbind`), vectors of zero length (including `NULL`) are ignored unless the result would have zero rows (columns), for S compatibility. (Zero-extent matrices do not occur in S3 and are not ignored in R.)

## Value

A matrix or data frame combining the . . . arguments column-wise or row-wise.

For `cbind` (`rbind`) the column (row) names are taken from the names of the arguments, or where those are not supplied by deparsing the expressions given (if that gives a sensible name). The names will depend on whether data frames are included: see the examples.

## Note

The method dispatching is *not* done via `UseMethod()`, but by C-internal dispatching. Therefore, there is no need for, e.g., `rbind.default`.

The dispatch algorithm is described in the source file (`'.../src/main/bind.c'`) as

1. For each argument we get the list of possible class memberships from the class attribute.
2. We inspect each class in turn to see if there is an applicable method.
3. If we find an applicable method we make sure that it is identical to any method determined for prior arguments. If it is identical, we proceed, otherwise we immediately drop through to the default code.

If you want to combine other objects with data frames, it may be necessary to coerce them to data frames first. (Note that this algorithm can result in calling the data frame method if the arguments are all either data frames or vectors, and this will result in the coercion of character vectors to factors.)

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`c` to combine vectors (and lists) as vectors, `data.frame` to combine vectors and matrices as a data frame.

**Examples**

```
m <- cbind(1, 1:7) # the '1' (= shorter vector) is recycled
m
m <- cbind(m, 8:14)[, c(1, 3, 2)] # insert a column
m
cbind(1:7, diag(3))# vector is subset -> warning

cbind(0, rbind(1, 1:3))
cbind(I=0, X=rbind(a=1, b=1:3)) # use some names
xx <- data.frame(I=rep(0,2))
cbind(xx, X=rbind(a=1, b=1:3)) # named differently

cbind(0, matrix(1, nrow=0, ncol=4))#> Warning (making sense)
dim(cbind(0, matrix(1, nrow=2, ncol=0)))#-> 2 x 1
```

---

char.expand

*Expand a String with Respect to a Target Table*

---

**Description**

Seeks a unique match of its first argument among the elements of its second. If successful, it returns this element; otherwise, it performs an action specified by the third argument.

**Usage**

```
char.expand(input, target, nomatch = stop("no match"))
```

**Arguments**

input	a character string to be expanded.
target	a character vector with the values to be matched against.
nomatch	an R expression to be evaluated in case expansion was not possible.

**Details**

This function is particularly useful when abbreviations are allowed in function arguments, and need to be uniquely expanded with respect to a target table of possible values.

**See Also**

[charmatch](#) and [pmatch](#) for performing partial string matching.

**Examples**

```
locPars <- c("mean", "median", "mode")
char.expand("me", locPars, warning("Could not expand!"))
char.expand("mo", locPars)
```

---

character

*Character Vectors*

---

**Description**

Create or test for objects of type "character".

**Usage**

```
character(length = 0)
as.character(x, ...)
is.character(x)
```

**Arguments**

length	desired length.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

**Details**

`as.character` and `is.character` are generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

**Value**

`character` creates a character vector of the specified length. The elements of the vector are all equal to "".

`as.character` attempts to coerce its argument to character type; like [as.vector](#) it strips attributes including names.

`is.character` returns TRUE or FALSE depending on whether its argument is of character type or not.

**Note**

`as.character` truncates components of language objects to 500 characters (was about 70 before 1.3.1).

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[paste](#), [substr](#) and [strsplit](#) for character concatenation and splitting, [chartr](#) for character translation and casefolding (e.g., upper to lower case) and [sub](#), [grep](#) etc for string matching and substitutions. Note that `help.search(keyword = "character")` gives even more links. [deparse](#), which is normally preferable to `as.character` for language objects.

## Examples

```
form <- y ~ a + b + c
as.character(form) ## length 3
deparse(form)      ## like the input
```

---

charmatch	<i>Partial String Matching</i>
-----------	--------------------------------

---

## Description

`charmatch` seeks matches for the elements of its first argument among those of its second.

## Usage

```
charmatch(x, table, nomatch = NA)
```

## Arguments

<code>x</code>	the values to be matched.
<code>table</code>	the values to be matched against.
<code>nomatch</code>	the value returned at non-matching positions.

## Details

Exact matches are preferred to partial matches (those where the value to be matched has an exact match to the initial part of the target, but the target is longer).

If there is a single exact match or no exact match and a unique partial match then the index of the matching value is returned; if multiple exact or multiple partial matches are found then 0 is returned and if no match is found then `NA` is returned.

## Author(s)

This function is based on a C function written by Terry Therneau.

**See Also**

[pmatch](#), [match](#).

[grep](#) or [regexpr](#) for more general (regexp) matching of strings.

**Examples**

```
charmatch("", "") # returns 1
charmatch("m", c("mean", "median", "mode")) # returns 0
charmatch("med", c("mean", "median", "mode")) # returns 2
```

---

 chartr

*Character Translation and Casefolding*


---

**Description**

Translate characters in character vectors, in particular from upper to lower case or vice versa.

**Usage**

```
chartr(old, new, x)
tolower(x)
toupper(x)
casefold(x, upper = FALSE)
```

**Arguments**

x	a character vector.
old	a character string specifying the characters to be translated.
new	a character string specifying the translations.
upper	logical: translate to upper or lower case?.

**Details**

chartr translates each character in x that is specified in old to the corresponding character specified in new. Ranges are supported in the specifications, but character classes and repeated characters are not. If old contains more characters than new, an error is signaled; if it contains fewer characters, the extra characters at the end of new are ignored.

tolower and toupper convert upper-case characters in a character vector to lower-case, or vice versa. Non-alphabetic characters are left unchanged.

casefold is a wrapper for tolower and toupper provided for compatibility with S-PLUS.

**See Also**

[sub](#) and [gsub](#) for other substitutions in strings.

**Examples**

```
x <- "MiXeD cAsE 123"
chartr("iXs", "why", x)
chartr("a-cX", "D-Fw", x)
tolower(x)
toupper(x)
```

---

 chol

*The Choleski Decomposition*


---

**Description**

Compute the Choleski factorization of a real symmetric positive-definite square matrix.

**Usage**

```
chol(x, pivot = FALSE, LINPACK = pivot)
La.chol(x)
```

**Arguments**

<code>x</code>	a real symmetric, positive-definite matrix
<code>pivot</code>	Should pivoting be used?
<code>LINPACK</code>	logical. Should LINPACK be used (for compatibility with R < 1.7.0)?

**Details**

`chol(pivot = TRUE)` provides an interface to the LINPACK routine DCHDC. `La.chol` provides an interface to the LAPACK routine DPOTRF.

Note that only the upper triangular part of `x` is used, so that  $R'R = x$  when `x` is symmetric.

If `pivot = FALSE` and `x` is not non-negative definite an error occurs. If `x` is positive semi-definite (i.e., some zero eigenvalues) an error will also occur, as a numerical tolerance is used.

If `pivot = TRUE`, then the Choleski decomposition of a positive semi-definite `x` can be computed. The rank of `x` is returned as `attr(Q, "rank")`, subject to numerical errors. The pivot is returned as `attr(Q, "pivot")`. It is no longer the case that `t(Q) %*% Q` equals `x`. However, setting `pivot <- attr(Q, "pivot")` and `oo <- order(pivot)`, it is true that `t(Q[, oo]) %*% Q[, oo]` equals `x`, or, alternatively, `t(Q) %*% Q` equals `x[pivot, pivot]`. See the examples.

**Value**

The upper triangular factor of the Choleski decomposition, i.e., the matrix  $R$  such that  $R'R = x$  (see example).

If pivoting is used, then two additional attributes "pivot" and "rank" are also returned.

**Warning**

The code does not check for symmetry.

If `pivot = TRUE` and `x` is not non-negative definite then there will be no error message but a meaningless result will occur. So only use `pivot = TRUE` when `x` is non-negative definite by construction.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.

Available on-line at [http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html).

**See Also**

[chol2inv](#) for its *inverse* (without pivoting), [backsolve](#) for solving linear systems with upper triangular left sides.

[qr](#), [svd](#) for related matrix factorizations.

**Examples**

```
( m <- matrix(c(5,1,1,3),2,2) )
( cm <- chol(m) )
t(cm) %*% cm #-- = 'm'
crossprod(cm) #-- = 'm'

# now for something positive semi-definite
x <- matrix(c(1:5, (1:5)^2), 5, 2)
x <- cbind(x, x[, 1] + 3*x[, 2])
m <- crossprod(x)
qr(m)$rank # is 2, as it should be

# chol() may fail, depending on numerical rounding:
# chol() unlike qr() does not use a tolerance.
try(chol(m))

(Q <- chol(m, pivot = TRUE)) # NB wrong rank here ... see Warning section.
## we can use this by
pivot <- attr(Q, "pivot")
oo <- order(pivot)
t(Q[, oo]) %*% Q[, oo] # recover m
```

---

`chol2inv`*Inverse from Choleski Decomposition*

---

**Description**

Invert a symmetric, positive definite square matrix from its Choleski decomposition.

**Usage**

```
chol2inv(x, size = NCOL(x), LINPACK = FALSE)
La.chol2inv(x, size = ncol(x))
```

**Arguments**

<code>x</code>	a matrix. The first <code>nc</code> columns of the upper triangle contain the Choleski decomposition of the matrix to be inverted.
<code>size</code>	the number of columns of <code>x</code> containing the Choleski decomposition.
<code>LINPACK</code>	logical. Should LINPACK be used (for compatibility with R < 1.7.0)?

**Details**

`chol2inv(LINPACK=TRUE)` provides an interface to the LINPACK routine DPODI.  
`La.chol2inv` provides an interface to the LAPACK routine DPOTRI.

**Value**

The inverse of the decomposed matrix.

**References**

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson, E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM. Available on-line at [http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html).

**See Also**

[chol](#), [solve](#).

**Examples**

```
cma <- chol(cma <- cbind(1, 1:3, c(1,3,7)))
ma %*% chol2inv(cma)
```

---

class	<i>Object Classes</i>
-------	-----------------------

---

### Description

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class of the first argument to the generic function.

### Usage

```
class(x)
class(x) <- value
unclass(x)
inherits(x, what, which = FALSE)

oldClass(x)
oldClass(x) <- value
```

### Arguments

x	a R object
what, value	a character vector naming classes.
which	logical affecting return value: see Details.

### Details

Many R objects have a `class` attribute, a character vector giving the names of the classes which the object “inherits” from. If the object does not have a `class` attribute, it has an implicit class, “matrix”, “array” or the result of `mode(x)`. (Functions `oldClass` and `oldClass<-` get and set the attribute, which can also be done directly.)

When a generic function `fun` is applied to an object with class attribute `c("first", "second")`, the system searches for a function called `fun.first` and, if it finds it, applies it to the object. If no such function is found, a function called `fun.second` is tried. If no class name produces a suitable function, the function `fun.default` is used (if it exists). If there is no class attribute, the implicit class is tried, then the default method.

The function `class` prints the vector of names of classes an object inherits from. Correspondingly, `class<-` sets the classes an object inherits from.

`unclass` returns (a copy of) its argument with its class attribute removed.

`inherits` indicates whether its first argument inherits from any of the classes specified in the `what` argument. If `which` is `TRUE` then an integer vector of the same length as `what` is returned. Each element indicates the position in the `class(x)` matched by the element of `what`; zero indicates no match. If `which` is `FALSE` then `TRUE` is returned by `inherits` if any of the names in `what` match with any `class`.

### Formal classes

An additional mechanism of *formal* classes is available in packages **methods** which is attached by default. For objects which have a formal class, its name is returned by `class` as a character vector of length one.

The replacement version of the function sets the class to the value provided. For classes that have a formal definition, directly replacing the class this way is strongly deprecated. The expression `as(object, value)` is the way to coerce an object to a particular class.

### Note

Functions `oldClass` and `oldClass<-` behave in the same way as functions of those names in S-PLUS 5/6, *but* in R `UseMethod` dispatches on the class as returned by `class` rather than `oldClass`. *However*, `group generics` dispatch on the `oldClass` for efficiency.

### See Also

[UseMethod](#), [NextMethod](#), [group generic](#).

### Examples

```
x <- 10
inherits(x, "a") #FALSE
class(x) <- c("a", "b")
inherits(x, "a") #TRUE
inherits(x, "a", TRUE) # 1
inherits(x, c("a", "b", "c"), TRUE) # 1 2 0
```

---

col

*Column Indexes*

---

### Description

Returns a matrix of integers indicating their column number in the matrix.

### Usage

```
col(x, as.factor = FALSE)
```

### Arguments

<code>x</code>	a matrix.
<code>as.factor</code>	a logical value indicating whether the value should be returned as a factor rather than as numeric.

### Value

An integer matrix with the same dimensions as `x` and whose `i j`-th element is equal to `j`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[row](#) to get rows.

## Examples

```
# extract an off-diagonal of a matrix
ma <- matrix(1:12, 3, 4)
ma[row(ma) == col(ma) + 1]

# create an identity 5-by-5 matrix
x <- matrix(0, nr = 5, nc = 5)
x[row(x) == col(x)] <- 1
```

---

colSums

*Form Row and Column Sums and Means*

---

## Description

Form row and column sums and means for numeric arrays.

## Usage

```
colSums(x, na.rm = FALSE, dims = 1)
rowSums(x, na.rm = FALSE, dims = 1)
colMeans(x, na.rm = FALSE, dims = 1)
rowMeans(x, na.rm = FALSE, dims = 1)
```

## Arguments

<code>x</code>	an array of two or more dimensions, containing numeric, complex, integer or logical values, or a numeric data frame.
<code>na.rm</code>	logical. Should missing values (including NaN) be omitted from the calculations?
<code>dims</code>	Which dimensions are regarded as “rows” or “columns” to sum over. For <code>row*</code> , the sum or mean is over dimensions <code>dims+1, ...</code> ; for <code>col*</code> it is over dimensions <code>1:dims</code> .

## Details

These functions are equivalent to use of [apply](#) with `FUN = mean` or `FUN = sum` with appropriate margins, but are a lot faster. As they are written for speed, they blur over some of the subtleties of NaN and NA. If `na.rm = FALSE` and either NaN or NA appears in a sum, the result will be one of NaN or NA, but which might be platform-dependent.

**Value**

A numeric or complex array of suitable size, or a vector if the result is one-dimensional. The `dimnames` (or `names` for a vector result) are taken from the original array.

If there are no values in a range to be summed over (after removing missing values with `na.rm = TRUE`), that component of the output is set to 0 (`*Sums`) or NA (`*Means`), consistent with `sum` and `mean`.

**See Also**

`apply`, `rowsum`

**Examples**

```
## Compute row and column sums for a matrix:
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
rowSums(x); colSums(x)
dimnames(x)[[1]] <- letters[1:8]
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
x[] <- as.integer(x)
rowSums(x); colSums(x)
x[] <- x < 3
rowSums(x); colSums(x)
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
x[3, ] <- NA; x[4, 2] <- NA
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
rowSums(x, na.rm = TRUE); colSums(x, na.rm = TRUE)
rowMeans(x, na.rm = TRUE); colMeans(x, na.rm = TRUE)

## an array
dim(UCBAdmissions)
rowSums(UCBAdmissions); rowSums(UCBAdmissions, dims = 2)
colSums(UCBAdmissions); colSums(UCBAdmissions, dims = 2)

## complex case
x <- cbind(x1 = 3 + 2i, x2 = c(4:1, 2:5) - 5i)
x[3, ] <- NA; x[4, 2] <- NA
rowSums(x); colSums(x); rowMeans(x); colMeans(x)
rowSums(x, na.rm = TRUE); colSums(x, na.rm = TRUE)
rowMeans(x, na.rm = TRUE); colMeans(x, na.rm = TRUE)
```

---

commandArgs

*Extract Command Line Arguments*

---

**Description**

Provides access to a copy of the command line arguments supplied when this R session was invoked.

**Usage**

```
commandArgs()
```

**Details**

These arguments are captured before the standard R command line processing takes place. This means that they are the unmodified values. If it were useful, we could provide support an argument which indicated whether we want the unprocessed or processed values.

This is especially useful with the `--args` command-line flag to R, as all of the command line after than flag is skipped.

**Value**

A character vector containing the name of the executable and the user-supplied command line arguments. The first element is the name of the executable by which R was invoked. As far as I am aware, the exact form of this element is platform dependent. It may be the fully qualified name, or simply the last component (or basename) of the application.

**See Also**

[BATCH](#)

**Examples**

```
commandArgs()
## Spawn a copy of this application as it was invoked.
## system(paste(commandArgs(), collapse=" "))
```

---

comment

*Query or Set a 'Comment' Attribute*

---

**Description**

These functions set and query a *comment* attribute for any R objects. This is typically useful for [data.frames](#) or model fits.

Contrary to other [attributes](#), the comment is not printed (by [print](#) or [print.default](#)).

**Usage**

```
comment(x)
comment(x) <- value
```

**Arguments**

x	any R object
value	a character vector

**See Also**

[attributes](#) and [attr](#) for “normal” attributes.

**Examples**

```
x <- matrix(1:12, 3,4)
comment(x) <- c("This is my very important data from experiment #0234",
               "Jun 5, 1998")

x
comment(x)
```

---

Comparison

*Relational Operators*

---

**Description**

Binary operators which allow the comparison of values in atomic vectors.

**Usage**

```
x < y
x > y
x <= y
x >= y
x == y
x != y
```

**Arguments**

`x`, `y` atomic vectors, or other objects for which methods have been written.

**Details**

The binary comparison operators are generic functions: methods can be written for them individually or via the [Ops](#) group generic function.

Comparison of strings in character vectors is lexicographic within the strings using the collating sequence of the locale in use: see [locales](#). The collating sequence of locales such as ‘en\_US’ is normally different from ‘C’ (which should use ASCII) and can be surprising.

At least one of `x` and `y` must be an atomic vector, but if the other is a list `R` attempts to coerce it to the type of the atomic vector: this will succeed if the list is made up of elements of length one that can be coerced to the correct type.

If the two arguments are atomic vectors of different types, they are both coerced to the first of character, complex, numeric, integer and logical.

**Value**

A vector of logicals indicating the result of the element by element comparison. The elements of shorter vectors are recycled as necessary.

Objects such as arrays or time-series can be compared this way provided they are conformable.

**Note**

Don't use `==` and `!=` for tests, such as in `if` expressions, where you must get a single `TRUE` or `FALSE`. Unless you are absolutely sure that nothing unusual can happen, you should use the `identical` function instead.

For numerical values, remember `==` and `!=` do not allow for the finite representation of fractions, nor for rounding error. Using `all.equal` with `identical` is almost always preferable. See the examples.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[Syntax](#) for operator precedence.

**Examples**

```
x <- rnorm(20)
x < 1
x[x > 0]

x1 <- 0.5 - 0.3i
x2 <- 0.3 - 0.1i
x1 == x2                # FALSE on most machines
identical(all.equal(x1, x2), TRUE) # TRUE everywhere
```

---

complex

*Complex Vectors*

---

**Description**

Basic functions which support complex arithmetic in R.

**Usage**

```

complex(length.out = 0, real = numeric(), imaginary = numeric(),
        modulus = 1, argument = 0)
as.complex(x, ...)
is.complex(x)

Re(x)
Im(x)
Mod(x)
Arg(x)
Conj(x)

```

**Arguments**

<code>length.out</code>	numeric. Desired length of the output vector, inputs being recycled as needed.
<code>real</code>	numeric vector.
<code>imaginary</code>	numeric vector.
<code>modulus</code>	numeric vector.
<code>argument</code>	numeric vector.
<code>x</code>	an object, probably of mode <code>complex</code> .
<code>...</code>	further arguments passed to or from other methods.

**Details**

Complex vectors can be created with `complex`. The vector can be specified either by giving its length, its real and imaginary parts, or modulus and argument. (Giving just the length generates a vector of complex zeroes.)

`as.complex` attempts to coerce its argument to be of complex type: like `as.vector` it strips attributes including names.

Note that `is.complex` and `is.numeric` are never both TRUE.

The functions `Re`, `Im`, `Mod`, `Arg` and `Conj` have their usual interpretation as returning the real part, imaginary part, modulus, argument and complex conjugate for complex values. Modulus and argument are also called the *polar coordinates*. If  $z = x + iy$  with real  $x$  and  $y$ ,  $\text{Mod}(z) = \sqrt{x^2 + y^2}$ , and for  $\phi = \text{Arg}(z)$ ,  $x = \cos(\phi)$  and  $y = \sin(\phi)$ . They are all generic functions: methods can be defined for them individually or via the `Complex` group generic.

In addition, the elementary trigonometric, logarithmic and exponential functions are available for complex values.

`is.complex` is generic: you can write methods to handle specific classes of objects, see [Internal-Methods](#).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```

0i ^ (-3:3)

matrix(1i^ (-6:5), nr=4)#- all columns are the same
0 ^ 1i # a complex NaN

## create a complex normal vector
z <- complex(real = rnorm(100), imag = rnorm(100))
## or also (less efficiently):
z2 <- 1:2 + 1i*(8:9)

## The Arg(.) is an angle:
zz <- (rep(1:4,len=9) + 1i*(9:1))/10
zz.shift <- complex(modulus = Mod(zz), argument= Arg(zz) + pi)
plot(zz, xlim=c(-1,1), ylim=c(-1,1), col="red", asp = 1,
     main = expression(paste("Rotation by ", " ", pi == 180^o)))
abline(h=0,v=0, col="blue", lty=3)
points(zz.shift, col="orange")

```

---

conditions

*Condition Handling and Recovery*


---

**Description**

These functions provide a mechanism for handling unusual conditions, including errors and warnings.

**Usage**

```

tryCatch(expr, ..., finally)
withCallingHandlers(expr, ...)

signalCondition(cond)

simpleCondition(message, call = NULL)
simpleError      (message, call = NULL)
simpleWarning   (message, call = NULL)

## S3 method for class 'condition':
as.character(x, ...)
## S3 method for class 'error':
as.character(x, ...)
## S3 method for class 'condition':
print(x, ...)
## S3 method for class 'restart':
print(x, ...)

conditionCall(c)

```

```

## S3 method for class 'condition':
conditionCall(c)
conditionMessage(c)
## S3 method for class 'condition':
conditionMessage(c)

withRestarts(expr, ...)

computeRestarts(cond = NULL)
findRestart(name, cond = NULL)
invokeRestart(r, ...)
invokeRestartInteractively(r)

isRestart(x)
restartDescription(r)
restartFormals(r)

.signalSimpleWarning(msg, call)
.handleSimpleError(h, msg, call)

```

### Arguments

<code>c</code>	a condition object.
<code>call</code>	call expression.
<code>cond</code>	a condition object.
<code>expr</code>	expression to be evaluated.
<code>finally</code>	expression to be evaluated before returning or exiting.
<code>h</code>	function.
<code>message</code>	character string.
<code>msg</code>	character string.
<code>name</code>	character string naming a restart.
<code>r</code>	restart object.
<code>x</code>	object.
<code>...</code>	additional arguments; see details below.

### Details

The condition system provides a mechanism for signaling and handling unusual conditions, including errors and warnings. Conditions are represented as objects that contain information about the condition that occurred, such as a message and the call in which the condition occurred. Currently conditions are S3-style objects, though this may eventually change.

Conditions are objects inheriting from the abstract class `condition`. Errors and warnings are objects inheriting from the abstract subclasses `error` and `warning`. The class `simpleError` is the class used by `stop` and all internal error signals. Similarly, `simpleWarning` is used by `warning`. The constructors by the same names take a string describing the condition as argument

and an optional call. The functions `conditionMessage` and `conditionCall` are generic functions that return the message and call of a condition.

Conditions are signaled by `signalCondition`. In addition, the `stop` and `warning` functions have been modified to also accept condition arguments.

The function `tryCatch` evaluates its expression argument in a context where the handlers provided in the `...` argument are available. The `finally` expression is then evaluated in the context in which `tryCatch` was called; that is, the handlers supplied to the current `tryCatch` call are not active when the `finally` expression is evaluated.

Handlers provided in the `...` argument to `tryCatch` are established for the duration of the evaluation of `expr`. If no condition is signaled when evaluating `expr` then `tryCatch` returns the value of the expression.

If a condition is signaled while evaluating `expr` then established handlers are checked, starting with the most recently established ones, for one matching the class of the condition. When several handlers are supplied in a single `tryCatch` then the first one is considered more recent than the second. If a handler is found then control is transferred to the `tryCatch` call that established the handler, the handler found and all more recent handlers are disestablished, the handler is called with the condition as its argument, and the result returned by the handler is returned as the value of the `tryCatch` call.

Calling handlers are established by `withCallingHandlers`. If a condition is signaled and the applicable handler is a calling handler, then the handler is called by `signalCondition` in the context where the condition was signaled but with the available handlers restricted to those below the handler called in the handler stack. If the handler returns, then the next handler is tried; once the last handler has been tried, `signalCondition` returns `NULL`.

User interrupts signal a condition of class `interrupt` that inherits directly from class `condition` before executing the default interrupt action.

Restarts are used for establishing recovery protocols. They can be established using `withRestarts`. One pre-established restart is an `abort` restart that represents a jump to top level.

`findRestart` and `computeRestarts` find the available restarts. `findRestart` returns the most recently established restart of the specified name. `computeRestarts` returns a list of all restarts. Both can be given a condition argument and will then ignore restarts that do not apply to the condition.

`invokeRestart` transfers control to the point where the specified restart was established and calls the restart's handler with the arguments, if any, given as additional arguments to `invokeRestart`. The restart argument to `invokeRestart` can be a character string, in which case `findRestart` is used to find the restart.

New restarts for `withRestarts` can be specified in several ways. The simplest is in `name=function` form where the function is the handler to call when the restart is invoked. Another simple variant is `name=string` where the string is stored in the `description` field of the restart object returned by `findRestart`; in this case the handler ignores its arguments and returns `NULL`. The most flexible form of a restart specification is as a list that can include several fields, including `handler`, `description`, and `test`. The `test` field should contain a function of one argument, a condition, that returns `TRUE` if the restart applies to the condition and `FALSE` if it does not; the default function returns `TRUE` for all conditions.

One additional field that can be specified for a restart is `interactive`. This should be a function of no arguments that returns a list of arguments to pass to the restart handler. The list could be obtained by interacting with the user if necessary. The function `invokeRestartInteractively` calls this function to obtain the arguments to use when invoking the restart. The default `interactive` method queries the user for values for the formal arguments of the handler function.

`.signalSimpleWarning` and `.handleSimpleError` are used internally and should not be called directly.

## References

The `tryCatch` mechanism is similar to Java error handling. Calling handlers are based on Common Lisp and Dylan. Restarts are based on the Common Lisp restart mechanism.

## See Also

`stop` and `warning` signal conditions, and `try` is essentially a simplified version of `tryCatch`.

## Examples

```
tryCatch(1, finally=print("Hello"))
e <- simpleError("test error")
## Not run: stop(e)
## Not run: tryCatch(stop(e), finally=print("Hello"))
## Not run: tryCatch(stop("fred"), finally=print("Hello"))
tryCatch(stop(e), error = function(e) e, finally=print("Hello"))
tryCatch(stop("fred"), error = function(e) e, finally=print("Hello"))
withCallingHandlers({ warning("A"); 1+2 }, warning = function(w) {})
{ try(invokeRestart("tryRestart")); 1}
## Not run: { withRestarts(stop("A"), abort = function() {}); 1}
withRestarts(invokeRestart("foo", 1, 2), foo = function(x, y) {x + y})
```

---

conflicts

*Search for Masked Objects on the Search Path*

---

## Description

`conflicts` reports on objects that exist with the same name in two or more places on the [search path](#), usually because an object in the user's workspace or a package is masking a system object of the same name. This helps discover unintentional masking.

## Usage

```
conflicts(where = search(), detail = FALSE)
```

**Arguments**

`where`            A subset of the search path, by default the whole search path.

`detail`            If TRUE, give the masked or masking functions for all members of the search path.

**Value**

If `detail=FALSE`, a character vector of masked objects. If `detail=TRUE`, a list of character vectors giving the masked or masking objects in that member of the search path. Empty vectors are omitted.

**Examples**

```
lm <- 1:3
conflicts(, TRUE)
## gives something like
# $.GlobalEnv
# [1] "lm"
#
# $package:base
# [1] "lm"

## Remove things from your "workspace" that mask others:
remove(list = conflicts(detail=TRUE)$GlobalEnv)
```

---

connections

*Functions to Manipulate Connections*


---

**Description**

Functions to create, open and close connections.

**Usage**

```
file(description = "", open = "", blocking = TRUE,
      encoding = getOption("encoding"))
pipe(description, open = "", encoding = getOption("encoding"))
fifo(description = "", open = "", blocking = FALSE,
      encoding = getOption("encoding"))
gzfile(description, open = "", encoding = getOption("encoding"),
      compression = 6)
unz(description, filename, open = "", encoding = getOption("encoding"))
bzfile(description, open = "", encoding = getOption("encoding"))
url(description, open = "", blocking = TRUE,
      encoding = getOption("encoding"))
socketConnection(host = "localhost", port, server = FALSE,
      blocking = FALSE, open = "a+",
```

```

        encoding = getOption("encoding"))

open(con, ...)
## S3 method for class 'connection':
open(con, open = "r", blocking = TRUE, ...)

close(con, ...)
## S3 method for class 'connection':
close(con, type = "rw", ...)

flush(con)

isOpen(con, rw = "")
isIncomplete(con)

```

### Arguments

<code>description</code>	character. A description of the connection. For file and pipe this is a path to the file to be opened. For url it is a complete URL, including schemes ( <code>http://</code> , <code>ftp://</code> or <code>file://</code> ). file also accepts complete URLs.
<code>filename</code>	a filename within a zip file.
<code>con</code>	a connection.
<code>host</code>	character. Host name for port.
<code>port</code>	integer. The TCP port number.
<code>server</code>	logical. Should the socket be a client or a server?
<code>open</code>	character. A description of how to open the connection (if at all). See Details for possible values.
<code>blocking</code>	logical. See ‘Blocking’ section below.
<code>encoding</code>	An integer vector of length 256.
<code>compression</code>	integer in 0–9. The amount of compression to be applied when writing, from none to maximal. The default is a good space/time compromise.
<code>type</code>	character. Currently ignored.
<code>rw</code>	character. Empty or "read" or "write", partial matches allowed.
<code>...</code>	arguments passed to or from other methods.

### Details

The first eight functions create connections. By default the connection is not opened (except for `socketConnection`), but may be opened by setting a non-empty value of argument `open`.

`gzfile` applies to files compressed by ‘gzip’, and `bzfile` to those compressed by ‘bzip2’: such connections can only be binary.

`unz` reads (only) single files within zip files, in binary mode. The description is the full path, with ‘.zip’ extension if required.

All platforms support `file`, `gzfile`, `bzfile`, `unz` and `url("file://")` connections. The other types may be partially implemented or not implemented at all. (They do work on most Unix platforms, and all but `fifo` on Windows.)

Proxies can be specified for `url` connections: see [download.file](#).

`open`, `close` and `seek` are generic functions: the following applies to the methods relevant to connections.

`open` opens a connection. In general functions using connections will open them if they are not open, but then close them again, so to leave a connection open call `open` explicitly.

Possible values for the mode `open` to open a connection are

- "r" or "rt"** Open for reading in text mode.
- "w" or "wt"** Open for writing in text mode.
- "a" or "at"** Open for appending in text mode.
- "rb"** Open for reading in binary mode.
- "wb"** Open for writing in binary mode.
- "ab"** Open for appending in binary mode.
- "r+", "r+b"** Open for reading and writing.
- "w+", "w+b"** Open for reading and writing, truncating file initially.
- "a+", "a+b"** Open for reading and appending.

Not all modes are applicable to all connections: for example URLs can only be opened for reading. Only `file` and `socket` connections can be opened for reading and writing/appending. For many connections there is little or no difference between text and binary modes, but there is for file-like connections on Windows, and `pushBack` is text-oriented and is only allowed on connections open for reading in text mode.

`close` closes and destroys a connection.

`flush` flushes the output stream of a connection open for write/append (where implemented).

If for a `file` connection the description is "", the file is immediately opened in "w+" mode and unlinked from the file system. This provides a temporary file to write to and then read from.

The encoding vector is used to map the input from a file or pipe to the platform's native character set. Supplied examples are `native.enc` as well as `MacRoman`, `WinAnsi` and `ISOLatin1`, whose actual encoding is platform-dependent. Missing characters are mapped to a space in these encodings.

## Value

`file`, `pipe`, `fifo`, `url`, `gzfile` and `socketConnection` return a connection object which inherits from class "connection" and has a first more specific class.

`isOpen` returns a logical value, whether the connection is currently open.

`isIncomplete` returns a logical value, whether last read attempt was blocked, or for an output text connection whether there is unflushed output.

## Blocking

The default condition for all but `fifo` and `socket` connections is to be in blocking mode. In that mode, functions do not return to the R evaluator until they are complete. In non-blocking mode, operations return as soon as possible, so on input they will return with whatever input is available (possibly none) and for output they will return whether or not the write succeeded.

The function `readLines` behaves differently in respect of incomplete last lines in the two modes: see its help page.

Even when a connection is in blocking mode, attempts are made to ensure that it does not block the event loop and hence the operation of GUI parts of R. These do not always succeed, and the whole process will be blocked during a DNS lookup on Unix, for example.

Most blocking operations on URLs and sockets are subject to the timeout set by `options("timeout")`. Note that this is a timeout for no response at all, not for the whole operation.

## Fifos

Fifos default to non-blocking. That follows Svr4 and it probably most natural, but it does have some implications. In particular, opening a non-blocking `fifo` connection for writing (only) will fail unless some other process is reading on the `fifo`.

Opening a `fifo` for both reading and writing (in any mode: one can only append to `fifos`) connects both sides of the `fifo` to the R process, and provides an similar facility to `file()`.

## Note

R's connections are modelled on those in S version 4 (see Chambers, 1998). However R goes well beyond the Svr4 model, for example in output text connections and URL, `gzfile`, `bzfile` and `socket` connections.

The default mode in R is "r" except for `socket` connections. This differs from Svr4, where it is the equivalent of "r+", known as "\*".

On platforms where `vsnprintf` does not return the needed length of output (e.g., Windows) there is a 100,000 character output limit on the length of line for `fifo`, `gzfile` and `bzfile` connections: longer lines will be truncated with a warning.

## References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

## See Also

`textConnection`, `seek`, `readLines`, `readBin`, `writeLines`, `writeBin`, `showConnections`, `pushBack`.

`capabilities` to see if `gzfile`, `url`, `fifo` and `socketConnection` are supported by this build of R.

**Examples**

```

zz <- file("ex.data", "w") # open an output file connection
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
cat("One more line\n", file = zz)
close(zz)
readLines("ex.data")
unlink("ex.data")

zz <- gzfile("ex.gz", "w") # compressed file
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(gzfile("ex.gz"))
unlink("ex.gz")

zz <- bzfile("ex.bz2", "w") # bzip2-ed file
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
print(readLines(bzfile("ex.bz2")))
unlink("ex.bz2")

## An example of a file open for reading and writing
Tfile <- file("test1", "w+")
c(isOpen(Tfile, "r"), isOpen(Tfile, "w")) # both TRUE
cat("abc\ndef\n", file=Tfile)
readLines(Tfile)
seek(Tfile, 0, rw="r") # reset to beginning
readLines(Tfile)
cat("ghi\n", file=Tfile)
readLines(Tfile)
close(Tfile)
unlink("test1")

## We can do the same thing with an anonymous file.
Tfile <- file()
cat("abc\ndef\n", file=Tfile)
readLines(Tfile)
close(Tfile)

if(capabilities("fifo")){
  zz <- fifo("foo", "w+")
  writeLines("abc", zz)
  print(readLines(zz))
  close(zz)
  unlink("foo")
}

## Not run: ## Unix examples of use of pipes

# read listing of current directory
readLines(pipe("ls -l"))

# remove trailing commas. Suppose

```

```

% cat data2
450, 390, 467, 654, 30, 542, 334, 432, 421,
357, 497, 493, 550, 549, 467, 575, 578, 342,
446, 547, 534, 495, 979, 479
# Then read this by
scan(pipe("sed -e s/,,$// data2"), sep=",")

# convert decimal point to comma in output
# both R strings and (probably) the shell need \ doubled
zz <- pipe(paste("sed s/\\\\. / >", "outfile"), "w")
cat(format(round(rnorm(100), 4)), sep = "\n", file = zz)
close(zz)
file.show("outfile", delete.file=TRUE)## End(Not run)

## Not run: ## example for Unix machine running a finger daemon

con <- socketConnection(port = 79, blocking = TRUE)
writeLines(paste(system("whoami", intern=TRUE), "\r", sep=""), con)
gsub(" *$", "", readLines(con))
close(con)## End(Not run)

## Not run: ## two R processes communicating via non-blocking sockets
# R process 1
con1 <- socketConnection(port = 6011, server=TRUE)
writeLines(LETTERS, con1)
close(con1)

# R process 2
con2 <- socketConnection(Sys.info()["nodename"], port = 6011)
# as non-blocking, may need to loop for input
readLines(con2)
while(isIncomplete(con2)) {Sys.sleep(1); readLines(con2)}
close(con2)## End(Not run)

```

---

Constants

*Built-in Constants*

---

## Description

Constants built into R.

## Usage

```

LETTERS
letters
month.abb
month.name
pi

```

## Details

R has a limited number of built-in constants (there is also a rather larger library of data sets which can be loaded with the function `data`).

The following constants are available:

- `LETTERS`: the 26 upper-case letters of the Roman alphabet;
- `letters`: the 26 lower-case letters of the Roman alphabet;
- `month.abb`: the three-letter abbreviations for the English month names;
- `month.name`: the English names for the months of the year;
- `pi`: the ratio of the circumference of a circle to its diameter.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

`data`.

## Examples

```
# John Machin (1705) computed 100 decimals of pi :  
pi - 4*(4*atan(1/5) - atan(1/239))
```

---

contributors

*R Project Contributors*

---

## Description

The R Who-is-who, describing who made significant contributions to the development of R.

## Usage

```
contributors()
```

**Description**

These are the basic control-flow constructs of the R language. They function in much the same way as control statements in any Algol-like language.

**Usage**

```
if(cond) expr
if(cond) cons.expr else alt.expr

for(var in seq) expr
while(cond) expr
repeat expr
break
next
```

**Arguments**

<code>cond</code>	A length-one logical vector that is not NA. Conditions of length greater than one are accepted with a warning, but only the first element is used.
<code>var</code>	A syntactical name for a variable.
<code>seq</code>	An expression evaluating to a vector (including a list).
<code>expr, cons.expr, alt.expr</code>	An <i>expression</i> in a formal sense. This is either a simple expression or a so called <i>compound expression</i> , usually of the form <code>{ expr1 ; expr2 }</code> .

**Details**

`break` breaks out of a `for`, `while` or `repeat` loop, that is goes to the first statement after the loop. `next` transfer control to the first statement of the loop (after advancing `var` along `seq` in a `for` loop). They both apply only to the innermost of nested loops.

Note that it is a common mistake to forget to put braces (`{ . . . }`) around your statements, e.g., after `if(. . .)` or `for(. . . .)`. In particular, you should not have a newline between `}` and `else` to avoid a syntax error in entering a `if . . . else` construct at the keyboard or via `source`. For that reason, one (somewhat extreme) attitude of defensive programming uses braces always, e.g., for `if` clauses.

The index `seq` in a `for` loop is evaluated at the start of the loop; changing it subsequently does not affect the loop. The variable `var` has the same type as `seq`. If `seq` is a factor (which is not strictly allowed) then its internal codes are used: the effect is that of `as.integer` not `as.vector`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[Syntax](#) for the basic R syntax and operators, [Paren](#) for parentheses and braces; further, [ifelse](#), [switch](#).

**Examples**

```
for(i in 1:5) print(1:i)
for(n in c(2,5,10,20,50)) {
  x <- rnorm(n)
  cat(n, ":", sum(x^2), "\n")
}
```

---

copyright

*Copyrights of Files Used to Build R*

---

**Description**

R is released under the ‘GNU Public License’: see [license](#) for details. The license describes your right to use R. Copyright is concerned with ownership of intellectual rights, and some of the software used has conditions that the copyright must be explicitly stated: see the Details section. We are grateful to these people and other contributors (see [contributors](#)) for the ability to use their work.

**Details**

The file ‘\$R\_HOME/COPYRIGHTS’ lists the copyrights in full detail.

---

count.fields

*Count the Number of Fields per Line*

---

**Description**

count.fields counts the number of fields, as separated by sep, in each of the lines of file read.

**Usage**

```
count.fields(file, sep = "", quote = "\"'", skip = 0,
             blank.lines.skip = TRUE, comment.char = "#")
```

**Arguments**

<code>file</code>	a character string naming an ASCII data file, or a <a href="#">connection</a> , which will be opened if necessary, and if so closed at the end of the function call.
<code>sep</code>	the field separator character. Values on each line of the file are separated by this character. By default, arbitrary amounts of whitespace can separate fields.
<code>quote</code>	the set of quoting characters
<code>skip</code>	the number of lines of the data file to skip before beginning to read data.
<code>blank.lines.skip</code>	logical: if TRUE blank lines in the input are ignored.
<code>comment.char</code>	character: a character vector of length one containing a single character or an empty string.

**Details**

This used to be used by [read.table](#) and can still be useful in discovering problems in reading a file by that function.

For the handling of comments, see [scan](#).

**Value**

A vector with the numbers of fields found.

**See Also**

[read.table](#)

**Examples**

```
cat("NAME", "1:John", "2:Paul", file = "foo", sep = "\n")
count.fields("foo", sep = ":")
unlink("foo")
```

---

crossprod

*Matrix Crossproduct*

---

**Description**

Given matrices `x` and `y` as arguments, `crossprod` returns their matrix cross-product. This is formally equivalent to, but faster than, the call `t(x) %*% y`.

**Usage**

```
crossprod(x, y = NULL)
```

**Arguments**

`x`, `y` matrices: `y = NULL` is taken to be the same matrix as `x`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

`%*%` and outer product `%o%`.

## Examples

```
(z <- crossprod(1:4))    # = sum(1 + 2^2 + 3^2 + 4^2)
drop(z)                 # scalar
```

---

cumsum

*Cumulative Sums, Products, and Extremes*

---

## Description

Returns a vector whose elements are the cumulative sums, products, minima or maxima of the elements of the argument.

## Usage

```
cumsum(x)
cumprod(x)
cummax(x)
cummin(x)
```

## Arguments

`x` a numeric object.

## Details

An NA value in `x` causes the corresponding and following elements of the return value to be NA.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (cumsum only.)

## Examples

```
cumsum(1:10)
cumprod(1:10)
cummin(c(3:1, 2:0, 4:2))
cummax(c(3:1, 2:0, 4:2))
```

cut

*Convert Numeric to Factor***Description**

cut divides the range of `x` into intervals and codes the values in `x` according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.

**Usage**

```
cut(x, ...)

## Default S3 method:
cut(x, breaks, labels = NULL,
     include.lowest = FALSE, right = TRUE, dig.lab = 3, ...)
```

**Arguments**

<code>x</code>	a numeric vector which is to be converted to a factor by cutting.
<code>breaks</code>	either a vector of cut points or number giving the number of intervals which <code>x</code> is to be cut into.
<code>labels</code>	labels for the levels of the resulting category. By default, labels are constructed using "(a,b]" interval notation. If <code>labels = FALSE</code> , simple integer codes are returned instead of a factor.
<code>include.lowest</code>	logical, indicating if an 'x[i]' equal to the lowest (or highest, for <code>right = FALSE</code> ) 'breaks' value should be included.
<code>right</code>	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
<code>dig.lab</code>	integer which is used when labels are not given. It determines the number of digits used in formatting the break numbers.
<code>...</code>	further arguments passed to or from other methods.

**Details**

If a `labels` parameter is specified, its values are used to name the factor levels. If none is specified, the factor level labels are constructed as "(b1, b2]", "(b2, b3]" etc. for `right=TRUE` and as "[b1, b2)", ...if `right=FALSE`. In this case, `dig.lab` indicates how many digits should be used in formatting the numbers `b1, b2, ...`

**Value**

A **factor** is returned, unless `labels = FALSE` which results in the mere integer level codes.

**Note**

Instead of `table(cut(x, br))`, `hist(x, br, plot = FALSE)` is more efficient and less memory hungry. Instead of `cut(*, labels = FALSE)`, `findInterval()` is more efficient.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`split` for splitting a variable according to a group factor; `factor`, `tabulate`, `table`, `findInterval()`.

**Examples**

```
Z <- rnorm(10000)
table(cut(Z, br = -6:6))
sum(table(cut(Z, br = -6:6, labels=FALSE)))
sum( hist (Z, br = -6:6, plot=FALSE)$counts)

cut(rep(1,5),4)#-- dummy
tx0 <- c(9, 4, 6, 5, 3, 10, 5, 3, 5)
x <- rep(0:8, tx0)
stopifnot(table(x) == tx0)

table( cut(x, b = 8))
table( cut(x, br = 3*(-2:5)))
table( cut(x, br = 3*(-2:5), right = FALSE))

##--- some values OUTSIDE the breaks :
table(cx <- cut(x, br = 2*(0:4)))
table(cxl <- cut(x, br = 2*(0:4), right = FALSE))
which(is.na(cx)); x[is.na(cx)] #-- the first 9 values 0
which(is.na(cxl)); x[is.na(cxl)] #-- the last 5 values 8

## Label construction:
y <- rnorm(100)
table(cut(y, breaks = pi/3*(-3:3)))
table(cut(y, breaks = pi/3*(-3:3), dig.lab=4))

table(cut(y, breaks = 1*(-3:3), dig.lab=4))
# extra digits don't "harm" here
table(cut(y, breaks = 1*(-3:3), right = FALSE))
#- the same, since no exact INT!
```

cut.POSIXt

*Convert a Date or Date-Time Object to a Factor***Description**

Method for `cut` applied to date-time objects.

**Usage**

```
## S3 method for class 'POSIXt':
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)

## S3 method for class 'Date':
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)
```

**Arguments**

<code>x</code>	an object inheriting from class "POSIXt" or "Date".
<code>breaks</code>	a vector of cut points <i>or</i> number giving the number of intervals which <code>x</code> is to be cut into <i>or</i> an interval specification, one of "sec", "min", "hour", "day", "DSTday", "week", "month" or "year", optionally preceded by an integer and a space, or followed by "s". For "Date" objects only "day", "week", "month" and "year" are allowed.
<code>labels</code>	labels for the levels of the resulting category. By default, labels are constructed from the left-hand end of the intervals (which are include for the default value of <code>right</code> ). If <code>labels = FALSE</code> , simple integer codes are returned instead of a factor.
<code>start.on.monday</code>	logical. If <code>breaks = "weeks"</code> , should the week start on Mondays or Sundays?
<code>right, ...</code>	arguments to be passed to or from other methods.

**Value**

A factor is returned, unless `labels = FALSE` which returns the integer level codes.

**See Also**

[seq.POSIXt](#), [seq.Date](#), [cut](#)

**Examples**

```
## random dates in a 10-week period
cut(ISOdate(2001, 1, 1) + 70*86400*runif(100), "weeks")
cut(as.Date("2001/1/1") + 70*runif(100), "weeks")
```

---

data.class                      *Object Classes*

---

### Description

Determine the class of an arbitrary R object.

### Usage

```
data.class(x)
```

### Arguments

`x`                      an R object.

### Value

character string giving the “class” of `x`.

The “class” is the (first element) of the `class` attribute if this is non-NULL, or inferred from the object’s `dim` attribute if this is non-NULL, or `mode(x)`.

Simply speaking, `data.class(x)` returns what is typically useful for method dispatching. (Or, what the basic creator functions already and maybe eventually all will attach as a class attribute.)

### Note

For compatibility reasons, there is one exception to the rule above: When `x` is `integer`, the result of `data.class(x)` is `"numeric"` even when `x` is classed.

### See Also

`class`

### Examples

```
x <- LETTERS
data.class(factor(x))                      # has a class attribute
data.class(matrix(x, nc = 13))            # has a dim attribute
data.class(list(x))                        # the same as mode(x)
data.class(x)                              # the same as mode(x)

stopifnot(data.class(1:2) == "numeric")# compatibility "rule"
```

---

data.frame	<i>Data Frames</i>
------------	--------------------

---

### Description

This function creates data frames, tightly coupled collections of variables which share many of the properties of matrices and of lists, used as the fundamental data structure by most of R's modeling software.

### Usage

```
data.frame(..., row.names = NULL, check.rows = FALSE,
           check.names = TRUE)
```

### Arguments

...	these arguments are of either the form <code>value</code> or <code>tag=value</code> . Component names are created based on the tag (if present) or the deparsed argument itself.
<code>row.names</code>	NULL or an integer or character string specifying a column to be used as row names, or a character vector giving the row names for the data frame.
<code>check.rows</code>	if TRUE then the rows are checked for consistency of length and names.
<code>check.names</code>	logical. If TRUE then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names and are not duplicated. If necessary they are adjusted (by <a href="#">make.names</a> ) so that they are.

### Details

A data frame is a list of variables of the same length with unique row names, given class "data.frame".

`data.frame` converts each of its arguments to a data frame by calling [as.data.frame](#) (`optional=TRUE`). As that is a generic function, methods can be written to change the behaviour of arguments according to their classes: R comes with many such methods. Character variables passed to `data.frame` are converted to factor columns unless protected by [I](#). If a list or data frame or matrix is passed to `data.frame` it is as if each component or column had been passed as a separate argument.

Objects passed to `data.frame` should have the same number of rows, but atomic vectors, factors and character vectors protected by [I](#) will be recycled a whole number of times if necessary.

If row names are not supplied in the call to `data.frame`, the row names are taken from the first component that has suitable names, for example a named vector or a matrix with `rownames` or a data frame. (If that component is subsequently recycled, the names are discarded with a warning.) If `row.names` was supplied as NULL or no suitable component was found the row names are the integer sequence starting at one.

If row names are supplied of length one and the data frame has a single row, the `row.names` is taken to specify the row names and not a column (by name or number).

Names are removed from vector inputs not protected by [I](#).

**Value**

A data frame, a matrix-like structure whose columns may be of differing types (numeric, logical, factor and character and so on).

**Note**

In versions of R prior to 1.4.0 logical columns were converted to factors (as in S3 but not S4).

**References**

Chambers, J. M. (1992) *Data for models*. Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[I](#), [plot.data.frame](#), [print.data.frame](#), [row.names](#), [\[.data.frame](#) for subsetting methods, [Math.data.frame](#) etc, about *Group* methods for `data.frames`; [read.table](#), [make.names](#).

**Examples**

```
L3 <- LETTERS[1:3]
(d <- data.frame(cbind(x=1, y=1:10), fac=sample(L3, 10, repl=TRUE)))

## The same with automatic column names:
data.frame(cbind( 1, 1:10),      sample(L3, 10, repl=TRUE))

is.data.frame(d)

## do not convert to factor, using I() :
(dd <- cbind(d, char = I(letters[1:10])))
rbind(class=sapply(dd, class), mode=sapply(dd, mode))

stopifnot(1:10 == row.names(d))# {coercion}

(d0 <- d[, FALSE]) # NULL data frame with 10 rows
(d.0 <- d[FALSE, ]) # <0 rows> data frame (3 cols)
(d00 <- d0[FALSE,]) # NULL data frame with 0 rows
```

---

data.matrix

*Data Frame to Numeric Matrix*


---

**Description**

Return the matrix obtained by converting all the variables in a data frame to numeric mode and then binding them together as the columns of a matrix. Factors and ordered factors are replaced by their internal codes.

**Usage**

```
data.matrix(frame)
```

**Arguments**

`frame` a data frame whose components are logical vectors, factors or numeric vectors.

**References**

Chambers, J. M. (1992) *Data for models*. Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[as.matrix](#), [data.frame](#), [matrix](#).

---

<code>date</code>	<i>System Date and Time</i>
-------------------	-----------------------------

---

**Description**

Returns a character string of the current system date and time.

**Usage**

```
date()
```

**Value**

The string has the form "Fri Aug 20 11:11:00 1999", i.e., length 24, since it relies on POSIX's `ctime` ensuring the above fixed format. Timezone and Daylight Saving Time are taken account of, but *not* indicated in the result.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[Sys.time](#)

**Examples**

```
(d <- date())
nchar(d) == 24
```

---

Dates

*Date Class*

---

### Description

Description of the class "Date" representing calendar dates.

### Details

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates. They are always printed following the rules of the current Gregorian calendar, even though that calendar was not in use long ago (it was adopted in 1752 in Great Britain and its colonies).

It is intended that the date should be an integer, but this is not enforced in the internal representation. Fractional days will be ignored when printing. It is possible to produce fractional days via the `mean` method or by adding or subtracting an object of class "`difftime`".

### See Also

`Sys.Date` for the current date.

`format.Date` for conversion to and from character strings.

`plot.Date` and `hist.Date` for plotting.

`weekdays` for convenience extraction functions.

`seq.Date`, `cut.Date`, `round.Date` for utility operations.

`DateTimeClasses` for date-time classes.

### Examples

```
(today <- Sys.Date())
format(today, "%d %b %Y") # with month as a word
(tenweeks <- seq(today, len=10, by="1 week")) # next ten weeks
weekdays(today)
months(tenweeks)
as.Date(.leap.seconds)
```

---

`DateTimeClasses`

*Date-Time Classes*

---

### Description

Description of the classes "POSIXlt" and "POSIXct" representing calendar dates and times (to the nearest second).

**Usage**

```
## S3 method for class 'POSIXct':
print(x, ...)

## S3 method for class 'POSIXct':
summary(object, digits = 15, ...)

time + number
time - number
time1 lop time2
```

**Arguments**

<code>x</code> , <code>object</code>	An object to be printed or summarized from one of the date-time classes.
<code>digits</code>	Number of significant digits for the computations: should be high enough to represent the least important time unit exactly.
<code>...</code>	Further arguments to be passed from or to other methods.
<code>time</code> , <code>time1</code> , <code>time2</code>	date-time objects.
<code>number</code>	a numeric object.
<code>lop</code>	One of <code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> or <code>&gt;=</code> .

**Details**

There are two basic classes of date/times. Class "POSIXct" represents the (signed) number of seconds since the beginning of 1970 as a numeric vector. Class "POSIXlt" is a named list of vectors representing

**sec** 0–61: seconds

**min** 0–59: minutes

**hour** 0–23: hours

**mday** 1–31: day of the month

**mon** 0–11: months after the first of the year.

**year** Years since 1900.

**wday** 0–6 day of the week, starting on Sunday.

**yday** 0–365: day of the year.

**isdst** Daylight savings time flag. Positive if in force, zero if not, negative if unknown.

The classes correspond to the ANSI C constructs of “calendar time” (the `time_t` data type) and “local time” (or broken-down time, the `struct tm` data type), from which they also inherit their names.

"POSIXct" is more convenient for including in data frames, and "POSIXlt" is closer to human-readable forms. A virtual class "POSIXt" inherits from both of the classes: it is used to allow operations such as subtraction to mix the two classes.

Logical comparisons and limited arithmetic are available for both classes. One can add or subtract a number of seconds or a `difftime` object from a date-time object, but not add two date-time objects. Subtraction of two date-time objects is equivalent to using `difftime`. Be aware that "POSIXlt" objects will be interpreted as being in the current timezone for these operations, unless a timezone has been specified.

"POSIXlt" objects will often have an attribute "tzzone", a character vector of length 3 giving the timezone name from the TZ environment variable and the names of the base timezone and the alternate (daylight-saving) timezone. Sometimes this may just be of length one, giving the timezone name.

"POSIXct" objects may also have an attribute "tzzone", a character vector of length one. If set, it will determine how the object is converted to class "POSIXlt" and in particular how it is printed. This is usually desirable, but if you want to specify an object in a particular timezone but to be printed in the current timezone you may want to remove the "tzzone" attribute (e.g. by `c(x)`).

Unfortunately, the conversion is complicated by the operation of time zones and leap seconds (22 days have been 86401 seconds long so far: the times of the extra seconds are in the object `.leap.seconds`). The details of this are entrusted to the OS services where possible. This will usually cover the period 1970–2037, and on Unix machines back to 1902 (when time zones were in their infancy). Outside those ranges we use our own C code. This uses the offset from GMT in use in the timezone in 2000, and uses the alternate (daylight-saving) timezone only if `isdst` is positive.

It seems that some systems use leap seconds but most do not. This is detected and corrected for at build time, so all "POSIXct" times used by R do not include leap seconds. (Conceivably this could be wrong if the system has changed since build time, just possibly by changing locales.)

Using `c` on "POSIXlt" objects converts them to the current time zone.

### Warning

Some Unix-like systems (especially Linux ones) do not have "TZ" set, yet have internal code that expects it (as does POSIX). We have tried to work around this, but if you get unexpected results try setting "TZ".

### See Also

[Dates](#) for dates without times.

[as.POSIXct](#) and [as.POSIXlt](#) for conversion between the classes.

[strptime](#) for conversion to and from character representations.

[Sys.time](#) for clock time as a "POSIXct" object.

[difftime](#) for time intervals.

[cut.POSIXt](#), [seq.POSIXt](#), [round.POSIXt](#) and [trunc.POSIXt](#) for methods for these classes.

[weekdays.POSIXt](#) for convenience extraction functions.

### Examples

```
(z <- Sys.time())           # the current date, as class "POSIXct"
```

```

Sys.time() - 3600           # an hour ago

as.POSIXlt(Sys.time(), "GMT") # the current time in GMT
format(.leap.seconds)       # all 22 leapseconds in your timezone
print(.leap.seconds, tz="PST8PDT") # and in Seattle's

```

---

dcf

---

*Read and Write Data in DCF Format*


---

## Description

Reads or writes an R object from/to a file in Debian Control File format.

## Usage

```

read.dcf(file, fields=NULL)
write.dcf(x, file = "", append = FALSE,
          indent = 0.1 * getOption("width"),
          width = 0.9 * getOption("width"))

```

## Arguments

file	either a character string naming a file or a connection. "" indicates output to the console.
fields	Fields to read from the DCF file. Default is to read all fields.
x	the object to be written, typically a data frame. If not, it is attempted to coerce x to a data frame.
append	logical. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.
indent	a positive integer specifying the indentation for continuation lines in output entries.
width	a positive integer giving the target column for wrapping lines in the output.

## Details

DCF is a simple format for storing databases in plain text files that can easily be directly read and written by humans. DCF is used in various places to store R system information, like descriptions and contents of packages.

The DCF rules as implemented in R are:

1. A database consists of one or more records, each with one or more named fields. Not every record must contain each field, a field may appear only once in a record.
2. Regular lines start with a non-whitespace character.
3. Regular lines are of form `tag:value`, i.e., have a name tag and a value for the field, separated by `:` (only the first `:` counts). The value can be empty (=whitespace only).

4. Lines starting with whitespace are continuation lines (to the preceding field) if at least one character in the line is non-whitespace.
5. Records are separated by one or more empty (=whitespace only) lines.

`read.dcf` returns a character matrix with one line per record and one column per field. Leading and trailing whitespace of field values is ignored. If a tag name is specified, but the corresponding value is empty, then an empty string of length 0 is returned. If the tag name of a fields is never used in a record, then NA is returned.

### See Also

[write.table](#).

### Examples

```
## Create a reduced version of the 'CONTENTS' file in package 'splines'
x <- read.dcf(file = system.file("CONTENTS", package = "splines"),
              fields = c("Entry", "Description"))
write.dcf(x)
```

---

debug

*Debug a function*

---

### Description

Set or unset the debugging flag on a function.

### Usage

```
debug(fun)
debug(fun)
```

### Arguments

`fun` any interpreted R function.

### Details

When a function flagged for debugging is entered, normal execution is suspended and the body of function is executed one statement at a time. A new browser context is initiated for each step (and the previous one destroyed). You take the next step by typing carriage return, `n` or `next`. You can see the values of variables by typing their names. Typing `c` or `cont` causes the debugger to continue to the end of the function (or loop if within a loop). You can debug new functions before you step in to them from inside the debugger. Typing `Q` quits the current execution and returns you to the top-level prompt. Typing `where` causes the debugger to print out the current stack trace (all functions that are active). If you have variables with names that are identical to the controls (eg. `c` or `n`) then you need to use `print(c)` and `print(n)` to evaluate them.

**See Also**

[browser](#), [traceback](#) to see the stack after an `Error: ... message`; [recover](#) for another debugging approach.

---

 Defunct

*Marking Objects as Defunct*


---

**Description**

When an object is removed from R it should be replaced by a call to `.Defunct`.

**Usage**

```
.Defunct(new, package = NULL)
```

**Arguments**

<code>new</code>	character string: A suggestion for a replacement function.
<code>package</code>	character string: The package to be used when suggesting where the defunct function might be listed.

**Details**

`.Defunct` is called from defunct functions. Functions should be listed in `help("pkg-defunct")` for an appropriate `pkg`, including `base`.

**See Also**

[Deprecated](#).

`base-defunct` and so on which list the defunct functions in the packages.

---

 delay

*Delay Evaluation*


---

**Description**

`delay` creates a *promise* to evaluate the given expression in the specified environment if its value is requested. This provides direct access to *lazy evaluation* mechanism used by R for the evaluation of (interpreted) functions.

**Usage**

```
delay(x, env = .GlobalEnv)
```

**Arguments**

`x` an expression.  
`env` an evaluation environment

**Details**

If promises are kept inside an `environment` or `list`, they can be accessed in several ways without evaluation, see the examples below.

When a promise is eventually forced, it is evaluated within the environment specified by `env` (who contents may have changed in the meantime).

**Value**

A *promise* to evaluate the expression. The value which is returned by `delay` can be assigned without forcing its evaluation, but any further accesses will cause evaluation.

**Examples**

```
x <- delay({
  for(i in 1:7)
    cat("yippee!\n")
  10
})

x^2 #- yippee
x^2 #- simple number

e <- (function(x, y = 1, z) environment()(1+2, "y", {cat(" HO! "); pi+2}))
(le <- as.list(e)) # a list with three promise components
utils::str(le)    # even shows what's promised

le$z # prints; does not eval
ez <- le$z
ez-2 # "HO!", pi
ez # 5.14
le$z # still promise
```

---

deparse

*Expression Deparsing*


---

**Description**

Turn unevaluated expressions into character strings.

**Usage**

```
deparse(expr, width.cutoff = 60,
        backtick = mode(expr) %in% c("call", "expression", "("),
        control = "showAttributes")
```

**Arguments**

<code>expr</code>	any R expression.
<code>width.cutoff</code>	integer in [20, 500] determining the cutoff at which line-breaking is tried.
<code>backtick</code>	logical indicating whether symbolic names should be enclosed in backticks if they don't follow the standard syntax.
<code>control</code>	character vector of deparsing options. See <code>.deparseOpts</code> .

**Details**

This function turns unevaluated expressions (where “expression” is taken in a wider sense than the strict concept of a vector of mode “expression” used in `expression`) into character strings (a kind of inverse `parse`).

A typical use of this is to create informative labels for data sets and plots. The example shows a simple use of this facility. It uses the functions `deparse` and `substitute` to create labels for a plot which are character string versions of the actual arguments to the function `myplot`.

The default for the `backtick` option is not to quote single symbols but only composite expressions. This is a compromise to avoid breaking existing code.

Using `control = "all"` comes closest to making `deparse()` an inverse of `parse()`. However, not all objects are deparseable even with this option and a warning will be issued if the function recognizes that it is being asked to do the impossible.

**Note**

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be deparsed as an attribute.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`substitute`, `parse`, `expression`.

**Examples**

```
require(stats)
deparse(args(lm))
deparse(args(lm), width = 500)
myplot <-
function(x, y) {
  plot(x, y, xlab=deparse(substitute(x)),
       ylab=deparse(substitute(y)))
}
e <- quote(`foo bar`)
deparse(e)
deparse(e, backtick=TRUE)
```

```
e <- quote(`foo bar`+1)
deparse(e)
deparse(e, control = "all")
```

---

deparseOpts                      *Options for Expression Deparsing*

---

## Description

Process the deparsing options for `deparse`, `dput` and `dump`.

## Usage

```
.deparseOpts(control)
```

## Arguments

`control`                      character vector of deparsing options.

## Details

This is called by `deparse`, `dput` and `dump` to process their `control` argument.

The `control` argument is a vector containing zero or more of the following strings. Partial string matching is used.

**keepInteger** Surround integer vectors by `as.integer()`, so they are not converted to floating point when re-parsed.

**quoteExpressions** Surround expressions with `quote()`, so they are not evaluated when re-parsed.

**showAttributes** If the object has attributes (other than a `source` attribute), use `structure()` to display them as well as the object value. This is the default for `deparse` and `dput`.

**useSource** If the object has a `source` attribute, display that instead of deparsing the object. Currently only applies to function definitions.

**warnIncomplete** Some exotic objects such as `environments`, external pointers, etc. can not be deparsed properly. This option causes a warning to be issued if any of those may give problems.

**all** An abbreviated way to specify all of the options listed above. May not be used with other options. This is the default for `dump`.

**delayPromises** Deparse promises as calls to `delay` rather than evaluating them. If the environment of a promise is other than the base or global environments, it is represented by `<environment>` and the deparsed code cannot be sourced.

For the most readable (but perhaps incomplete) display, use `control = NULL`. This displays the object's value, but not its attributes. The default is to display the attributes as well, but not to use any of the other options to make the result parseable.

Using `control = "all"` comes closest to making `deparse()` an inverse of `parse()`. However, not all objects are deparseable even with this option. A warning will be issued if the function recognizes that it is being asked to do the impossible.

### Value

A numerical value corresponding to the options selected.

---

Deprecated

*Marking Objects as Deprecated*

---

### Description

When an object is about removed from R it is first deprecated and should include a call to `.Deprecated`.

### Usage

```
.Deprecated(new, package=NULL)
```

### Arguments

<code>new</code>	character string: A suggestion for a replacement function.
<code>package</code>	character string: The package to be used when suggesting where the deprecated function might be listed.

### Details

`.Deprecated("<new name>")` is called from deprecated functions. The original help page for these functions is often available at `help("oldName-deprecated")` (note the quotes). Functions should be listed in `help("pkg-deprecated")` for an appropriate `pkg`, including `base`.

### See Also

[Defunct](#)

[base-deprecated](#) and so on which list the deprecated functions in the packages.

---

`det`*Calculate the Determinant of a Matrix*

---

**Description**

`det` calculates the determinant of a matrix. `determinant` is a generic function that returns separately the modulus of the determinant, optionally on the logarithm scale, and the sign of the determinant.

**Usage**

```
det(x, ...)  
determinant(x, logarithm = TRUE, ...)
```

**Arguments**

<code>x</code>	numeric matrix.
<code>logarithm</code>	logical; if <code>TRUE</code> (default) return the logarithm of the modulus of the determinant.
<code>...</code>	Optional arguments. At present none are used. Previous versions of <code>det</code> allowed an optional <code>method</code> argument. This argument will be ignored but will not produce an error.

**Value**

For `det`, the determinant of `x`. For `determinant`, a list with components

<code>modulus</code>	a numeric value. The modulus (absolute value) of the determinant if <code>logarithm</code> is <code>FALSE</code> ; otherwise the logarithm of the modulus.
<code>sign</code>	integer; either <code>+1</code> or <code>-1</code> according to whether the determinant is positive or negative.

**Note**

Often, computing the determinant is *not* what you should be doing to solve a given problem.

Prior to version 1.8.0 the `det` function had a `method` argument to allow use of either a QR decomposition or an eigenvalue-eigenvector decomposition. The `determinant` function now uses an LU decomposition and the `det` function is simply a wrapper around a call to `determinant`.

**Examples**

```
(x <- matrix(1:4, ncol=2))  
unlist(determinant(x))  
det(x)  
  
det(print(cbind(1,1:3,c(2,0,1))))
```

---

`detach`*Detach Objects from the Search Path*

---

### Description

Detach a database, i.e., remove it from the `search()` path of available R objects. Usually, this is either a `data.frame` which has been `attached` or a package which was required previously.

### Usage

```
detach(name, pos = 2, version)
```

### Arguments

<code>name</code>	The object to detach. Defaults to <code>search()[pos]</code> . This can be a name or a character string but <i>not</i> a character vector.
<code>pos</code>	Index position in <code>search()</code> of database to detach. When <code>name</code> is a number, <code>pos = name</code> is used.
<code>version</code>	A character string denoting a version number of the package to be loaded. If no version is given, a suitable default is chosen.

### Value

The attached database is returned invisibly, either as `data.frame` or as `list`.

### Note

You cannot detach either the workspace (position 1) or the **base** package (the last item in the search list).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[attach](#), [library](#), [search](#), [objects](#).

### Examples

```
require(splines)#package
detach(package:splines)
## could equally well use detach("package:splines")
## but NOT pkg <- "package:splines"; detach(pkg)
## Instead, use
library(splines)
pkg <- "package:splines"
```



**See Also**

[upper.tri](#), [lower.tri](#), [matrix](#).

**Examples**

```
require(stats)
dim(diag(3))
diag(10,3,4) # guess what?
all(diag(1:3) == {m <- matrix(0,3,3); diag(m) <- 1:3; m})

diag(var(M <- cbind(X=1:5, Y=rnorm(5))))#-> vector with names "X" and "Y"
rownames(M) <- c(colnames(M),rep("",3));
M; diag(M) # named as well
```

---

diff

*Lagged Differences*


---

**Description**

Returns suitably lagged and iterated differences.

**Usage**

```
diff(x, ...)

## Default S3 method:
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'POSIXt':
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'Date':
diff(x, lag = 1, differences = 1, ...)
```

**Arguments**

**x** a numeric vector or matrix containing the values to be differenced.  
**lag** an integer indicating which lag to use.  
**differences** an integer indicating the order of the difference.  
**...** further arguments to be passed to or from methods.

**Details**

diff is a generic function with a default method and ones for classes "[ts](#)", "[POSIXt](#)" and "[Date](#)".

[NA](#)'s propagate.

**Value**

If `x` is a vector of length `n` and `differences=1`, then the computed result is equal to the successive differences `x[(1+lag):n] - x[1:(n-lag)]`.

If `difference` is larger than one this algorithm is applied recursively to `x`. Note that the returned value is a vector which is shorter than `x`.

If `x` is a matrix then the difference operations are carried out on each column separately.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`diff.ts`, `diffinv`.

**Examples**

```
diff(1:10, 2)
diff(1:10, 2, 2)
x <- cumsum(cumsum(1:10))
diff(x, lag = 2)
diff(x, differences = 2)

diff(.leap.seconds)
```

---

difftime

*Time Intervals*

---

**Description**

Create, print and round time intervals.

**Usage**

```
time1 - time2

difftime(time1, time2, tz = "",
          units = c("auto", "secs", "mins", "hours", "days", "weeks"))

as.difftime(tim, format = "%X")

## S3 method for class 'difftime':
round(x, digits = 0)
```

**Arguments**

<code>time1, time2</code>	date-time objects.
<code>tz</code>	a timezone specification to be used for the conversion. System-specific, but "" is the current time zone, and "GMT" is UTC.
<code>units</code>	character. Units in which the results are desired. Can be abbreviated.
<code>tim</code>	character string specifying a time interval.
<code>format</code>	character specifying the format of <code>tim</code> .
<code>x</code>	an object inheriting from class "difftime".
<code>digits</code>	integer. Number of significant digits to retain.

**Details**

Function `difftime` takes a difference of two date/time objects (of either class) and returns an object of class "difftime" with an attribute indicating the units. There is a [round](#) method for objects of this class, as well as methods for the group-generic (see [Ops](#)) logical and arithmetic operations.

If `units = "auto"`, a suitable set of units is chosen, the largest possible (excluding "weeks") in which all the absolute differences are greater than one.

Subtraction of two date-time objects gives an object of this class, by calling `difftime` with `units="auto"`. Alternatively, `as.difftime()` works on character-coded time intervals.

Limited arithmetic is available on "difftime" objects: they can be added or subtracted, and multiplied or divided by a numeric vector. In addition, adding or subtracting a numeric vector implicitly converts the numeric vector to a "difftime" object with the same units as the "difftime" object.

**See Also**

[DateTimeClasses](#).

**Examples**

```
(z <- Sys.time() - 3600)
Sys.time() - z           # just over 3600 seconds.

## time interval between releases of 1.2.2 and 1.2.3.
ISOdate(2001, 4, 26) - ISOdate(2001, 2, 26)

as.difftime(c("0:3:20", "11:23:15"))
as.difftime(c("3:20", "23:15", "2:"), format= "%H:%M")# 3rd gives NA
```

---

dim	<i>Dimensions of an Object</i>
-----	--------------------------------

---

**Description**

Retrieve or set the dimension of an object.

**Usage**

```
dim(x)
dim(x) <- value
```

**Arguments**

x	an R object, for example a matrix, array or data frame.
value	For the default method, either NULL or a numeric vector which coerced to integer (by truncation).

**Details**

The functions `dim` and `dim<-` are generic.

`dim` has a method for `data.frames`, which returns the length of the `row.names` attribute of `x` and the length of `x` (the numbers of “rows” and “columns”).

**Value**

For an array (and hence in particular, for a matrix) `dim` retrieves the `dim` attribute of the object. It is NULL or a vector of mode `integer`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`ncol`, `nrow` and `dimnames`.

**Examples**

```
x <- 1:12 ; dim(x) <- c(3,4)
x

# simple versions of nrow and ncol could be defined as follows
nrow0 <- function(x) dim(x)[1]
ncol0 <- function(x) dim(x)[2]
```

---

dimnames                      *Dimnames of an Object*

---

### Description

Retrieve or set the dimnames of an object.

### Usage

```
dimnames(x)
dimnames(x) <- value
```

### Arguments

`x`                      an R object, for example a matrix, array or data frame.  
`value`                  a possible value for `dimnames(x)`: see “Value”.

### Details

The functions `dimnames` and `dimnames<-` are generic.

For an [array](#) (and hence in particular, for a [matrix](#)), they retrieve or set the `dimnames` attribute (see [attributes](#)) of the object. The list `value` can have names, and these will be used to label the dimensions of the array where appropriate.

Both have methods for data frames. The `dimnames` of a data frame are its `row.names` attribute and its `names`.

As from R 1.8.0 factor components of `value` will be coerced to character.

### Value

The `dimnames` of a matrix or array can be `NULL` or a list of the same length as `dim(x)`. If a list, its components are either `NULL` or a character vector the length of the appropriate dimension of `x`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[rownames](#), [colnames](#); [array](#), [matrix](#), [data.frame](#).

### Examples

```
## simple versions of rownames and colnames
## could be defined as follows
rownames0 <- function(x) dimnames(x)[[1]]
colnames0 <- function(x) dimnames(x)[[2]]
```

---

`do.call`*Execute a Function Call*

---

**Description**

`do.call` executes a function call from the name of the function and a list of arguments to be passed to it.

**Usage**

```
do.call(what, args)
```

**Arguments**

<code>what</code>	a character string naming the function to be called.
<code>args</code>	a <i>list</i> of arguments to the function call. The names attribute of <code>args</code> gives the argument names.

**Value**

The result of the (evaluated) function call.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[call](#) which creates an unevaluated call.

**Examples**

```
do.call("complex", list(imag = 1:3))
```

---

`double`*Double Precision Vectors*

---

**Description**

Create, coerce to or test for a double-precision vector.

**Usage**

```
double(length = 0)
as.double(x, ...)
is.double(x)

single(length = 0)
as.single(x, ...)
```

**Arguments**

length	desired length.
x	object to be coerced or tested.
...	further arguments passed to or from other methods.

**Value**

double creates a double precision vector of the specified length. The elements of the vector are all equal to 0.

as.double attempts to coerce its argument to be of double type: like [as.vector](#) it strips attributes including names.

is.double returns TRUE or FALSE depending on whether its argument is of double type or not. It is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

**Note**

R has no single precision data type. All real numbers are stored in double precision format. The functions `as.single` and `single` are identical to `as.double` and `double` except they set the attribute `Csingle` that is used in the `.C` and `.Fortran` interface, and they are intended only to be used in that context.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[integer](#).

**Examples**

```
is.double(1)
all(double(3) == 0)
```

---

`dput`*Write an Internal Object to a File*

---

**Description**

Writes an ASCII text representation of an R object to a file or connection, or uses one to recreate the object.

**Usage**

```
dput(x, file = "", control = "showAttributes")
dget(file)
```

**Arguments**

<code>x</code>	an object.
<code>file</code>	either a character string naming a file or a connection. "" indicates output to the console.
<code>control</code>	character vector indicating deparsing options. See <code>.deparseOpts</code> for their description.

**Details**

`dput` opens `file` and deparses the object `x` into that file. The object name is not written (contrary to `dump`). If `x` is a function the associated environment is stripped. Hence scoping information can be lost.

Deparsing an object is difficult, and not always possible. With the default `control = c("showAttributes")`, `dput()` attempts to deparse in a way that is readable, but for more complex or unusual objects, not likely to be parsed as identical to the original. Use `control = "all"` for the most complete deparsing; use `control = NULL` for the simplest deparsing, not even including attributes.

`dput` will warn if fewer characters were written to a file than expected, which may indicate a full or corrupt file system.

To display saved source rather than deparsing the internal representation include `"useSource"` in `control`. R currently saves source only for function definitions.

**Note**

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be written as an attribute.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[deparse](#), [dump](#), [write](#).

**Examples**

```
## Write an ASCII version of mean to the file "foo"
dput(mean, "foo")
## And read it back into 'bar'
bar <- dget("foo")
unlink("foo")
## Create a function with comments
baz <- function(x) {
  # Subtract from one
  1-x
}
## and display it
dput(baz)
## and now display the saved source
dput(baz, control = "useSource")
```

---

drop

*Drop Redundant Extent Information*

---

**Description**

Delete the dimensions of an array which have only one level.

**Usage**

```
drop(x)
```

**Arguments**

`x` an array (including a matrix).

**Value**

If `x` is an object with a `dim` attribute (e.g., a matrix or [array](#)), then `drop` returns an object like `x`, but with any extents of length one removed. Any accompanying `dimnames` attribute is adjusted and returned with `x`.

Array subsetting ([\[](#)) performs this reduction unless used with `drop = FALSE`, but sometimes it is useful to invoke `drop` directly.

**See Also**

[drop1](#) which is used for dropping terms in models.

**Examples**

```
dim(drop(array(1:12, dim=c(1,3,1,1,2,1,2))))# = 3 2 2
drop(1:3 %*% 2:4)# scalar product
```

---

 dump

*Text Representations of R Objects*


---

**Description**

This function takes a vector of names of R objects and produces text representations of the objects on a file or connection. A dump file can usually be [sourced](#) into another R (or S) session.

**Usage**

```
dump(list, file = "dumpdata.R", append = FALSE,
      control = "all", envir = parent.frame(), evaluate = TRUE)
```

**Arguments**

<code>list</code>	character. The names of one or more R objects to be dumped.
<code>file</code>	either a character string naming a file or a connection. " " indicates output to the console.
<code>append</code>	if TRUE, output will be appended to <code>file</code> ; otherwise, it will overwrite the contents of <code>file</code> .
<code>control</code>	character vector indicating deparsing options. See <a href="#">.deparseOpts</a> for their description.
<code>envir</code>	the environment to search for objects.
<code>evaluate</code>	logical. Should promises be evaluated?

**Details**

At present [sourcing](#) may not produce an identical copy of dumped objects. A warning is issued if it is likely that problems will arise, for example when dumping exotic objects such as [environments](#) and external pointers.

`dump` will also warn if fewer characters were written to a file than expected, which may indicate a full or corrupt file system.

A dump file can be [sourced](#) into another R (or perhaps S) session, but the function [save](#) is designed to be used for transporting R data, and will work with R objects that `dump` does not handle.

To produce a more readable representation of an object, use `control = NULL`. This will skip attributes, and will make other simplifications that make `source` less likely to produce an identical copy. See [deparse](#) for details.

To deparse the internal function representation rather than displaying the saved source, use `control = c("keepInteger", "quoteExpressions", "showAttributes",`

"warnIncomplete"). This will lose all formatting and comments, but may be useful in those cases where the saved source is no longer correct.

Promises will normally only be encountered by users as a result of lazy-loading (when the default `evaluate = TRUE` is essential) and after the use of `delay`, when `evaluate = FALSE` might be intended.

### Note

The `envir` argument was added at version 1.7.0, and changed the default search path for named objects to include the environment from which `dump` was called.

As `dump` is defined in the base namespace, the **base** package will be searched *before* the global environment unless `dump` is called from the top level or the `envir` argument is given explicitly.

To avoid the risk of a source attribute out of sync with the actual function definition, the source attribute of a function will never be dumped as an attribute.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`dput`, `dget`, `write`.  
[save](#) for a more reliable way to save R objects.

### Examples

```
x <- 1; y <- 1:10
dump(ls(patt='^[xyz]'), "xyz.Rdumped")
unlink("xyz.Rdumped")
```

---

duplicated

*Determine Duplicate Elements*

---

### Description

Determines which elements of a vector of data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.

### Usage

```
duplicated(x, incomparables = FALSE, ...)

## S3 method for class 'array':
duplicated(x, incomparables = FALSE, MARGIN = 1, ...)
```

**Arguments**

<code>x</code>	an atomic vector or a data frame or an array.
<code>incomparables</code>	a vector of values that cannot be compared. Currently, <code>FALSE</code> is the only possible value, meaning that all values can be compared.
<code>...</code>	arguments for particular methods.
<code>MARGIN</code>	the array margin to be held fixed: see <a href="#">apply</a> .

**Details**

This is a generic function with methods for vectors, data frames and arrays (including matrices).

The data frame method works by pasting together a character representation of the rows separated by `\r`, so may be imperfect if the data frame has characters with embedded carriage returns or columns which do not reliably map to characters.

The array method calculates for each element of the sub-array specified by `MARGIN` if the remaining dimensions are identical to those for an earlier element (in row-major order). This would most commonly be used to find duplicated rows (the default) or columns (with `MARGIN = 2`).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[unique](#).

**Examples**

```
x <- c(9:20, 1:5, 3:7, 0:8)
## extract unique elements
(xu <- x[!duplicated(x)])
## xu == unique(x) but unique(x) is more efficient

duplicated(iris)[140:143]

duplicated(iris3, MARGIN = c(1, 3))
```

**Description**

Load or unload shared libraries, and test whether a C function or Fortran subroutine is available.

**Usage**

```

dyn.load(x, local = TRUE, now = TRUE)
dyn.unload(x)

is.loaded(symbol, PACKAGE = " ")
symbol.C(name)
symbol.For(name)

```

**Arguments**

<code>x</code>	a character string giving the pathname to a shared library or DLL.
<code>local</code>	a logical value controlling whether the symbols in the shared library are stored in their own local table and not shared across shared libraries, or added to the global symbol table. Whether this has any effect is system-dependent.
<code>now</code>	a logical controlling whether all symbols are resolved (and relocated) immediately the library is loaded or deferred until they are used. This control is useful for developers testing whether a library is complete and has all the necessary symbols, and for users to ignore missing symbols. Whether this has any effect is system-dependent.
<code>symbol</code>	a character string giving a symbol name.
<code>PACKAGE</code>	if supplied, confine the search for the name to the DLL given by this argument (plus the conventional extension, <code>.so</code> , <code>.sl</code> , <code>.dll</code> , ...). This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols. Use <code>PACKAGE="base"</code> for symbols linked in to R. This is used in the same way as in <code>.C</code> , <code>.Call</code> , <code>.Fortran</code> and <code>.External</code> functions
<code>name</code>	a character string giving either the name of a C function or Fortran subroutine. Fortran names probably need to be given entirely in lower case (but this may be system-dependent).

**Details**

See ‘See Also’ and the *Writing R Extensions* manual for how to create a suitable shared library. Note that unlike some versions of S-PLUS, `dyn.load` does not load an object (`.o`) file but a shared library or DLL.

Unfortunately a very few platforms (Compaq Tru64) do not handle the `PACKAGE` argument correctly, and may incorrectly find symbols linked into R.

The additional arguments to `dyn.load` mirror the different aspects of the mode argument to the `dlopen()` routine on UNIX systems. They are available so that users can exercise greater control over the loading process for an individual library. In general, the defaults values are appropriate and you should override them only if there is good reason and you understand the implications.

The `local` argument allows one to control whether the symbols in the DLL being attached are visible to other DLLs. While maintaining the symbols in their own namespace is good practice, the ability to share symbols across related “chapters” is useful in many cases. Additionally, on certain platforms and versions of an operating system, certain libraries must have their symbols loaded globally to successfully resolve all symbols.

One should be careful of the potential side-effect of using lazy loading via the `now` argument as `FALSE`. If a routine is called that has a missing symbol, the process will terminate immediately and unsaved session variables will be lost. The intended use is for library developers to call specify a value `TRUE` to check that all symbols are actually resolved and for regular users to all with `FALSE` so that missing symbols can be ignored and the available ones can be called.

The initial motivation for adding these was to avoid such termination in the `_init()` routines of the Java virtual machine library. However, symbols loaded locally may not be (read probably) available to other DLLs. Those added to the global table are available to all other elements of the application and so can be shared across two different DLLs.

Some systems do not provide (explicit) support for local/global and lazy/eager symbol resolution. This can be the source of subtle bugs. One can arrange to have warning messages emitted when unsupported options are used. This is done by setting either of the options `verbose` or `warn` to be non-zero via the `options` function. Currently, we know of only 2 platforms that do not provide a value for local load (`RTLD_LOCAL`). These are IRIX6.4 and unpatched versions of Solaris 2.5.1.

There is a short discussion of these additional arguments with some example code available at <http://cm.bell-labs.com/stat/duncan/R/dynload>.

## Value

The function `dyn.load` is used for its side effect which links the specified shared library to the executing R image. Calls to `.C`, `.Call`, `.Fortran` and `.External` can then be used to execute compiled C functions or Fortran subroutines contained in the library. The return value of `dyn.load` is an object of class `DLLInfo`. See [getLoadedDLLs](#) for information about this class.

The function `dyn.unload` unlinks the shared library.

Functions `symbol.C` and `symbol.For` map function or subroutine names to the symbol name in the compiled code: `is.loaded` checks if the symbol name is loaded and hence available for use in `.C` or `.Fortran`.

## Note

The creation of shared libraries and the runtime linking of them into executing programs is very platform dependent. In recent years there has been some simplification in the process because the C subroutine call `dlopen` has become the standard for doing this under UNIX. Under UNIX `dyn.load` uses the `dlopen` mechanism and should work on all platforms which support it. On Windows it uses the standard mechanisms for loading 32-bit DLLs.

The original code for loading DLLs in UNIX was provided by Heiner Schwarte. The compatibility code for HP-UX was provided by Luke Tierney.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[library.dynam](#) to be used inside a package's `.First.lib` initialization.

[SHLIB](#) for how to create suitable shared objects.

[.C](#), [.Fortran](#), [.External](#), [.Call](#).

## Examples

```
is.loaded(symbol.For("hcass2")) #-> probably TRUE, as stats is loaded
```

---

eapply

*Apply a Function over values in an environment*

---

## Description

`eapply` applies `FUN` to the named values from an environment and returns the results as a list. The user can request that all named objects are used (normally names that begin with a dot are not). The output is not sorted and no parent environments are searched.

## Usage

```
eapply(env, FUN, ..., all.names = FALSE)
```

## Arguments

<code>env</code>	environment to be used.
<code>FUN</code>	the function to be applied. In the case of functions like <code>+</code> , <code>%*%</code> , etc., the function name must be quoted.
<code>...</code>	optional arguments to <code>FUN</code> .
<code>all.names</code>	a logical indicating whether to apply the function to all values

## See Also

[lapply](#).

## Examples

```
env <- new.env()
env$a <- 1:10
env$beta <- exp(-3:3)
env$logic <- c(TRUE,FALSE,FALSE,TRUE)
# compute the list mean for each list element
eapply(env,mean)
# median and quartiles for each list element
eapply(env, quantile, probs = 1:3/4)
eapply(env, quantile)
```

---

eigen

*Spectral Decomposition of a Matrix*


---

**Description**

Computes eigenvalues and eigenvectors.

**Usage**

```
eigen(x, symmetric, only.values = FALSE, EISPACK = FALSE)
```

**Arguments**

<code>x</code>	a matrix whose spectral decomposition is to be computed.
<code>symmetric</code>	if TRUE, the matrix is assumed to be symmetric (or Hermitian if complex) and only its lower triangle is used. If <code>symmetric</code> is not specified, the matrix is inspected for symmetry.
<code>only.values</code>	if TRUE, only the eigenvalues are computed and returned, otherwise both eigenvalues and eigenvectors are returned.
<code>EISPACK</code>	logical. Should EISPACK be used (for compatibility with R < 1.7.0)?

**Details**

By default `eigen` uses the LAPACK routines DSYEVR/DSYEV, DGEEV, ZHEEV and ZGEEV whereas `eigen(EISPACK=TRUE)` provides an interface to the EISPACK routines RS, RG, CH and CG.

If `symmetric` is unspecified, the code attempts to determine if the matrix is symmetric up to plausible numerical inaccuracies. It is faster and surer to set the value yourself.

`eigen` is preferred to `eigen(EISPACK = TRUE)` for new projects, but its eigenvectors may differ in sign and (in the asymmetric case) in normalization. (They may also differ between methods and between platforms.)

In the real symmetric case, LAPACK routine DSYEVR is used which requires IEEE 754 arithmetic. Should this not be supported on your platform, DSYEV is used, with a warning.

Computing the eigenvectors is the slow part for large matrices.

**Value**

The spectral decomposition of `x` is returned as components of a list with components

<code>values</code>	a vector containing the $p$ eigenvalues of <code>x</code> , sorted in <i>decreasing</i> order, according to <code>Mod(values)</code> in the asymmetric case when they might be complex (even for real matrices). For real asymmetric matrices the vector will be complex only if complex conjugate pairs of eigenvalues are detected.
---------------------	---

vectors either a  $p \times p$  matrix whose columns contain the eigenvectors of  $x$ , or NULL if only `values` is TRUE.

For `eigen(, symmetric = FALSE, EISPACK = TRUE)` the choice of length of the eigenvectors is not defined by EISPACK. In all other cases the vectors are normalized to unit length.

Recall that the eigenvectors are only defined up to a constant: even when the length is specified they are still only defined up to a scalar of modulus one (the sign for real matrices).

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Smith, B. T, Boyle, J. M., Dongarra, J. J., Garbow, B. S., Ikebe, Y., Klema, V., and Moler, C. B. (1976). *Matrix Eigensystems Routines – EISPACK Guide*. Springer-Verlag Lecture Notes in Computer Science.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.

Available on-line at [http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html).

## See Also

`svd`, a generalization of `eigen`; `qr`, and `chol` for related decompositions.

To compute the determinant of a matrix, the `qr` decomposition is much more efficient: `det`.

`capabilities` to test for IEEE 754 arithmetic.

## Examples

```
eigen(cbind(c(1,-1),c(-1,1)))
eigen(cbind(c(1,-1),c(-1,1)), symmetric = FALSE)# same (different algorithm).

eigen(cbind(1,c(1,-1)), only.values = TRUE)
eigen(cbind(-1,2:1)) # complex values
eigen(print(cbind(c(0,1i), c(-1i,0))))# Hermite ==> real Eigen values
## 3 x 3:
eigen(cbind( 1,3:1,1:3))
eigen(cbind(-1,c(1:2,0),0:2)) # complex values
```

---

environment

*Environment Access*

---

## Description

Get, set, test for and create environments.

**Usage**

```
environment(fun = NULL)
environment(fun) <- value

is.environment(obj)

.GlobalEnv
globalenv()
.BaseNamespaceEnv

new.env(hash = FALSE, parent = parent.frame())

parent.env(env)
parent.env(env) <- value
```

**Arguments**

<code>fun</code>	a <a href="#">function</a> , a <a href="#">formula</a> , or <code>NULL</code> , which is the default.
<code>value</code>	an environment to associate with the function
<code>obj</code>	an arbitrary R object.
<code>hash</code>	a logical, if <code>TRUE</code> the environment will be hashed
<code>parent</code>	an environment to be used as the parent of the environment created.
<code>env</code>	an environment

**Details**

The global environment `.GlobalEnv` is the first item on the search path, more often known as the user's workspace. It can also be accessed by `globalenv()`.

The object `.BaseNamespaceEnv` is the namespace environment for the base package. The environment of the base package itself is represented by `NULL`.

The replacement function `parent.env<-` is extremely dangerous as it can be used to destructively change environments in ways that violate assumptions made by the internal C code. It may be removed in the near future.

`is.environment` is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

**Value**

If `fun` is a function or a formula then `environment(fun)` returns the environment associated with that function or formula. If `fun` is `NULL` then the current evaluation environment is returned.

The assignment form sets the environment of the function or formula `fun` to the `value` given.

`is.environment(obj)` returns `TRUE` iff `obj` is an environment.

`new.env` returns a new (empty) environment enclosed in the parent's environment, by default.

`parent.env` returns the parent environment of its argument.

`parent.env<-` sets the parent environment of its first argument.

**See Also**

The `envir` argument of `eval`.

**Examples**

```
##-- all three give the same:
environment()
environment(environment)
.GlobalEnv

ls(envir=environment(approxfun(1:2,1:2, method="const")))

is.environment(.GlobalEnv) # TRUE

e1 <- new.env(TRUE, NULL) # these two have parent package:base.
e2 <- new.env(FALSE, NULL)
assign("a", 3, env=e2)
parent.env(e1) <- e2
get("a", env=e1)
```

---

 eval

*Evaluate an (Unevaluated) Expression*


---

**Description**

Evaluate an R expression in a specified environment.

**Usage**

```
eval(expr, envir = parent.frame(),
      enclos = if(is.list(envir) || is.pairlist(envir)) parent.frame())
evalq(expr, envir, enclos)
eval.parent(expr, n = 1)
local(expr, envir = new.env())
```

**Arguments**

<code>expr</code>	object of mode <a href="#">expression</a> or call or an “unevaluated expression”.
<code>envir</code>	the <a href="#">environment</a> in which <code>expr</code> is to be evaluated. May also be a list, a data frame, or an integer as in <code>sys.call</code> .
<code>enclos</code>	Relevant when <code>envir</code> is a list or a data frame. Specifies the enclosure, i.e., where R looks for objects not found in <code>envir</code> .
<code>n</code>	parent generations to go back

## Details

`eval` evaluates the expression `expr` argument in the environment specified by `envir` and returns the computed value. If `envir` is not specified, then `sys.frame(sys.frame())`, the environment where the call to `eval` was made is used.

The `evalq` form is equivalent to `eval(quote(expr), ...)`.

As `eval` evaluates its first argument before passing it to the evaluator, it allows you to assign complicated expressions to symbols and then evaluate them. `evalq` avoids this.

`eval.parent(expr, n)` is a shorthand for `eval(expr, parent.frame(n))`.

If `envir` is a data frame or list, it is copied into a temporary environment, and the copy is used for evaluation. So if `expr` changes any of the components named in the data frame/list, the changes are lost.

`local` evaluates an expression in a local environment. It is equivalent to `evalq` except the its default argument creates a new, empty environment. This is useful to create anonymous recursive functions and as a kind of limited namespace feature since variables defined in the environment are not visible from the outside.

## Note

Due to the difference in scoping rules, there are some differences between R and S in this area. In particular, the default enclosure in S is the global environment.

When evaluating expressions in data frames that has been passed as argument to a function, the relevant enclosure is often the caller's environment, i.e., one needs `eval(x, data, parent.frame())`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`eval` only.)

## See Also

[expression](#), [quote](#), [sys.frame](#), [parent.frame](#), [environment](#).

Further, [force](#) to *force* evaluation, typically of function arguments.

## Examples

```
eval(2 ^ 2 ^ 3)
mEx <- expression(2^2^3); mEx; 1 + eval(mEx)
eval({ xx <- pi; xx^2}) ; xx

a <- 3 ; aa <- 4 ; evalq(evalq(a+b+aa, list(a=1)), list(b=5)) # == 10
a <- 3 ; aa <- 4 ; evalq(evalq(a+b+aa, -1), list(b=5))      # == 12

ev <- function() {
  e1 <- parent.frame()
  ## Evaluate a in e1
  aa <- eval(expression(a),e1)
  ## evaluate the expression bound to a in e1
```

```

    a <- expression(x+y)
    list(aa = aa, eval = eval(a, e1))
  }
tst.ev <- function(a = 7) { x <- pi; y <- 1; ev() }
tst.ev()#-> aa : 7,  eval : 4.14

##
## Uses of local()
##

# Mutual recursives.
# gg gets value of last assignment, an anonymous version of f.

gg <- local({
  k <- function(y)f(y)
  f <- function(x) if(x) x*k(x-1) else 1
})
gg(10)
sapply(1:5, gg)

# Nesting locals. a is private storage accessible to k
gg <- local({
  k <- local({
    a <- 1
    function(y){print(a <- a+1);f(y)}
  })
  f <- function(x) if(x) x*k(x-1) else 1
})
sapply(1:5, gg)

ls(envir=environment(gg))
ls(envir=environment(get("k", envir=environment(gg))))

```

---

exists

*Is an Object Defined?*


---

### Description

Look for an R object of the given name.

### Usage

```
exists(x, where = -1, envir = , frame, mode = "any", inherits = TRUE)
```

### Arguments

**x** a variable name (given as a character string).

**where** where to look for the object (see the details section); if omitted, the function will search as if the name of the object appeared unquoted in an expression.

<code>envir</code>	an alternative way to specify an environment to look in, but it's usually simpler to just use the <code>where</code> argument.
<code>frame</code>	a frame in the calling list. Equivalent to giving <code>where</code> as <code>sys.frame(frame)</code> .
<code>mode</code>	the mode of object sought.
<code>inherits</code>	should the enclosing frames of the environment be searched?

### Details

The `where` argument can specify the environment in which to look for the object in any of several ways: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an `environment` (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

This function looks to see if the name `x` has a value bound to it in the specified environment. If `inherits` is `TRUE` and a value is not found for `x` in the specified environment, the enclosing frames of the environment (and the search path) are searched until the name `x` is encountered. **Warning:** `inherits=TRUE` is the default behaviour for R but not for S.

If `mode` is specified then only objects of that mode are sought. The `mode` may specify collections such as `"numeric"` and `"function"`: any member of the collection will suffice.

### Value

Logical, true if and only if an object of the correct name and mode is found.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[get](#).

### Examples

```
## Define a substitute function if necessary:
if(!exists("some.fun", mode="function"))
  some.fun <- function(x) { cat("some.fun(x)\n"); x }
search()
exists("ls", 2) # true even though ls is in pos=3
exists("ls", 2, inherits = FALSE) # false
```

---

<code>expand.grid</code>	<i>Create a Data Frame from All Combinations of Factors</i>
--------------------------	---

---

**Description**

Create a data frame from all combinations of the supplied vectors or factors. See the description of the return value for precise details of the way this is done.

**Usage**

```
expand.grid(...)
```

**Arguments**

...                    Vectors, factors or a list containing these.

**Value**

A data frame containing one row for each combination of the supplied factors. The first factors vary fastest. The columns are labelled by the factors if these are supplied as named arguments or named components of a list.

Attribute "out.attrs" is a list which gives the dimension and dimnames for use by [predict](#) methods.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**Examples**

```
expand.grid(height = seq(60, 80, 5), weight = seq(100, 300, 50),
             sex = c("Male", "Female"))
```

---

<code>expression</code>	<i>Unevaluated Expressions</i>
-------------------------	--------------------------------

---

**Description**

Creates or tests for objects of mode "expression".

**Usage**

```
expression(...)
```

```
is.expression(x)
```

```
as.expression(x, ...)
```

**Arguments**

`...` valid R expressions.  
`x` an arbitrary R object.

**Value**

`expression` returns a vector of mode "expression" containing its arguments as unevaluated "calls".

`is.expression` returns TRUE if `expr` is an expression object and FALSE otherwise.

`as.expression` attempts to coerce its argument into an expression object.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`call`, `eval`, `function`. Further, `text` and `legend` for plotting math expressions.

**Examples**

```
length(ex1 <- expression(1+ 0:9))# 1
ex1
eval(ex1)# 1:10

length(ex3 <- expression(u,v, 1+ 0:9))# 3
mode(ex3 [3]) # expression
mode(ex3[[3]])# call
rm(ex3)
```

---

 Extract

---

*Extract or Replace Parts of an Object*


---

**Description**

Operators acting on vectors, arrays and lists to extract or replace subsets.

**Usage**

```
x[i]
x[i, j, ... , drop = TRUE]
x[[i]]
x[[i, j, ...]]
x$name
```

## Arguments

<code>x</code>	object from which to extract elements or in which to replace elements.
<code>i, j, ..., name</code>	elements to extract or replace. <code>i, j</code> are <code>numeric</code> or <code>character</code> or empty whereas <code>name</code> must be <code>character</code> or an (unquoted) name. Numeric values are coerced to integer as by <code>as.integer</code> . For [-indexing only: <code>i, j, ...</code> can be logical vectors, indicating elements/slices to select. Such vectors are recycled if necessary to match the corresponding extent. When indexing arrays, <code>i</code> can be a (single) matrix with as many columns as there are dimensions of <code>x</code> ; the result is then a vector with elements corresponding to the sets of indices in each row of <code>i</code> . <code>i, j, ...</code> can also be negative integers, indicating elements/slices to leave out of the selection.
<code>drop</code>	For matrices, and arrays. If <code>TRUE</code> the result is coerced to the lowest possible dimension (see examples below). This only works for extracting elements, not for the replacement forms.

## Details

These operators are generic. You can write methods to handle subsetting of specific classes of objects, see [InternalMethods](#) as well as `[.data.frame]` and `[.factor]`. The descriptions here apply only to the default methods.

The most important distinction between `[`, `[[` and `$` is that the `[` can select more than one element whereas the other two select a single element. `$` does not allow computed indices, whereas `[[` does. `x$name` is equivalent to `x[["name"]]` if `x` is recursive (see `is.recursive`) and `NULL` otherwise.

The `[[` operator requires all relevant subscripts to be supplied. With the `[` operator an empty index (a comma separated blank) indicates that all entries in that dimension are selected.

If one of these expressions appears on the left side of an assignment then that part of `x` is set to the value of the right hand side of the assignment.

Indexing by factors is allowed and is equivalent to indexing by the numeric codes (see `factor`) and not by the character values which are printed (for which use `[as.character(i)]`).

When operating on a list, the `[[` operator gives the specified element of the list while the `[` operator returns a list with the specified element(s) in it.

As from R 1.7.0 `[[` can be applied recursively to lists, so that if the single index `i` is a vector of length `p`, `alist[[i]]` is equivalent to `alist[[i1]]...[[ip]]` providing all but the final indexing results in a list.

The operators `$` and `$<-` do not evaluate their second argument. It is translated to a string and that string is used to locate the correct component of the first argument.

When `$<-` is applied to a `NULL` `x`, it coerces `x` to `list()`. This is what happens with `[[<-` if `y` is of length greater than one: if `y` has length 1 or 0, `x` is coerced to a zero-length vector of the type of `value`,

As from R 1.9.0 both `$` and `[[` can be applied to environments. Only character arguments are allowed and no partial matching is done (this is in contrast to the behavior for lists). The semantics

of these operations is basically that of `get(i, env=x, inherits=FALSE)`. If no match is found then `NULL` is returned. The assignment versions, `$<-` and `[<-`, can also be used. Again, only character arguments are allowed and no partial matching is done. The semantics in this case are those of `assign(i, value, env=x, inherits=FALSE)`. Such an assignment will either create a new binding or change the existing binding in `x`.

### NAs in indexing

When subscripting, a numerical, logical or character NA picks an unknown element and so returns NA in the corresponding element of a logical, integer, numeric, complex or character result, and `NULL` for a list.

When replacing (that is using subscripting on the lhs of an assignment) NA does not select any element to be replaced. As there is ambiguity as to whether an element of the rhs should be used or not (and R handled this inconsistently prior to R 2.0.0), this is only allowed if the rhs value is of length one (so the two interpretations would have the same outcome).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[list](#), [array](#), [matrix](#).

[\[.data.frame\]](#) and [\[.factor\]](#) for the behaviour when applied to `data.frame` and `factors`.

[Syntax](#) for operator precedence, and the *R Language* reference manual about indexing details.

### Examples

```
x <- 1:12; m <- matrix(1:6,nr=2); li <- list(pi=pi, e = exp(1))
x[10]                # the tenth element of x
x <- x[-1]           # delete the 1st element of x
m[1,]                # the first row of matrix m
m[1, , drop = FALSE] # is a 1-row matrix
m[,c(TRUE,FALSE,TRUE)] # logical indexing
m[cbind(c(1,2,1),3:1)] # matrix index
m <- m[,-1]          # delete the first column of m
li[[1]]              # the first element of list li
y <- list(1,2,a=4,5)
y[c(3,4)]            # a list containing elements 3 and 4 of y
y$a                  # the element of y named a

## non-integer indices are truncated:
(i <- 3.999999999) # "4" is printed
(1:5)[i]           # 3

## recursive indexing into lists
z <- list( a=list( b=9, c='hello'), d=1:5)
unlist(z)
z[[c(1, 2)]]
```

```
z[[c(1, 2, 1)]] # both "hello"
z[[c("a", "b")]] <- "new"
unlist(z)

## check $ and [[ for environments
e1 <- new.env()
e1$a <- 10
e1[["a"]]
e1[["b"]] <- 20
e1$b
ls(e1)
```

---

Extract.data.frame *Extract or Replace Parts of a Data Frame*

---

## Description

Extract or replace subsets of data frames.

## Usage

```
x[i]
x[i] <- value
x[i, j, drop = TRUE]
x[i, j] <- value

x[[i]]
x[[i]] <- value
x[[i, j]]
x[[i, j]] <- value

x$name
x$name <- value
```

## Arguments

<code>x</code>	data frame.
<code>i, j</code>	elements to extract or replace. <code>i, j</code> are numeric or character or, for <code>[</code> only, empty. Numeric values are coerced to integer as if by <a href="#">as.integer</a> . For replacement by <code>[</code> , a logical matrix is allowed.
<code>drop</code>	logical. If <code>TRUE</code> the result is coerced to the lowest possible dimension: however, see the Warning below.
<code>value</code>	A suitable replacement value: it will be repeated a whole number of times if necessary and it may be coerced: see the Coercion section. If <code>NULL</code> , deletes the column if a single column is selected.
<code>name</code>	name or literal character string.

## Details

Data frames can be indexed in several modes. When `[` and `[[` are used with a single index, they index the data frame as if it were a list. In this usage a `drop` argument is ignored, with a warning. Using `$` is equivalent to using `[[` with a single index.

When `[` and `[[` are used with two indices they act like indexing a matrix: `[[` can only be used to select one element.

If `[` returns a data frame it will have unique (and non-missing) row names, if necessary transforming the row names using `make.unique`. Similarly, column names will be transformed (if columns are selected more than once).

When `drop = TRUE`, this is applied to the subsetting of any matrices contained in the data frame as well as to the data frame itself.

The replacement methods can be used to add whole column(s) by specifying non-existent column(s), in which case the column(s) are added at the right-hand edge of the data frame and numerical indices must be contiguous to existing indices. On the other hand, rows can be added at any row after the current last row, and the columns will be in-filled with missing values. Missing values in the indices are not allowed for replacement.

For `[` the replacement value can be a list: each element of the list is used to replace (part of) one column, recycling the list as necessary. If the columns specified by number are created, the names (if any) of the corresponding list elements are used to name the columns. If the replacement is not selecting rows, list values can contain `NULL` elements which will cause the corresponding columns to be deleted.

Matrixing indexing using `[` is not recommended, and barely supported. For extraction, `x` is first coerced to a matrix. For replacement a logical matrix (only) can be used to select the elements to be replaced in the same ways as for a matrix (except that missing values are not allowed at all).

## Value

For `[` a data frame, list or a single column (the latter two only when dimensions have been dropped). If matrix indexing is used for extraction a matrix results.

For `[[` a column of the data frame (extraction with one index) or a length-one vector (extraction with two indices).

For `[<-`, `[[<-` and `$<-`, a data frame.

## Coercion

The story over when replacement values are coerced is a complicated one, and one that has changed during `R`'s development. This section is a guide only.

When `[` and `[[` are used to add or replace a whole column, no coercion takes place but `value` will be replicated (by calling the generic function `rep`) to the right length if an exact number of repeats can be used.

When `[` is used with a logical matrix, each value is coerced to the type of the column in which it is to be placed.

When `[` and `[[` are used with two indices, the column will be coerced as necessary to accommodate the value.

Note that when the replacement value is an array (including a matrix) it is *not* treated as a series of columns (as `data.frame` and `as.data.frame` do) but inserted as a single column.

### Warning

Although the default for `drop` is `TRUE`, the default behaviour when only one *row* is left is equivalent to specifying `drop = FALSE`. To drop from a data frame to a list, `drop = FALSE` has to be specified explicitly.

### See Also

`subset` which is often easier for extraction, `data.frame`, `Extract`.

### Examples

```
sw <- swiss[1:5, 1:4] # select a manageable subset

sw[1:3]           # select columns
sw[, 1:3]         # same
sw[4:5, 1:3]     # select rows and columns
sw[1]             # a one-column data frame
sw[, 1, drop = FALSE] # the same
sw[, 1]           # a (unnamed) vector
sw[[1]]           # the same

sw[1,]           # a one-row data frame
sw[1,, drop=TRUE] # a list

swiss[ c(1, 1:2), ] # duplicate row, unique row names are created

sw[sw <= 6] <- 6 # logical matrix indexing
sw

## adding a column
sw["new1"] <- LETTERS[1:5] # adds a character column
sw[["new2"]] <- letters[1:5] # ditto
sw[, "new3"] <- LETTERS[1:5] # ditto
# but this got converted to a factor in 1.7.x

sw$new4 <- 1:5
sapply(sw, class)
sw$new4 <- NULL # delete the column
sw
sw[6:8] <- list(letters[10:14], NULL, aa=1:5) # delete col7, update 6, append
sw

## matrices in a data frame
A <- data.frame(x=1:3, y=I(matrix(4:6)), z=I(matrix(letters[1:9],3,3)))
A[1:3, "y"] # a matrix, was a vector prior to 1.8.0
A[1:3, "z"] # a matrix
A[, "y"]    # a matrix
```

---

Extract.factor      *Extract or Replace Parts of a Factor*

---

## Description

Extract or replace subsets of factors.

## Usage

```
x[i, drop = FALSE]
```

```
x[i] <- value
```

## Arguments

x	a factor
i	a specification of indices – see <a href="#">Extract</a> .
drop	logical. If true, unused levels are dropped.
value	character: a set of levels. Factor values are coerced to character.

## Details

When unused levels are dropped the ordering of the remaining levels is preserved.

If value is not in levels(x), a missing value is assigned with a warning.

Any [contrasts](#) assigned to the factor are preserved unless drop=TRUE.

## Value

A factor with the same set of levels as x unless drop=TRUE.

## See Also

[factor](#), [Extract](#).

## Examples

```
## following example(factor)
(ff <- factor(substring("statistics", 1:10, 1:10), levels=letters))
ff[, drop=TRUE]
factor(letters[7:10])[2:3, drop = TRUE]
```

**Description**

Returns the (parallel) maxima and minima of the input values.

**Usage**

```
max(..., na.rm=FALSE)
min(..., na.rm=FALSE)

pmax(..., na.rm=FALSE)
pmin(..., na.rm=FALSE)
```

**Arguments**

`...` numeric arguments.  
`na.rm` a logical indicating whether missing values should be removed.

**Value**

`max` and `min` return the maximum or minimum of all the values present in their arguments, as [integer](#) if all are [integer](#), or as [double](#) otherwise.

The minimum and maximum of an empty set are `+Inf` and `-Inf` (in this order!) which ensures *transitivity*, e.g., `min(x1, min(x2)) == min(x1, x2)`. In R versions before 1.5, `min(integer(0)) == .Machine$integer.max`, and analogously for `max`, preserving argument *type*, whereas from R version 1.5.0, `max(x) == -Inf` and `min(x) == +Inf` whenever `length(x) == 0` (after removing missing values if requested).

If `na.rm` is `FALSE` an NA value in any of the arguments will cause a value of NA to be returned, otherwise NA values are ignored.

`pmax` and `pmin` take several vectors (or matrices) as arguments and return a single vector giving the parallel maxima (or minima) of the vectors. The first element of the result is the maximum (minimum) of the first elements of all the arguments, the second element of the result is the maximum (minimum) of the second elements of all the arguments and so on. Shorter vectors are recycled if necessary. If `na.rm` is `FALSE`, NA values in the input vectors will produce NA values in the output. If `na.rm` is `TRUE`, NA values are ignored. [attributes](#) (such as [names](#) or [dim](#)) are transferred from the first argument (if applicable).

`max` and `min` are generic functions: methods can be defined for them individually or via the [Summary](#) group generic.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[range](#) (both min and max) and [which.min](#) ([which.max](#)) for the *arg min*, i.e., the location where an extreme value occurs.

**Examples**

```
require(stats)
min(5:1, pi)
pmin(5:1, pi)
x <- sort(rnorm(100)); cH <- 1.35
pmin(cH, quantile(x)) # no names
pmin(quantile(x), cH) # has names
plot(x, pmin(cH, pmax(-cH, x)), type='b', main= "Huber's function")
```

---

 factor

*Factors*


---

**Description**

The function `factor` is used to encode a vector as a factor (the terms ‘category’ and ‘enumerated type’ are also used for factors). If `ordered` is `TRUE`, the factor levels are assumed to be ordered. For compatibility with `S` there is also a function `ordered`.

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

**Usage**

```
factor(x, levels = sort(unique.default(x), na.last = TRUE),
      labels = levels, exclude = NA, ordered = is.ordered(x))
ordered(x, ...)

is.factor(x)
is.ordered(x)

as.factor(x)
as.ordered(x)
```

**Arguments**

<code>x</code>	a vector of data, usually taking a small number of distinct values
<code>levels</code>	an optional vector of the values that <code>x</code> might have taken. The default is the set of values taken by <code>x</code> , sorted into increasing order.
<code>labels</code>	<i>either</i> an optional vector of labels for the levels (in the same order as <code>levels</code> after removing those in <code>exclude</code> ), <i>or</i> a character string of length 1.
<code>exclude</code>	a vector of values to be excluded when forming the set of levels. This should be of the same type as <code>x</code> , and will be coerced if necessary.

`ordered`            logical flag to determine if the levels should be regarded as ordered (in the order given).  
 . . .                (in `ordered( . )`): any of the above, apart from `ordered` itself.

### Details

The type of the vector `x` is not restricted.

Ordered factors differ from factors only in their class, but methods and the model-fitting functions treat the two classes quite differently.

The encoding of the vector happens as follows. First all the values in `exclude` are removed from `levels`. If `x[i]` equals `levels[j]`, then the *i*-th element of the result is *j*. If no match is found for `x[i]` in `levels`, then the *i*-th element of the result is set to `NA`.

Normally the ‘levels’ used as an attribute of the result are the reduced set of levels after removing those in `exclude`, but this can be altered by supplying `labels`. This should either be a set of new labels for the levels, or a character string, in which case the levels are that character string with a sequence number appended.

`factor(x, exclude=NULL)` applied to a factor is a no-operation unless there are unused levels: in that case, a factor with the reduced level set is returned. If `exclude` is used it should also be a factor with the same level set as `x` or a set of codes for the levels to be excluded.

The codes of a factor may contain `NA`. For a numeric `x`, set `exclude=NULL` to make `NA` an extra level ("`NA`"), by default the last level.

If "`NA`" is a level, the way to set a code to be missing is to use `is.na` on the left-hand-side of an assignment. Under those circumstances missing values are printed as `<NA>`.

`is.factor` is generic: you can write methods to handle specific classes of objects, see [Internal-Methods](#).

### Value

`factor` returns an object of class "`factor`" which has a set of integer codes the length of `x` with a "`levels`" attribute of mode `character`. If `ordered` is true (or `ordered` is used) the result has class `c("ordered", "factor")`.

Applying `factor` to an ordered or unordered factor returns a factor (of the same type) with just the levels which occur: see also [\[.factor\]](#) for a more transparent way to achieve this.

`is.factor` returns `TRUE` or `FALSE` depending on whether its argument is of type factor or not. Correspondingly, `is.ordered` returns `TRUE` when its argument is ordered and `FALSE` otherwise.

`as.factor` coerces its argument to a factor. It is an abbreviated form of `factor`.

`as.ordered(x)` returns `x` if this is ordered, and `ordered(x)` otherwise.

### Warning

The interpretation of a factor depends on both the codes and the "`levels`" attribute. Be careful only to compare factors with the same set of levels (in the same order). In particular, `as.numeric` applied to a factor is meaningless, and may happen by implicit coercion. To “revert” a factor `f` to its original numeric values, `as.numeric(levels(f))[f]` is recommended and slightly more efficient than `as.numeric(as.character(f))`.

The levels of a factor are by default sorted, but the sort order may well depend on the locale at the time of creation, and should not be assumed to be ASCII.

### Note

Storing character data as a factor is more efficient storage if there is even a small proportion of repeats. On a 32-bit machine storing a string of  $n$  bytes takes  $28 + 8\lceil(n + 1)/8\rceil$  bytes whereas storing a factor code takes 4 bytes. (On a 64-bit machine 28 is replaced by 56 or more.) Only if they were computed from the same values (rather than, say, read from a file) will identical strings share storage.

### References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

### See Also

[ [.factor](#) for subsetting of factors.

[gl](#) for construction of “balanced” factors and [C](#) for factors with specified contrasts. [levels](#) and [nlevels](#) for accessing the levels, and [unclass](#) to get integer codes.

### Examples

```
(ff <- factor(substring("statistics", 1:10, 1:10), levels=letters))
as.integer(ff) # the internal codes
factor(ff)     # drops the levels that do not occur
ff[, drop=TRUE] # the same, more transparently

factor(letters[1:20], label="letter")

class(ordered(4:1))# "ordered", inheriting from "factor"

## suppose you want "NA" as a level, and to allowing missing values.
(x <- factor(c(1, 2, "NA"), exclude = ""))
is.na(x)[2] <- TRUE
x # [1] 1 <NA> NA, <NA> used because NA is a level.
is.na(x)
# [1] FALSE TRUE FALSE
```

---

file.access

*Ascertain File Accessibility*

---

### Description

Utility function to access information about files on the user’s file systems.

### Usage

```
file.access(names, mode = 0)
```

## Arguments

names	character vector containing file names.
mode	integer specifying access mode required.

## Details

The mode value can be the exclusive or of the following values

- 0** test for existence.
- 1** test for execute permission.
- 2** test for write permission.
- 4** test for read permission.

Permission will be computed for real user ID and real group ID (rather than the effective IDs).

Please note that it is not good to use this function to test before trying to open a file. On a multi-tasking system, it is possible that the accessibility of a file will change between the time you call `file.access()` and the time you try to open the file, and in recent Windows versions the underlying function in `msvcrt.dll` sometimes returns inaccurate values. It is better to wrap file open attempts in `try` instead.

## Value

An integer vector with values 0 for success and -1 for failure.

## Note

This is intended as a replacement for the S-PLUS function `access`, a wrapper for the C function of the same name, which explains the return value encoding. Note that the return value is **false** for **success**.

## See Also

[file.info](#), [try](#)

## Examples

```
fa <- file.access(dir("."))
table(fa) # count successes & failures
```

---

file.choose	<i>Choose a File Interactively</i>
-------------	------------------------------------

---

**Description**

Choose a file interactively.

**Usage**

```
file.choose(new = FALSE)
```

**Arguments**

new	Logical: choose the style of dialog box presented to the user: at present only new = FALSE is used.
-----	---

**Value**

A character vector of length one giving the file path.

**See Also**

[list.files](#) for non-interactive selection.

---

file.info	<i>Extract File Information</i>
-----------	---------------------------------

---

**Description**

Utility function to extract information about files on the user's file systems.

**Usage**

```
file.info(...)
```

**Arguments**

...	character vectors containing file names.
-----	--

**Details**

What is meant by “file access” and hence the last access time is system-dependent.

On most systems symbolic links are followed, so information is given about the file to which the link points rather than about the link.

**Value**

A data frame with row names the file names and columns

<code>size</code>	double: File size in bytes.
<code>isdir</code>	logical: Is the file a directory?
<code>mode</code>	integer of class "octmode". The file permissions, printed in octal, for example 644.
<code>mtime</code> , <code>ctime</code> , <code>atime</code>	integer of class "POSIXct": file modification, creation and last access times.
<code>uid</code>	integer: the user ID of the file's owner.
<code>gid</code>	integer: the group ID of the file's group.
<code>uname</code>	character: <code>uid</code> interpreted as a user name.
<code>grname</code>	character: <code>gid</code> interpreted as a group name.

Unknown user and group names will be NA.

Entries for non-existent or non-readable files will be NA. The `uid`, `gid`, `uname` and `grname` columns may not be supplied on a non-POSIX Unix system.

**Note**

This function will only be operational on systems with the `stat` system call, but that seems very widely available.

Some (broken) systems allow files of more than 2Gb to be created but not accessed by the `stat` system call. Such files will show up as non-readable (and very likely not be readable by any of R's input functions).

**See Also**

[files](#), [file.access](#), [list.files](#), and [DateTimeClasses](#) for the date formats.

**Examples**

```
ncol(finf <- file.info(dir()))# at least six
## Not run: finf # the whole list
## Those that are more than 100 days old :
finf[difftime(Sys.time(), finf[,"mtime"], units="days") > 100 , 1:4]

file.info("no-such-file-exists")
```

---

file.path	<i>Construct Path to File</i>
-----------	-------------------------------

---

**Description**

Construct the path to a file from components in a platform-independent way.

**Usage**

```
file.path(..., fsep = .Platform$file.sep)
```

**Arguments**

...	character vectors.
fsep	the path separator to use.

**Value**

A character vector of the arguments concatenated term-by-term and separated by fsep if all arguments have positive length; otherwise, an empty character vector.

---

file.show	<i>Display One or More Files</i>
-----------	----------------------------------

---

**Description**

Display one or more files.

**Usage**

```
file.show(..., header = rep("", nfiles), title = "R Information",
          delete.file=FALSE, pager=getOption("pager"))
```

**Arguments**

...	one or more character vectors containing the names of the files to be displayed.
header	character vector (of the same length as the number of files specified in ...) giving a header for each file being displayed. Defaults to empty strings.
title	an overall title for the display. If a single separate window is used for the display, title will be used as the window title. If multiple windows are used, their titles should combine the title and the file-specific header.
delete.file	should the files be deleted after display? Used for temporary files.
pager	the pager to be used.

**Details**

This function provides the core of the R help system, but it can be used for other purposes as well.

**Note**

How the pager is implemented is highly system dependent.

The basic Unix version concatenates the files (using the headers) to a temporary file, and displays it in the pager selected by the `pager` argument, which is a character vector specifying a system command to run on the set of files.

Most GUI systems will use a separate pager window for each file, and let the user leave it up while R continues running. The selection of such pagers could either be done using “magic” pager names being intercepted by lower-level code (such as "internal" and "console" on Windows), or by letting `pager` be an R function which will be called with the same arguments as `file.show` and take care of interfacing to the GUI.

Not all implementations will honour `delete.file`.

**Author(s)**

Ross Ihaka, Brian Ripley.

**See Also**

[files](#), [list.files](#), [help](#).

**Examples**

```
file.show(file.path(R.home(), "COPYRIGHTS"))
```

---

files

*File Manipulation*

---

**Description**

These functions provide a low-level interface to the computer's file system.

**Usage**

```
file.create(...)  
file.exists(...)  
file.remove(...)  
file.rename(from, to)  
file.append(file1, file2)  
file.copy(from, to, overwrite = FALSE)  
file.symlink(from, to)  
dir.create(path, showWarnings = TRUE)
```

**Arguments**

`...`, `file1`, `file2`, `from`, `to`  
character vectors, containing file names.

`path`  
a character vector containing a single path name.

`overwrite`  
logical; should the destination files be overwritten?

`showWarnings`  
logical; should the warnings on failure be shown?

**Details**

The `...` arguments are concatenated to form one character string: you can specify the files separately or as one vector. All of these functions expand path names: see [path.expand](#).

`file.create` creates files with the given names if they do not already exist and truncates them if they do.

`file.exists` returns a logical vector indicating whether the files named by its argument exist.

`file.remove` attempts to remove the files named in its argument.

`file.rename` attempts to rename a single file.

`file.append` attempts to append the files named by its second argument to those named by its first. The R subscript recycling rule is used to align names given in vectors of different lengths.

`file.copy` works in a similar way to `file.append` but with the arguments in the natural order for copying. Copying to existing destination files is skipped unless `overwrite = TRUE`. The `to` argument can specify a single existing directory.

`file.symlink` makes symbolic links on those Unix-like platforms which support them. The `to` argument can specify a single existing directory.

`dir.create` creates the last element of the path.

**Value**

`dir.create` and `file.rename` return a logical, true for success.

The remaining functions return a logical vector indicating which operation succeeded for each of the files attempted.

`dir.create` will return failure if the directory already exists.

**Author(s)**

Ross Ihaka, Brian Ripley

**See Also**

[file.info](#), [file.access](#), [file.path](#), [file.show](#), [list.files](#), [unlink](#), [basename](#), [path.expand](#).

**Examples**

```

cat("file A\n", file="A")
cat("file B\n", file="B")
file.append("A", "B")
file.create("A")
file.append("A", rep("B", 10))
if(interactive()) file.show("A")
file.copy("A", "C")
dir.create("tmp")
file.copy(c("A", "B"), "tmp")
list.files("tmp")
setwd("tmp")
file.remove("B")
file.symlink(file.path("../", c("A", "B")), ".")
setwd("../")
unlink("tmp", recursive=TRUE)
file.remove("A", "B", "C")

```

---

findInterval	<i>Find Interval Numbers or Indices</i>
--------------	---

---

**Description**

Find the indices of  $x$  in  $vec$ , where  $vec$  must be sorted (non-decreasingly); i.e., if  $i < findInterval(x, v)$ , we have  $v_{i_j} \leq x_j < v_{i_j+1}$  where  $v_0 := -\infty$ ,  $v_{N+1} := +\infty$ , and  $N <- length(vec)$ . At the two boundaries, the returned index may differ by 1, depending on the optional arguments `rightmost.closed` and `all.inside`.

**Usage**

```
findInterval(x, vec, rightmost.closed = FALSE, all.inside = FALSE)
```

**Arguments**

<code>x</code>	numeric.
<code>vec</code>	numeric, sorted (weakly) increasingly, of length $N$ , say.
<code>rightmost.closed</code>	logical; if true, the rightmost interval, $vec[N-1] \dots vec[N]$ is treated as <i>closed</i> , see below.
<code>all.inside</code>	logical; if true, the returned indices are coerced into $\{1, \dots, N-1\}$ , i.e., 0 is mapped to 1 and $N$ to $N-1$ .

**Details**

The function `findInterval` finds the index of one vector  $x$  in another,  $vec$ , where the latter must be non-decreasing. Where this is trivial, equivalent to `apply( outer(x, vec, ">="), 1, sum)`, as a matter of fact, the internal algorithm uses interval search ensuring

$O(n \log N)$  complexity where  $n \leftarrow \text{length}(x)$  (and  $N \leftarrow \text{length}(vec)$ ). For (almost) sorted  $x$ , it will be even faster, basically  $O(n)$ .

This is the same computation as for the empirical distribution function, and indeed, `findInterval(t, sort(X))` is *identical* to  $nF_n(t; X_1, \dots, X_n)$  where  $F_n$  is the empirical distribution function of  $X_1, \dots, X_n$ .

When `rightmost.closed = TRUE`, the result for `x[j] = vec[N]` ( $= \max(vec)$ ), is  $N - 1$  as for all other values in the last interval.

### Value

vector of length `length(x)` with values in  $0:N$  (and NA) where  $N \leftarrow \text{length}(vec)$ , or values coerced to  $1:(N-1)$  iff `all.inside = TRUE` (equivalently coercing all  $x$  values *inside* the intervals). Note that NAs are propagated from  $x$ , and Inf values are allowed in both  $x$  and  $vec$ .

### Author(s)

Martin Maechler

### See Also

`approx(*, method = "constant")` which is a generalization of `findInterval()`, `ecdf` for computing the empirical distribution function which is (up to a factor of  $n$ ) also basically the same as `findInterval()`.

### Examples

```
N <- 100
X <- sort(round(rt(N, df=2), 2))
tt <- c(-100, seq(-2, 2, len=201), +100)
it <- findInterval(tt, X)
tt[it < 1 | it >= N] # only first and last are outside range(X)
```

---

force

*Force evaluation of an Argument*

---

### Description

Forces the evaluation of a function argument.

### Usage

```
force(x)
```

### Arguments

`x` a formal argument.

**Details**

`force` forces the evaluation of a formal argument. This can be useful if the argument will be captured in a closure by the lexical scoping rules and will later be altered by an explicit assignment or an implicit assignment in a loop or an apply function.

**Note**

`force` does not force the evaluation of [promises](#).

**Examples**

```
f <- function(y) function() y
lf <- vector("list", 5)
for (i in seq(along = lf)) lf[[i]] <- f(i)
lf[[1]]() # returns 5

g <- function(y) { force(y); function() y }
lg <- vector("list", 5)
for (i in seq(along = lg)) lg[[i]] <- g(i)
lg[[1]]() # returns 1
```

---

 Foreign

*Foreign Function Interface*


---

**Description**

Functions to make calls to compiled code that has been loaded into R.

**Usage**

```
.C(name, ..., NAOK = FALSE, DUP = TRUE, PACKAGE)
.Fortran(name, ..., NAOK = FALSE, DUP = TRUE, PACKAGE)
.External(name, ..., PACKAGE)
.Call(name, ..., PACKAGE)
.External.graphics(name, ..., PACKAGE)
.Call.graphics(name, ..., PACKAGE)
```

**Arguments**

name	a character string giving the name of a C function or Fortran subroutine.
...	arguments to be passed to the foreign function.
NAOK	if TRUE then any <a href="#">NA</a> or <a href="#">NaN</a> or <a href="#">Inf</a> values in the arguments are passed on to the foreign function. If FALSE, the presence of NA or NaN or Inf values is regarded as an error.
DUP	if TRUE then arguments are “duplicated” before their address is passed to C or Fortran.

`PACKAGE` if supplied, confine the search for the name to the DLL given by this argument (plus the conventional extension, `.so`, `.sl`, `.dll`, ...). This is intended to add safety for packages, which can ensure by using this argument that no other package can override their external symbols. Use `PACKAGE="base"` for symbols linked in to R.

### Details

The functions `.C` and `.Fortran` can be used to make calls to C and Fortran code.

`.External` and `.External.graphics` can be used to call compiled code that uses R objects in the same way as internal R functions.

`.Call` and `.Call.graphics` can be used call compiled code which makes use of internal R objects. The arguments are passed to the C code as a sequence of R objects. It is included to provide compatibility with S version 4.

For details about how to write code to use with `.Call` and `.External`, see the chapter on “System and foreign language interfaces” in “Writing R Extensions” in the ‘doc/manual’ subdirectory of the R source tree.

### Value

The functions `.C` and `.Fortran` return a list similar to the ... list of arguments passed in, but reflecting any changes made by the C or Fortran code.

`.External`, `.Call`, `.External.graphics`, and `.Call.graphics` return an R object.

These calls are typically made in conjunction with `dyn.load` which links DLLs to R.

The `.graphics` versions of `.Call` and `.External` are used when calling code which makes low-level graphics calls. They take additional steps to ensure that the device driver display lists are updated correctly.

### Argument types

The mapping of the types of R arguments to C or Fortran arguments in `.C` or `.Fortran` is

R	C	Fortran
integer	int *	integer
numeric	double *	double precision
– or –	float *	real
complex	Rcomplex *	double complex
logical	int *	integer
character	char **	[see below]
list	SEXP *	not allowed
other	SEXP	not allowed

Numeric vectors in R will be passed as type `double *` to C (and as `double precision` to Fortran) unless (i) `.C` or `.Fortran` is used, (ii) `DUP` is false and (iii) the argument has attribute `Csingle` set to TRUE (use `as.single` or `single`). This mechanism is only intended to be used to facilitate the interfacing of existing C and Fortran code.

The C type `Rcomplex` is defined in ‘Complex.h’ as a typedef `struct {double r;`

`double i; }`. Fortran type `double complex` is an extension to the Fortran standard, and the availability of a mapping of `complex` to Fortran may be compiler dependent.

*Note:* The C types corresponding to `integer` and `logical` are `int`, not `long` as in S.

The first character string of a character vector is passed as a C character array to Fortran: that string may be usable as `character*255` if its true length is passed separately. Only up to 255 characters of the string are passed back. (How well this works, or even if it works at all, depends on the C and Fortran compilers and the platform.)

Missing (NA) string values are passed to `.C` as the string "NA". As the C `char` type can represent all possible bit patterns there appears to be no way to distinguish missing strings from the string "NA". If this distinction is important use `.Call`.

Functions, expressions, environments and other language elements are passed as the internal R pointer type `SEXP`. This type is defined in 'Rinternals.h' or the arguments can be declared as generic pointers, `void *`. Lists are passed as C arrays of `SEXP` and can be declared as `void *` or `SEXP *`. Note that you cannot assign values to the elements of the list within the C routine. Assigning values to elements of the array corresponding to the list bypasses R's memory management/garbage collection and will cause problems. Essentially, the array corresponding to the list is read-only. If you need to return S objects created within the C routine, use the `.Call` interface.

R functions can be invoked using `call_S` or `call_R` and can be passed lists or the simple types as arguments.

### Header files for external code

Writing code for use with `.External` and `.Call` will use internal R structures. If possible use just those defined in 'Rinternals.h' and/or the macros in 'Rdefines.h', as other header files are not installed and are even more likely to be changed.

### Note

*DUP=FALSE is dangerous.*

There are two dangers with using `DUP=FALSE`.

The first is that if you pass a local variable to `.C/.Fortran` with `DUP=FALSE`, your compiled code can alter the local variable and not just the copy in the return list. Worse, if you pass a local variable that is a formal parameter of the calling function, you may be able to change not only the local variable but the variable one level up. This will be very hard to trace.

The second is that lists are passed as a single R `SEXP` with `DUP=FALSE`, not as an array of `SEXP`. This means the accessor macros in 'Rinternals.h' are needed to get at the list elements and the lists cannot be passed to `call_S/call_R`. New code using R objects should be written using `.Call` or `.External`, so this is now only a minor issue.

(Prior to R version 1.2.0 there has a third danger, that objects could be moved in memory by the garbage collector. The current garbage collector never moves objects.)

It is safe and useful to set `DUP=FALSE` if you do not change any of the variables that might be affected, e.g.,

```
.C("Cfunction", input=x, output=numeric(10)).
```

In this case the output variable did not exist before the call so it cannot cause trouble. If the input variable is not changed in the C code of `Cfunction` you are safe.

Neither `.Call` nor `.External` copy their arguments. You should treat arguments you receive through these interfaces as read-only.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`.C` and `.Fortran`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (`.Call`.)

## See Also

[dyn.load](#).

---

formals

*Access to and Manipulation of the Formal Arguments*

---

## Description

Get or set the formal arguments of a function.

## Usage

```
formals(fun = sys.function(sys.parent()))
formals(fun, envir = parent.frame()) <- value
```

## Arguments

<code>fun</code>	a function object, or see Details.
<code>envir</code>	environment in which the function should be defined.
<code>value</code>	a list of R expressions.

## Details

For the first form, `fun` can be a character string naming the function to be manipulated, which is searched for from the parent environment. If it is not specified, the function calling `formals` is used.

## Value

`formals` returns the formal argument list of the function specified.

The assignment form sets the formals of a function to the list on the right hand side.

## See Also

[args](#) for a “human-readable” version, [alist](#), [body](#), [function](#).

**Examples**

```
length(formals(lm))      # the number of formal arguments
names(formals(boxplot)) # formal arguments names

f <- function(x)a+b
formals(f) <- alist(a=,b=3) # function(a,b=3)a+b
f(2) # result = 5
```

---

format

*Encode in a Common Format*


---

**Description**

Format an R object for pretty printing: `format.pval` is intended for formatting p-values.

**Usage**

```
format(x, ...)

## S3 method for class 'AsIs':
format(x, width = 12, ...)

## S3 method for class 'data.frame':
format(x, ..., justify = "none")

## Default S3 method:
format(x, trim = FALSE, digits = NULL,
       nsmall = 0, justify = c("left", "right", "none"),
       big.mark = "", big.interval = 3,
       small.mark = "", small.interval = 5,
       decimal.mark = ".", ...)

## S3 method for class 'factor':
format(x, ...)

format.pval(pv, digits = max(1, getOption("digits") - 2),
            eps = .Machine$double.eps, na.form = "NA")

prettyNum(x, big.mark = "", big.interval = 3,
          small.mark = "", small.interval = 5,
          decimal.mark = ".", ...)
```

**Arguments**

`x` any R object (conceptually); typically numeric.  
`trim` logical; if TRUE, leading blanks are trimmed off the strings.

<code>digits</code>	how many significant digits are to be used for <code>numeric</code> <code>x</code> . The default, <code>NULL</code> , uses <code>options()\$digits</code> . This is a suggestion: enough decimal places will be used so that the smallest (in magnitude) number has this many significant digits.
<code>nsmall</code>	number of digits which will always appear to the right of the decimal point in formatting real/complex numbers in non-scientific formats. Allowed values <code>0 &lt;= nsmall &lt;= 20</code> .
<code>justify</code>	should character vector be left-justified, right-justified or left alone. When justifying, the field width is that of the longest string.
<code>big.mark</code>	character; if not empty used as mark between every <code>big.interval</code> decimals <i>before</i> (hence <code>big</code> ) the decimal point.
<code>big.interval</code>	see <code>big.mark</code> above; defaults to 3.
<code>small.mark</code>	character; if not empty used as mark between every <code>small.interval</code> decimals <i>after</i> (hence <code>small</code> ) the decimal point.
<code>small.interval</code>	see <code>small.mark</code> above; defaults to 5.
<code>decimal.mark</code>	the character used to indicate the numeric decimal point.
<code>pv</code>	a numeric vector.
<code>eps</code>	a numerical tolerance: see <code>Details</code> .
<code>na.form</code>	character representation of NAs.
<code>width</code>	the returned vector has elements of at most <code>width</code> .
<code>...</code>	further arguments passed to or from other methods.

## Details

These functions convert their first argument to a vector (or array) of character strings which have a common format (as is done by `print`), fulfilling `length(format*(x, *)) == length(x)`. The trimming with `trim = TRUE` is useful when the strings are to be used for plot `axis` annotation.

`format.AsIs` deals with columns of complicated objects that have been extracted from a data frame.

`format.pval` is mainly an auxiliary function for `print.summary.lm` etc., and does separate formatting for fixed, floating point and very small values; those less than `eps` are formatted as "`<[eps]`" (where "[eps]" stands for `format(eps, digits)`).

The function `formatC` provides a rather more flexible formatting facility for numbers, but does *not* provide a common format for several numbers, nor it is platform-independent.

`format.data.frame` formats the data frame column by column, applying the appropriate method of `format` for each column.

`prettyNum` is the utility function for prettifying `x`. If `x` is not a character, `format(x[i], ...)` is applied to each element, and then it is left unchanged if all the other arguments are at their defaults. Note that `prettyNum(x)` may behave unexpectedly if `x` is a character not resulting from something like `format(<number>)`.

**Note**

Currently `format` drops trailing zeroes, so `format(6.001, digits=2)` gives "6" and `format(c(6.0, 13.1), digits=2)` gives `c(" 6", "13")`.

Character(s) " in input strings `x` are escaped to `\`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`format.info` indicates how something would be formatted; `formatC`, `paste`, `as.character`, `sprintf`.

**Examples**

```
format(1:10)

zz <- data.frame("(row names)"= c("aaaaa", "b"), check.names=FALSE)
format(zz)
format(zz, justify="left")

## use of nsmall
format(13.7)
format(13.7, nsmall=3)

r <- c("76491283764.97430", "29.12345678901", "-7.1234", "-100.1", "1123")
## American:
prettyNum(r, big.mark = ",")
## Some Europeans:
prettyNum(r, big.mark = "'", decimal.mark = ",")

(dd <- sapply(1:10, function(i)paste((9:0)[1:i],collapse="")))
prettyNum(dd, big.mark="'")

pN <- stats::pnorm(1:7, lower=FALSE)
cbind(format(pN, small.mark = " ", digits = 15))
cbind(formatC(pN, small.mark = " ", digits = 17, format = "f"))
```

---

format.Date

*Date Conversion Functions to and from Character*


---

**Description**

Functions to convert between character representations and objects of class "Date" representing calendar dates.

## Usage

```
as.Date(x, ...)
## S3 method for class 'character':
as.Date(x, format = "", ...)

## S3 method for class 'Date':
format(x, ...)

## S3 method for class 'Date':
as.character(x, ...)
```

## Arguments

<code>x</code>	An object to be converted.
<code>format</code>	A character string. The default is "%Y-%m-%d". For details see <a href="#">strftime</a> .
<code>...</code>	Further arguments to be passed from or to other methods, including <code>format</code> for <code>as.character</code> and <code>as.Date</code> methods.

## Details

The usual vector re-cycling rules are applied to `x` and `format` so the answer will be of length that of the longer of the vectors.

Locale-specific conversions to and from character strings are used where appropriate and available. This affects the names of the days and months.

The `as.Date` methods accept character strings, factors, logical NA and objects of classes "[POSIXlt](#)" and "[POSIXct](#)". (The last are converted to days by ignoring the time after midnight in the representation of the time in UTC.) Also objects of class "date" (from package [date](#) or [survival](#)) and "dates" (from package [chron](#)).

The `format` and `as.character` methods ignore any fractional part of the date.

## Value

The `format` and `as.character` methods return a character vector representing the date.

The `as.Date` methods return an object of class "[Date](#)".

## Note

The default formats follow the rules of the ISO 8601 international standard which expresses a day as "2001-02-03".

If the date string does not specify the date completely, the returned answer may be system-specific. The most common behaviour is to assume that a missing year, month or day is the current one. If it specifies a date incorrectly, reliable implementations will give an error and the date is reported as NA. Unfortunately some common implementations (such as 'glibc') are unreliable and guess at the intended meaning.

Years before 1CE (aka 1AD) will probably not be handled correctly.

## References

International Organization for Standardization (1988, 1997, ...) *ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times*. The 1997 version is available on-line at <ftp://ftp.qsl.net/pub/g1smd/8601v03.pdf>

## See Also

[Date](#) for details of the date class; [locales](#) to query or set a locale.

Your system's help pages on `strftime` and `strptime` to see how to specify their formats.

## Examples

```
## locale-specific version of the date
format(Sys.Date(), "%a %b %d")

## read in date info in format 'ddmmyyyy'
## This will give NA(s) in some locales; setting the C locale
## as in the commented lines will overcome this on most systems.
## lct <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- as.Date(x, "%d%b%Y")
## Sys.setlocale("LC_TIME", lct)
z

## read in date/time info in format 'm/d/y'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
as.Date(dates, "%m/%d/%y")
```

---

format.info

*format(.) Information*

---

## Description

Information is returned on how `format(x, digits = options("digits"))` would be formatted.

## Usage

```
format.info(x, nsmall = 0)
```

## Arguments

`x` (numeric) vector; potential argument of `format(x, ...)`.  
`nsmall` (see `format(*, nsmall)`).

**Value**

An *integer vector* of length 3, say `r`.

<code>r[1]</code>	width (number of characters) used for <code>format(x)</code>
<code>r[2]</code>	number of digits after decimal point.
<code>r[3]</code>	in <code>0:2</code> ; if $\geq 1$ , <i>exponential</i> representation would be used, with exponent length of <code>r[3]+1</code> .

**Note**

The result **depends** on the value of `options("digits")`.

**See Also**

`format`, `formatC`.

**Examples**

```
dd <- options("digits") ; options(digits = 7) #-- for the following
format.info(123) # 3 0 0
format.info(pi) # 8 6 0
format.info(1e8) # 5 0 1 - exponential "1e+08"
format.info(1e222)#6 0 2 - exponential "1e+222"

x <- pi*10^c(-10,-2,0:2,8,20)
names(x) <- formatC(x,w=1,dig=3,format="g")
cbind(sapply(x,format))
t(sapply(x, format.info))

## using at least 8 digits right of "."
t(sapply(x, format.info, nsmall = 8))

# Reset old options:
options(dd)
```

**Description**

Formatting numbers individually and flexibly, using C style format specifications. `format.char` is a helper function for `formatC`.

**Usage**

```
formatC(x, digits = NULL, width = NULL,
        format = NULL, flag = "", mode = NULL,
        big.mark = "", big.interval = 3,
        small.mark = "", small.interval = 5,
        decimal.mark = ".")

format.char(x, width = NULL, flag = "-")
```

**Arguments**

**x** an atomic numerical or character object, typically a vector of real numbers.

**digits** the desired number of digits after the decimal point (`format = "f"`) or *significant* digits (`format = "g", "e" or "fg"`).  
Default: 2 for integer, 4 for real numbers. If less than 0, the C default of 6 digits is used.

**width** the total field width; if both `digits` and `width` are unspecified, `width` defaults to 1, otherwise to `digits + 1`. `width = 0` will use `width = digits`, `width < 0` means left justify the number in this field (equivalent to `flag = "-"`). If necessary, the result will have more characters than `width`.

**format** equal to "d" (for integers), "f", "e", "E", "g", "G", "fg" (for reals), or "s" (for strings). Default is "d" for integers, "g" for reals.  
"f" gives numbers in the usual xxx.xxx format; "e" and "E" give n.ddde+nn or n.dddE+nn (scientific format); "g" and "G" put `x[i]` into scientific format only if it saves space to do so.  
"fg" uses fixed format as "f", but `digits` as the minimum number of *significant* digits. That this can lead to quite long result strings, see examples below. Note that unlike `signif` this prints large numbers with more significant digits than `digits`.

**flag** format modifier as in Kernighan and Ritchie (1988, page 243). "0" pads leading zeros; "-" does left adjustment, others are "+", " ", and "#".

**mode** "double" (or "real"), "integer" or "character". Default: Determined from the storage mode of `x`.

**big.mark**, **big.interval**, **small.mark**, **small.interval**, **decimal.mark** used for prettying longer decimal sequences, passed to `prettyNum`: that help page explains the details.

**Details**

If you set `format` it over-rides the setting of `mode`, so `formatC(123.45, mode="double", format="d")` gives 123.

The rendering of scientific format is platform-dependent: some systems use `n.ddde+nnn` or `n.dddenn` rather than `n.ddde+nn`.

`formatC` does not necessarily align the numbers on the decimal point, so `formatC(c(6.11, 13.1), digits=2, format="fg")` gives `c("6.1", " 13")`. If you want common formatting for several numbers, use `format`.

**Value**

A character object of same size and attributes as `x`. Unlike `format`, each number is formatted individually. Looping over each element of `x`, `sprintf(...)` is called (inside the C function `str_signif`).

`format.char(x)` and `formatC`, for character `x`, do simple (left or right) padding with white space.

**Author(s)**

Originally written by Bill Dunlap, later much improved by Martin Maechler, it was first adapted for R by Friedrich Leisch.

**References**

Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language*. Second edition. Prentice Hall.

**See Also**

`format`, `sprintf` for more general C like formatting.

**Examples**

```
xx <- pi * 10^(-5:4)
cbind(format(xx, digits=4), formatC(xx))
cbind(formatC(xx, wid = 9, flag = "-"))
cbind(formatC(xx, dig = 5, wid = 8, format = "f", flag = "0"))
cbind(format(xx, digits=4), formatC(xx, dig = 4, format = "fg"))

format.char(c("a", "Abc", "no way"), wid = -7) # <=> flag = "-"
formatC(c("a", "Abc", "no way"), wid = -7) # <=> flag = "-"
formatC(c((-1:1)/0, c(1,100)*pi), wid=8, dig=1)

xx <- c(1e-12, -3.98765e-10, 1.45645e-69, 1e-70, pi*1e37, 3.44e4)
##      1      2      3      4      5      6
formatC(xx)
formatC(xx, format="fg") # special "fixed" format.
formatC(xx, format="f", dig=80)#>> also long strings
```

**Description**

Format vectors of items and their descriptions as 2-column tables or LaTeX-style description lists.

**Usage**

```
formatDL(x, y, style = c("table", "list"),
         width = 0.9 * getOption("width"), indent = NULL)
```

**Arguments**

<code>x</code>	a vector giving the items to be described, or a list of length 2 or a matrix with 2 columns giving both items and descriptions.
<code>y</code>	a vector of the same length as <code>x</code> with the corresponding descriptions. Only used if <code>x</code> does not already give the descriptions.
<code>style</code>	a character string specifying the rendering style of the description information. If "table", a two-column table with items and descriptions as columns is produced (similar to Texinfo's @table environment. If "list", a LaTeX-style tagged description list is obtained.
<code>width</code>	a positive integer giving the target column for wrapping lines in the output.
<code>indent</code>	a positive integer specifying the indentation of the second column in table style, and the indentation of continuation lines in list style. Must not be greater than <code>width/2</code> , and defaults to <code>width/3</code> for table style and <code>width/9</code> for list style.

**Details**

After extracting the vectors of items and corresponding descriptions from the arguments, both are coerced to character vectors.

In table style, items with more than `indent - 3` characters are displayed on a line of their own.

**Value**

a character vector with the formatted entries.

**Examples**

```
## Use R to create the 'INDEX' for package 'splines' from its 'CONTENTS'
x <- read.dcf(file = system.file("CONTENTS", package = "splines"),
             fields = c("Entry", "Description"))
x <- as.data.frame(x)
writeLines(formatDL(x$Entry, x$Description))
## or equivalently: writeLines(formatDL(x))
## Same information in tagged description list style:
writeLines(formatDL(x$Entry, x$Description, style = "list"))
## or equivalently: writeLines(formatDL(x, style = "list"))
```

---

function

*Function Definition*

---

## Description

These functions provide the base mechanisms for defining new functions in the R language.

## Usage

```
function( arglist ) expr
return(value)
```

## Arguments

<code>arglist</code>	Empty or one or more name or name=expression terms.
<code>value</code>	An expression.

## Details

In R (unlike S) the names in an argument list cannot be quoted non-standard names.

If `value` is missing, `NULL` is returned. If it is a single expression, the value of the evaluated expression is returned.

If the end of a function is reached without calling `return`, the value of the last evaluated expression is returned.

## Warning

Prior to R 1.8.0, `value` could be a series of non-empty expressions separated by commas. In that case the value returned is a list of the evaluated expressions, with names set to the expressions where these are the names of R objects. That is, `a=foo()` names the list component `a` and gives it value the result of evaluating `foo()`.

This has been deprecated (and a warning is given), as it was never documented in S, and whether or not the list is named differs by S versions.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[args](#) and [body](#) for accessing the arguments and body of a function.

[debug](#) for debugging; [invisible](#) for `return(. )`ing *invisibly*.

**Examples**

```
norm <- function(x) sqrt(x%*%x)
norm(1:4)

## An anonymous function:
(function(x,y){ z <- x^2 + y^2; x+y+z })(0:7, 1)
```

gc

*Garbage Collection***Description**

A call of `gc` causes a garbage collection to take place. `gcinfo` sets a flag so that automatic collection is either silent (`verbose=FALSE`) or prints memory usage statistics (`verbose=TRUE`).

**Usage**

```
gc(verbose = getOption("verbose"))
gcinfo(verbose)
```

**Arguments**

`verbose` logical; if `TRUE`, the garbage collection prints statistics about cons cells and the vector heap.

**Details**

A call of `gc` causes a garbage collection to take place. This takes place automatically without user intervention, and the primary purpose of calling `gc` is for the report on memory usage.

However, it can be useful to call `gc` after a large object has been removed, as this may prompt R to return memory to the operating system.

**Value**

`gc` returns a matrix with rows `"Ncells"` (*cons cells*, usually 28 bytes each on 32-bit systems and 56 bytes on 64-bit systems, and `"Vcells"` (*vector cells*, 8 bytes each), and columns `"used"` and `"gc trigger"`, each also interpreted in megabytes (rounded up to the next 0.1Mb).

If maxima have been set for either `"Ncells"` or `"Vcells"`, a fifth column is printed giving the current limits in Mb (with NA denoting no limit).

`gcinfo` returns the previous value of the flag.

**See Also**

[Memory](#) on R's memory management, and [gctorture](#) if you are an R hacker. [reg.finalizer](#) for actions to happen upon garbage collection.

**Examples**

```
gc() #- do it now
gcinfo(TRUE) #-- in the future, show when R does it
x <- integer(100000); for(i in 1:18) x <- c(x,i)
gcinfo(verbose = FALSE)#-- don't show it anymore

gc(TRUE)
```

---

`gc.time`*Report Time Spent in Garbage Collection*

---

**Description**

This function reports the time spent in garbage collection so far in the R session while GC timing was enabled..

**Usage**

```
gc.time(on = TRUE)
```

**Arguments**

`on` logical; if TRUE, GC timing is enabled.

**Value**

A numerical vector of length 5 giving the user CPU time, the system CPU time, the elapsed time and children's user and system CPU times (normally both zero).

**Warnings**

This is experimental functionality, likely to be removed as soon as the next release.

The timings are rounded up by the sampling interval for timing processes, and so are likely to be over-estimates.

**See Also**

[gc](#), [proc.time](#) for the timings for the session.

**Examples**

```
gc.time()
```

---

gctorture	<i>Torture Garbage Collector</i>
-----------	----------------------------------

---

**Description**

Provokes garbage collection on (nearly) every memory allocation. Intended to ferret out memory protection bugs. Also makes R run *very* slowly, unfortunately.

**Usage**

```
gctorture(on = TRUE)
```

**Arguments**

`on`                    logical; turning it on/off.

**Value**

Previous value.

**Author(s)**

Peter Dalgaard

---

get	<i>Return the Value of a Named Object</i>
-----	---

---

**Description**

Search for an R object with a given name and return it.

**Usage**

```
get(x, pos=-1, envir=as.environment(pos), mode="any", inherits=TRUE)
mget(x, envir, mode = "any",
      ifnotfound = list(function(x) stop(paste("value for \"",
      x, "\" not found", sep = "")), call. = FALSE)), inherits = FALSE)
```

**Arguments**

<code>x</code>	a variable name (given as a character string).
<code>pos</code>	where to look for the object (see the details section); if omitted, the function will search as if the name of the object appeared unquoted in an expression.
<code>envir</code>	an alternative way to specify an environment to look in; see the details section.
<code>mode</code>	the mode of object sought.
<code>inherits</code>	should the enclosing frames of the environment be searched?
<code>ifnotfound</code>	A list of values to be used if the item is not found.

## Details

The `pos` argument can specify the environment in which to look for the object in any of several ways: as an integer (the position in the `search` list); as the character string name of an element in the search list; or as an `environment` (including using `sys.frame` to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

This function looks to see if the name `x` has a value bound to it in the specified environment. If `inherits` is `TRUE` and a value is not found for `x` in the specified environment, the enclosing frames of the environment (and the search path) are searched until the name `x` is encountered.

**Warning:** `inherits=TRUE` is the default behaviour for R but not for S.

The mode may specify collections such as `"numeric"` and `"function"`: any member of the collection will suffice.

Using a `NULL` environment is equivalent to using the current environment.

For `mget` multiple values are returned in a named `list`. This is true even if only one value is requested. The value in `mode` and `ifnotfound` can be either the same length as the number of requested items or of length 1. The argument `ifnotfound` must be a list containing either the value to use if the requested item is not found or a function of one argument which will be called if the item is not found. The argument is the name of the item being requested. The default value for `inherits` is `FALSE`, in contrast to the default behavior for `get`.

## Value

The object found. (If no object is found an error results.)

## Note

The reverse of `a <- get(nam)` is `assign(nam, a)`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

`exists`, `assign`.

## Examples

```
get("%o%")

##test mget
e1 <- new.env()
mget(letters, e1, ifnotfound=LETTERS)
```

---

`getCallingDLL`*Compute DLL for native interface call*

---

### Description

This is an internal function that is called from R's C code to determine the enclosing namespace of a `.C/.Call/.Fortran/.External` call which has no `PACKAGE` argument. If the call has been made from a function within a namespace, then we can find the DLL associated with that namespace. The purpose of this is to avoid having to use the `PACKAGE` argument in these native calls and so better support versions of packages.

This is an internal function that may be migrated to internal C code in the future and so should not be used by R programmers.

### Usage

```
getCallingDLL(f = sys.function(1), doStop = FALSE)
```

### Arguments

<code>f</code>	the function whose namespace and DLL are to be found. By default, this is the current function being called which is the one in which the native routine is being invoked.
<code>doStop</code>	a logical value indicating whether failure to find a namespace and/or DLL is an error ( <code>TRUE</code> ) or not ( <code>FALSE</code> ). The default is <code>FALSE</code> so that when this is called because there is no <code>PACKAGE</code> argument in a <code>.C</code> , <code>.Call</code> , <code>.Fortran</code> , <code>.External</code> call, no error occurs and the regular lookup is performed by searching all DLLs in order.

### See Also

`.C`, `.Call`, `.Fortran`, `.External`

### Examples

```
if(exists("ansari.test"))
  getCallingDLL(ansari.test)
```

---

`getDLLRegisteredRoutines`*Reflectance Information for C/Fortran routines in a DLL*

---

## Description

This function allows us to query the set of routines in a DLL that are registered with R to enhance dynamic lookup, error handling when calling native routines, and potentially security in the future. This function provides a description of each of the registered routines in the DLL for the different interfaces, i.e. `.C`, `.Call`, `.Fortran` and `.External`.

## Usage

```
getDLLRegisteredRoutines(dll)
```

## Arguments

`dll` a `DLLInfo` object. This can be retrieved from the object returned from `getLoadedDLLs`. Essentially, it provides information about the name and location of the DLL, whether R uses dynamic lookup when resolving symbols in the DLL and also references to the R data structures that allow us to resolve symbols (i.e. `DllInfo` and `HINSTANCE`).

## Details

This takes the registration information after it has been registered and processed by the R internals. In other words, it uses the extended information

## Value

A list with four elements corresponding to the routines registered for the `.C`, `.Call`, `.Fortran` and `.External` interfaces. Each element is a list with as many elements as there were routines registered for that interface. Each element identifies a routine and is an object of class `NativeSymbolInfo`. An object of this class has the following fields:

<code>name</code>	the registered name of the routine (not necessarily the name in the C code).
<code>address</code>	the memory address of the routine as resolved in the loaded DLL. This may be <code>NULL</code> if the symbol has not yet been resolved.
<code>dll</code>	an object of class <code>DLLInfo</code> describing the DLL. This is same for all elements returned from <code>getDLLRegisteredRoutines</code> .
<code>numParameters</code>	the number of arguments the native routine is to be called with. In the future, we will provide information about the types of the parameters also.

## Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

**References**

"Writing R Extensions Manual" for symbol registration. R News, Volume 1/3, September 2001. "In search of C/C++ & Fortran Symbols"

**See Also**

[getLoadedDLLs](#)

**Examples**

```
dlls <- getLoadedDLLs()
getDLLRegisteredRoutines(dlls[["base"]])
```

---

getLoadedDLLs

*Get DLLs Loaded in Current Session*

---

**Description**

This function provides a way to get a list of all the Dynamically Loadable Libraries (DLLs) that are currently loaded in the current R session.

**Usage**

```
getLoadedDLLs()
```

**Details**

This queries the internal table that manages the DLLs.

**Value**

A list with an element corresponding to each DLL that is currently loaded in the session. Each object is an object of class which has the following entries.

name	the abbreviated name.
path	the fully qualified name of the file which was dynamically loaded.
dynamicLookup	a logical value indicating whether R uses only the registration information to resolve symbols or whether it searches the entire symbol table of the DLL.
handle	a reference to the C-level data structure that provides access to the contents of the DLL. This is an object of class <code>DLLHandle</code> .

**Note**

We are starting to use the `handle` elements in the DLL object to resolve symbols more directly in R.

**Author(s)**

Duncan Temple Lang <duncan@wald.ucdavis.edu>.

**See Also**

[getNativeSymbolInfo](#)

**Examples**

```
getLoadedDLLs()
```

---

```
getNativeSymbolInfo
```

*Obtain a description of a native (C/Fortran) symbol*

---

**Description**

This finds and returns as comprehensive a description of a dynamically loaded or “exported” built-in native symbol. It returns information about the name of the symbol, the library in which it is located and, if available, the number of arguments it expects and by which interface it should be called (i.e. `.Call`, `.C`, `.Fortran`, or `.External`). Additionally, it returns the address of the symbol and this can be passed to other C routines which can invoke. Specifically, this provides a way to explicitly share symbols between different dynamically loaded package libraries. Also, it provides a way to query where symbols were resolved, and aids diagnosing strange behavior associated with dynamic resolution.

**Usage**

```
getNativeSymbolInfo(name, PACKAGE)
```

**Arguments**

<code>name</code>	the name of the native symbol as used in a call to <code>is.loaded</code> , etc.
<code>PACKAGE</code>	an optional argument that specifies to which dynamically loaded library we restrict the search for this symbol. If this is "base", we search in the R executable itself.

**Details**

This uses the same mechanism for resolving symbols as is used in all the native interfaces (`.Call`, etc.). If the symbol has been explicitly registered by the shared library in which it is contained, information about the number of arguments and the interface by which it should be called will be returned. Otherwise, a generic native symbol object is returned.

**Value**

If the symbol is not found, an error is raised. Otherwise, the value is a list containing the following elements:

<code>name</code>	the name of the symbol, as given by the <code>name</code> argument.
<code>address</code>	the native memory address of the symbol which can be used to invoke the routine, and also compare with other symbol address. This is an external pointer object and of class <code>NativeSymbol</code> .
<code>package</code>	a list containing 3 elements: <ul style="list-style-type: none"> <li><b>name</b> the short form of the library name which can be used as the value of the <code>PACKAGE</code> argument in the different native interface functions.</li> <li><b>path</b> the fully qualified name of the shared library file.</li> <li><b>dynamicLookup</b> a logical value indicating whether dynamic resolution is used when looking for symbols in this library, or only registered routines can be located.</li> </ul>
<code>numParameters</code>	the number of arguments that should be passed in a call to this routine.

Additionally, the list will have an additional class, being `CRoutine`, `CallRoutine`, `FortranRoutine` or `ExternalRoutine` corresponding to the R interface by which it should be invoked.

**Note**

One motivation for accessing this reflectance information is to be able to pass native routines to C routines as “function pointers” in C. This allows us to treat native routines and R functions in a similar manner, such as when passing an R function to C code that makes callbacks to that function at different points in its computation (e.g., `nls`). Additionally, we can resolve the symbol just once and avoid resolving it repeatedly or using the internal cache. In the future, one may be able to treat `NativeSymbol` objects directly as callback objects.

**Author(s)**

Duncan Temple Lang

**References**

For information about registering native routines, see “In Search of C/C++ & FORTRAN Routines”, R News, volume 1, number 3, 2001, p20–23 (<http://CRAN.R-project.org/doc/Rnews/>).

**See Also**

`is.loaded`, `.C`, `.Fortran`, `.External`, `.Call`, `dyn.load`.

**Examples**

```
library(stats) # normally loaded
getNativeSymbolInfo("dansari")

getNativeSymbolInfo(symbol.For("hcass2"))
```

---

getNumCConverters *Management of .C argument conversion list*

---

**Description**

These functions provide facilities to manage the extensible list of converters used to translate R objects to C pointers for use in `.C` calls. The number and a description of each element in the list can be retrieved. One can also query and set the activity status of individual elements, temporarily ignoring them. And one can remove individual elements.

**Usage**

```
getNumCConverters()
getCConverterDescriptions()
getCConverterStatus()
setCConverterStatus(id, status)
removeCConverter(id)
```

**Arguments**

<code>id</code>	either a number or a string identifying the element of interest in the converter list. A string is matched against the description strings for each element to identify the element. Integers are specified starting at 1 (rather than 0).
<code>status</code>	a logical value specifying whether the element is to be considered active ( <code>TRUE</code> ) or not ( <code>FALSE</code> ).

**Details**

The internal list of converters is potentially used when converting individual arguments in a `.C` call. If an argument has a non-trivial class attribute, we iterate over the list of converters looking for the first that “matches”. If we find a matching converter, we have it create the C-level pointer corresponding to the R object. When the call to the C routine is complete, we use the same converter for that argument to reverse the conversion and create an R object from the current value in the C pointer. This is done separately for all the arguments.

The functions documented here provide R user-level capabilities for investigating and managing the list of converters. There is currently no mechanism for adding an element to the converter list within the R language. This must be done in C code using the routine `R_addToCConverter()`.

**Value**

`getNumCConverters` returns an integer giving the number of elements in the list, both active and inactive.

`getCConverterDescriptions` returns a character vector containing the description string of each element of the converter list.

`getCConverterStatus` returns a logical vector with a value for each element in the converter list. Each value indicates whether that converter is active (TRUE) or inactive (FALSE). The names of the elements are the description strings returned by `getCConverterDescriptions`.

`setCConverterStatus` returns the logical value indicating the activity status of the specified element before the call to change it took effect. This is TRUE for active and FALSE for inactive.

`removeCConverter` returns TRUE if an element in the converter list was identified and removed. In the case that no such element was found, an error occurs.

**Author(s)**

Duncan Temple Lang

**References**

<http://developer.R-project.org/CObjectConversion.pdf>

**See Also**

[.C](#)

**Examples**

```
getNumCConverters()
getCConverterDescriptions()
getCConverterStatus()
## Not run:
old <- setCConverterStatus(1, FALSE)

setCConverterStatus(1, old)
## End(Not run)
## Not run:
removeCConverter(1)
removeCConverter(getCConverterDescriptions()[1])
## End(Not run)
```

---

getpid

*Get the Process ID of the R Session*

---

**Description**

Get the process ID of the R Session. It is guaranteed by the operating system that two R sessions running simultaneously will have different IDs, but it is possible that R sessions running at different times will have the same ID.

**Usage**

```
Sys.getpid()
```

**Value**

An integer, usually a small integer between 0 and 32767 under Unix-alikes and a much small integer under Windows.

**Examples**

```
Sys.getpid()
```

---

getwd	<i>Get or Set Working Directory</i>
-------	-------------------------------------

---

**Description**

`getwd` returns an absolute filename representing the current working directory of the R process; `setwd(dir)` is used to set the working directory to `dir`.

**Usage**

```
getwd()  
setwd(dir)
```

**Arguments**

`dir`            A character string.

**Value**

`getwd` returns a character vector, or `NULL` if the working directory is not available on that platform. `setwd` returns `NULL` invisibly. It will give an error if it does not succeed.

**Note**

These functions are not implemented on all platforms.

**See Also**

[list.files](#) for the *contents* of a directory.

**Examples**

```
(WD <- getwd())  
if (!is.null(WD)) setwd(WD)
```

---

**gl** *Generate Factor Levels*

---

**Description**

Generate factors by specifying the pattern of their levels.

**Usage**

```
gl(n, k, length = n*k, labels = 1:n, ordered = FALSE)
```

**Arguments**

n	an integer giving the number of levels.
k	an integer giving the number of replications.
length	an integer giving the length of the result.
labels	an optional vector of labels for the resulting factor levels.
ordered	a logical indicating whether the result should be ordered or not.

**Value**

The result has levels from 1 to n with each value replicated in groups of length k out to a total length of length.

gl is modelled on the *GLIM* function of the same name.

**See Also**

The underlying `factor()`.

**Examples**

```
## First control, then treatment:  
gl(2, 8, label = c("Control", "Treat"))  
## 20 alternating 1s and 2s  
gl(2, 1, 20)  
## alternating pairs of 1s and 2s  
gl(2, 2, 20)
```

**Description**

grep searches for matches to `pattern` (its first argument) within the character vector `x` (second argument). `regexpr` does too, but returns more detail in a different format.

`sub` and `gsub` perform replacement of matches determined by regular expression matching.

**Usage**

```
grep(pattern, x, ignore.case = FALSE, extended = TRUE, perl = FALSE,
      value = FALSE, fixed = FALSE)
sub(pattern, replacement, x,
     ignore.case = FALSE, extended = TRUE, perl = FALSE)
gsub(pattern, replacement, x,
      ignore.case = FALSE, extended = TRUE, perl = FALSE)
regexpr(pattern, text, extended = TRUE, perl = FALSE, fixed = FALSE)
```

**Arguments**

<code>pattern</code>	character string containing a <a href="#">regular expression</a> (or character string for <code>fixed = TRUE</code> ) to be matched in the given character vector.
<code>x</code> , <code>text</code>	a character vector where matches are sought.
<code>ignore.case</code>	if <code>FALSE</code> , the pattern matching is <i>case sensitive</i> and if <code>TRUE</code> , case is ignored during matching.
<code>extended</code>	if <code>TRUE</code> , extended regular expression matching is used, and if <code>FALSE</code> basic regular expressions are used.
<code>perl</code>	logical. Should perl-compatible regexps be used? Has priority over <code>extended</code> .
<code>value</code>	if <code>FALSE</code> , a vector containing the ( <i>integer</i> ) indices of the matches determined by <code>grep</code> is returned, and if <code>TRUE</code> , a vector containing the matching elements themselves is returned.
<code>fixed</code>	logical. If <code>TRUE</code> , <code>pattern</code> is a string to be matched as is. Overrides all other arguments.
<code>replacement</code>	a replacement for matched pattern in <code>sub</code> and <code>gsub</code> .

**Details**

Arguments which should be character strings or character vectors are coerced to character if possible.

The two `*sub` functions differ only in that `sub` replaces only the first occurrence of a `pattern` whereas `gsub` replaces all occurrences.

For `regexpr` it is an error for `pattern` to be `NA`, otherwise `NA` is permitted and matches only itself.

The regular expressions used are those specified by POSIX 1003.2, either extended or basic, depending on the value of the extended argument, unless `perl = TRUE` when they are those of PCRE, <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>. (The exact set of patterns supported may depend on the version of PCRE installed on the system in use.)

### Value

For `grep` a vector giving either the indices of the elements of `x` that yielded a match or, if `value` is `TRUE`, the matched elements.

For `sub` and `gsub` a character vector of the same length as the original.

For `regexpr` an integer vector of the same length as `text` giving the starting position of the first match, or `-1` if there is none, with attribute `"match.length"` giving the length of the matched text (or `-1` for no match).

### Warning

The standard regular-expression code has been reported to be very slow or give errors when applied to extremely long character strings (tens of thousands of characters or more): the code used when `perl=TRUE` seems faster and more reliable for such usages.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (`grep`)

### See Also

[regular expression](#) (aka [regexp](#)) for the details of the pattern specification.

[agrep](#) for approximate matching.

[tolower](#), [toupper](#) and [chartr](#) for character translations. [charmatch](#), [pmatch](#), [match](#). [apropos](#) uses regexps and has nice examples.

### Examples

```
grep("[a-z]", letters)

txt <- c("arm","foot","lefroo", "bafoobar")
if(any(i <- grep("foo",txt)))
  cat("'foo' appears at least once in\n\t",txt,"\n")
i # 2 and 4
txt[i]

## Double all 'a' or 'b's; "\" must be escaped, i.e., 'doubled'
gsub("([ab])", "\\1_\\1_", "abc and ABC")

txt <- c("The", "licenses", "for", "most", "software", "are",
  "designed", "to", "take", "away", "your", "freedom",
  "to", "share", "and", "change", "it.",
  "", "By", "contrast,", "the", "GNU", "General", "Public", "License",
  "is", "intended", "to", "guarantee", "your", "freedom", "to",
```

```

    "share", "and", "change", "free", "software", "--",
    "to", "make", "sure", "the", "software", "is",
    "free", "for", "all", "its", "users")
  ( i <- grep("[gu]", txt) ) # indices
  stopifnot( txt[i] == grep("[gu]", txt, value = TRUE) )
  (ot <- sub("[b-e]", ".", txt))
  txt[ot != gsub("[b-e]", ".", txt)]#- gsub does "global" substitution

  txt[gsub("g","#", txt) !=
    gsub("g","#", txt, ignore.case = TRUE)] # the "G" words

  regexpr("en", txt)

  ## trim trailing white space
  str = 'Now is the time      '
  sub(' +$', '', str) ## spaces only
  sub('[:space:]+$$', '', str) ## white space, POSIX-style
  sub('\\s+$', '', str, perl = TRUE) ## Perl-style white space

```

---

groupGeneric

*Group Generic Functions*


---

## Description

Group generic functions can be defined with either S3 and S4 methods (with different groups). Methods are defined for the group of functions as a whole.

A method defined for an individual member of the group takes precedence over a method defined for the group as a whole.

When package **methods** is attached there are objects visible with the names of the group generics: these functions should never be called directly (a suitable error message will result if they are).

## Usage

```

## S3 methods have prototypes:
Math(x, ...)
Ops(e1, e2)
Summary(x, ...)
Complex(z)

## S4 methods have prototypes:
Arith(e1, e2)
Compare(e1, e2)
Ops(e1, e2)
Math(x)
Math2(x, digits)
Summary(x, ..., na.rm = FALSE)
Complex(z)

```

**Arguments**

`x`, `z`, `e1`, `e2` objects.  
`digits`            number of digits to be used in `round` or `signif`.  
`...`                further arguments passed to or from methods.  
`na.rm`              logical: should missing values be removed?

**S3 Group Dispatching**

There are four *groups* for which S3 methods can be written, namely the "Math", "Ops", "Summary" and "Complex" groups. These are not R objects, but methods can be supplied for them and base R contains `factor`, `data.frame` and `difftime` methods for the first three groups. (There are also a `ordered` method for Ops, `POSIXt` methods for Math and Ops, as well as a `ts` method for Ops in package `stats`.)

## 1. Group "Math":

- `abs`, `sign`, `sqrt`,  
`floor`, `ceiling`, `trunc`,  
`round`, `signif`
- `exp`, `log`,  
`cos`, `sin`, `tan`,  
`acos`, `asin`, `atan`  
`cosh`, `sinh`, `tanh`,  
`acosh`, `asinh`, `atanh`
- `lgamma`, `gamma`, `gammaCody`,  
`digamma`, `trigamma`, `tetragamma`, `pentagamma`
- `cumsum`, `cumprod`, `cummax`, `cummin`

## 2. Group "Ops":

- `"+"`, `"-"`, `"*"`, `"/"`, `"^"`, `"%%"`, `"%/%"`
- `"&"`, `"|"`, `"!"`
- `"=="`, `"!="`, `"<"`, `"<="`, `">="`, `">"`

## 3. Group "Summary":

- `all`, `any`
- `sum`, `prod`
- `min`, `max`
- `range`

## 4. Group Complex:

- `Arg`, `Conj`, `Im`, `Mod`, `Re`

Note that a method will be used for either one of these groups or one of its members *only* if it corresponds to a "class" attribute, as the internal code dispatches on `oldClass` and not on `class`. This is for efficiency: having to dispatch on, say, `Ops.integer` would be too slow.

The number of arguments supplied for "Math" group generic methods is not checked prior to dispatch. (Most have default methods expecting one argument, but three expect two.)

## S4 Group Dispatching

When package **methods** is attached (which it is by default), formal (S4) methods can be defined for groups.

The functions belonging to the various groups are as follows:

```

Arith "+", "-", "*", "^", "%%", "%/%", "/"
Compare "==", ">", "<", "!=", "<=", ">="
Ops "Arith", "Compare"
Math "log", "sqrt", "log10", "cumprod", "abs", "acos", "acosh", "asin",
      "asinh", "atan", "atanh", "ceiling", "cos", "cosh", "cumsum", "exp",
      "floor", "gamma", "lgamma", "sin", "sinh", "tan", "tanh", "trunc"
Math2 "round", "signif"
Summary "max", "min", "range", "prod", "sum", "any", "all"
Complex "Arg", "Conj", "Im", "Mod", "Re"

```

Functions with the group names exist in the **methods** package but should not be called directly.

All the functions in these groups (other than the group generics themselves) are basic functions in R. They are not by default S4 generic functions, and many of them are defined as primitives, meaning that they do not have formal arguments. However, you can still define formal methods for them. The effect of doing so is to create an S4 generic function with the appropriate arguments, in the environment where the method definition is to be stored. It all works more or less as you might expect, admittedly via a bit of trickery in the background.

Note: currently those members which are not primitive functions have to have been converted to S4 generic functions (preferably *before* setting an S4 group generic method) as it only sets methods for known S4 generics. This can be done by a call to `setGeneric`, for example `setGeneric("round", group="Math2")`.

## References

- Appendix A, *Classes and Methods of*  
 Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.  
 Chambers, J. M. (1998) *Programming with Data*. Springer, pp. 352–4.

## See Also

[methods](#) for methods of non-Internal generic functions.

## Examples

```

methods("Math")
methods("Ops")
methods("Summary")

d.fr <- data.frame(x=1:9, y=rnorm(9))
data.class(1 + d.fr) == "data.frame" ##-- add to d.f. ...

```

---

gzcon

*(De)compress I/O Through Connections*


---

### Description

gzcon provides a modified connection that wraps an existing connection, and decompresses reads or compresses writes through that connection. Standard gzip headers are assumed.

### Usage

```
gzcon(con, level = 6, allowNonCompressed = TRUE)
```

### Arguments

con	a connection.
level	integer between 0 and 9, the compression level when writing.
allowNonCompressed	logical. When reading, should non-compressed files (lacking the gzip magic header) be allowed?

### Details

If con is open then the modified connection is opened. Closing the wrapper connection will also close the underlying connection.

Reading from a connection which does not supply a gzip magic header is equivalent to reading from the original connection if allowNonCompressed is true, otherwise an error.

The original connection is unusable: any object pointing to it will now refer to the modified connection.

### Value

An object inheriting from class "connection". This is the same connection *number* as supplied, but with a modified internal structure.

### See Also

[gzfile](#)

### Examples

```
## Not run:
## This example may not still be available
## print the value to see what objects were created.
con <- url("http://hesweb1.med.virginia.edu/biostat/s/data/sav/kprats.sav")
print(load(con))
## End(Not run)
```

```
## gzfile and gzcon can inter-work.
## Of course here one would used gzfile, but file() can be replaced by
## any other connection generator.
zz <- gzfile("ex.gz", "w")
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzcon(file("ex.gz")))
close(zz)
unlink("ex.gz")

zz <- gzcon(file("ex.gz", "wb"))
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file = zz, sep = "\n")
close(zz)
readLines(zz <- gzfile("ex.gz"))
close(zz)
unlink("ex.gz")
```

---

Hyperbolic

*Hyperbolic Functions*

---

### Description

These functions give the obvious hyperbolic functions. They respectively compute the hyperbolic cosine, sine, tangent, and their inverses, arc-cosine, arc-sine, arc-tangent (or “*area cosine*”, etc).

### Usage

```
cosh(x)
sinh(x)
tanh(x)
acosh(x)
asinh(x)
atanh(x)
```

### Arguments

`x` a numeric vector

### Details

These are generic functions: methods can be defined for them individually or via the [Math](#) group generic.

### See Also

The trigonometric functions, [cos](#), [sin](#), [tan](#), and their inverses [acos](#), [asin](#), [atan](#).

The logistic distribution function [plogis](#) is a shifted version of [tanh](#)( ).

---

`identical`*Test Objects for Exact Equality*

---

**Description**

The safe and reliable way to test two objects for being *exactly* equal. It returns TRUE in this case, FALSE in every other case.

**Usage**

```
identical(x, y)
```

**Arguments**

`x`, `y`            any R objects.

**Details**

A call to `identical` is the way to test exact equality in `if` and `while` statements, as well as in logical expressions that use `&&` or `||`. In all these applications you need to be assured of getting a single logical value.

Users often use the comparison operators, such as `==` or `!=`, in these situations. It looks natural, but it is not what these operators are designed to do in R. They return an object like the arguments. If you expected `x` and `y` to be of length 1, but it happened that one of them wasn't, you will *not* get a single FALSE. Similarly, if one of the arguments is NA, the result is also NA. In either case, the expression `if(x == y) . . .` won't work as expected.

The function `all.equal` is also sometimes used to test equality this way, but it was intended for something different. First, it tries to allow for "reasonable" differences in numeric results. Second, it returns a descriptive character vector instead of FALSE when the objects do not match. Therefore, it is not the right function to use for reliable testing either. (If you *do* want to allow for numeric fuzziness in comparing objects, you can combine `all.equal` and `identical`, as shown in the examples below.)

The computations in `identical` are also reliable and usually fast. There should never be an error. The only known way to kill `identical` is by having an invalid pointer at the C level, generating a memory fault. It will usually find inequality quickly. Checking equality for two large, complicated objects can take longer if the objects are identical or nearly so, but represent completely independent copies. For most applications, however, the computational cost should be negligible.

As from R 1.6.0, `identical` sees NaN as different from `as.double(NA)`, but all NaNs are equal (and all NA of the same type are equal).

**Value**

A single logical value, TRUE or FALSE, never NA and never anything other than a single value.

**Author(s)**

John Chambers

## References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

## See Also

[all.equal](#) for descriptions of how two objects differ; [Comparison](#) for operators that generate elementwise comparisons.

## Examples

```
identical(1, NULL) ## FALSE -- don't try this with ==
identical(1, 1.)  ## TRUE in R (both are stored as doubles)
identical(1, as.integer(1)) ## FALSE, stored as different types

x <- 1.0; y <- 0.99999999999
## how to test for object equality allowing for numeric fuzz
identical(all.equal(x, y), TRUE)
## If all.equal thinks the objects are different, it returns a
## character string, and this expression evaluates to FALSE

# even for unusual R objects :
identical(.GlobalEnv, environment())
```

---

ifelse

*Conditional Element Selection*

---

## Description

`ifelse` returns a value with the same shape as `test` which is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is `TRUE` or `FALSE`.

## Usage

```
ifelse(test, yes, no)
```

## Arguments

<code>test</code>	an object which can be coerced to logical mode.
<code>yes</code>	return values for true elements of <code>test</code> .
<code>no</code>	return values for false elements of <code>test</code> .

## Details

If `yes` or `no` are too short, their elements are recycled. `yes` will be evaluated if and only if any element of `test` is true, and analogously for `no`.

Missing values in `test` giving missing values in the result.

**Value**

A vector of the same length and attributes (including class) as `test` and data values from the values of `yes` or `no`. The mode of the answer will be coerced to logical to accommodate first any values taken from `yes` and then any values taken from `no`.

**Warning**

The mode of the result may depend on the value of `test`, and the class attribute of the result is taken from `test` and may be inappropriate for the values selected from `yes` and `no`.

Sometimes it is better to use a construction such as `(tmp <- yes; tmp[!test] <- no[!test]; tmp)`, possibly extended to handle missing values in `test`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[if](#).

**Examples**

```
x <- c(6:-4)
sqrt(x)#- gives warning
sqrt(ifelse(x >= 0, x, NA))# no warning

## Note: the following also gives the warning !
ifelse(x >= 0, sqrt(x), NA)
```

---

integer

*Integer Vectors*

---

**Description**

Creates or tests for objects of type "integer".

**Usage**

```
integer(length = 0)
as.integer(x, ...)
is.integer(x)
```

**Arguments**

<code>length</code>	desired length.
<code>x</code>	object to be coerced or tested.
<code>...</code>	further arguments passed to or from other methods.

## Details

Integer vectors exist so that data can be passed to C or Fortran code which expects them, and so that small integer data can be represented exactly and compactly.

Note that on almost all implementations of R the range of representable integers is restricted to about  $\pm 2 \times 10^9$ : `doubles` can hold much larger integers exactly.

## Value

`integer` creates a integer vector of the specified length. Each element of the vector is equal to 0. `as.integer` attempts to coerce its argument to be of integer type. The answer will be `NA` unless the coercion succeeds. Real values larger in modulus than the largest integer are coerced to `NA` (unlike S which gives the most extreme integer of the same sign). Non-integral numeric values are truncated towards zero (i.e., `as.integer(x)` equals `trunc(x)` there), and imaginary parts of complex numbers are discarded (with a warning). Like `as.vector` it strips attributes including names.

`is.integer` returns `TRUE` or `FALSE` depending on whether its argument is of integer type or not. `is.integer` is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#). There is a method for factors which returns `FALSE`. (Prior to R 2.0.0, there was no such method and for most (but not all) factors `is.integer` returned `TRUE`.)

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[round](#) (and [ceiling](#) and [floor](#) on that help page) to convert to integral values.

## Examples

```
## as.integer() truncates:  
x <- pi * c(-1:1,10)  
as.integer(x)
```

---

interaction

*Compute Factor Interactions*

---

## Description

`interaction` computes a factor which represents the interaction of the given factors. The result of `interaction` is always unordered.

## Usage

```
interaction(..., drop = FALSE, sep = ".")
```

**Arguments**

<code>...</code>	the factors for which interaction is to be computed, or a single list giving those factors.
<code>drop</code>	if <code>drop</code> is <code>TRUE</code> , empty factor levels are dropped from the result. The default is to retain all factor levels.
<code>sep</code>	string to construct the new level labels by joining the constituent ones.

**Value**

A factor which represents the interaction of the given factors. The levels are labelled as the levels of the individual factors joined by `sep`, i.e. `.` by default.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

`factor`; `:` where `f:g` is the same as `interaction(f,g, sep=":")` when `f` and `g` are factors.

**Examples**

```
a <- gl(2, 4, 8)
b <- gl(2, 2, 8, label = c("ctrl", "treat"))
s <- gl(2, 1, 8, label = c("M", "F"))
interaction(a, b)
interaction(a, b, s, sep = ":")
```

---

interactive

*Is R Running Interactively?*

---

**Description**

Return `TRUE` when `R` is being used interactively and `FALSE` otherwise.

**Usage**

```
interactive()
```

**See Also**

`source`, `.First`

**Examples**

```
.First <- function() if(interactive()) x11()
```

---

Internal

*Call an Internal Function*

---

### Description

`.Internal` performs a call to an internal code which is built in to the R interpreter. Only true R wizards should even consider using this function.

### Usage

```
.Internal(call)
```

### Arguments

`call` a call expression

### See Also

[.Primitive](#), [.C](#), [.Fortran](#).

---

InternalMethods

*Internal Generic Functions*

---

### Description

Many R-internal functions are *generic* and allow methods to be written for.

### Details

The following builtin functions are *generic* as well, i.e., you can write [methods](#) for them:

`[`, `[[`, `$`, `[<-`, `[[<-`, `$<-`,

`length`, `length<-`,

`dimnames<-`, `dimnames`, `dim<-`, `dim`

`c`, `unlist`,

`as.character`, `as.vector`, `is.array`, `is.atomic`, `is.call`, `is.character`,  
`is.complex`, `is.double`, `is.environment`, `is.function`, `is.integer`,  
`is.language`, `is.logical`, `is.list`, `is.matrix`, `is.na`, `is.nan`, `is.null`,  
`is.numeric`, `is.object`, `is.pairlist`, `is.recursive`, `is.single`, `is.symbol`.

### See Also

[methods](#) for the methods of non-Internal generic functions.

---

invisible	<i>Change the Print Mode to Invisible</i>
-----------	---

---

### Description

Return a (temporarily) invisible copy of an object.

### Usage

```
invisible(x)
```

### Arguments

`x` an arbitrary R object.

### Details

This function can be useful when it is desired to have functions return values which can be assigned, but which do not print when they are not assigned.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[return, function.](#)

### Examples

```
# These functions both return their argument
f1 <- function(x) x
f2 <- function(x) invisible(x)
f1(1)# prints
f2(1)# does not
```

---

`is.finite`*Finite, Infinite and NaN Numbers*

---

**Description**

`is.finite` and `is.infinite` return a vector of the same length as `x`, indicating which elements are finite (not infinite and not missing).

`Inf` and `-Inf` are positive and negative “infinity” whereas `NaN` means “Not a Number”.

**Usage**

```
is.finite(x)
is.infinite(x)
Inf
NaN
is.nan(x)
```

**Arguments**

`x` (numerical) object to be tested.

**Details**

`is.finite` returns a vector of the same length as `x` the `j`th element of which is `TRUE` if `x[j]` is finite (i.e., it is not one of the values `NA`, `NaN`, `Inf` or `-Inf`). All elements of character and generic (list) vectors are false, so `is.finite` is only useful for logical, integer, numeric and complex vectors. Complex numbers are finite if both the real and imaginary parts are.

`is.infinite` returns a vector of the same length as `x` the `j`th element of which is `TRUE` if `x[j]` is infinite (i.e., equal to one of `Inf` or `-Inf`).

`is.nan` tests if a numeric value is `NaN`. Do not test equality to `NaN`, or even use `identical`, since systems typically have many different `NaN` values. In most ports of `R` one of these is used for the numeric missing value `NA`. It is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

**Note**

In `R`, basically all mathematical functions (including basic [Arithmetic](#)), are supposed to work properly with `+/- Inf` and `NaN` as input or output.

The basic rule should be that calls and relations with `Infs` really are statements with a proper mathematical *limit*.

**References**

ANSI/IEEE 754 Floating-Point Standard.

This link does not work any more (2003/12) Currently (6/2002), Bill Metzenthén's (billm@suburbia.net) tutorial and examples at <http://www.suburbia.net/~billm/>

**See Also**

[NA](#), ‘*Not Available*’ which is not a number as well, however usually used for missing values and applies to many modes, not just numeric.

**Examples**

```
pi / 0 ## = Inf a non-zero number divided by zero creates infinity
0 / 0  ## = NaN

1/0 + 1/0# Inf
1/0 - 1/0# NaN

stopifnot(
  1/0 == Inf,
  1/Inf == 0
)
sin(Inf)
cos(Inf)
tan(Inf)
```

---

is.function	<i>Is an Object of Type Function?</i>
-------------	---------------------------------------

---

**Description**

Checks whether its argument is a function.

**Usage**

```
is.function(x)
```

**Arguments**

x                    an R object.

**Details**

is.function is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

**Value**

TRUE if x is a function, and FALSE otherwise.

---

is.language	<i>Is an Object a Language Object?</i>
-------------	--

---

### Description

is.language returns TRUE if x is either a variable [name](#), a [call](#), or an [expression](#).

### Usage

```
is.language(x)
```

### Arguments

x                    object to be tested.

### Details

is.language is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### Examples

```
ll <- list(a = expression(x^2 - 2*x + 1), b = as.name("Jim"),
          c = as.expression(exp(1)), d = call("sin", pi))
sapply(ll, typeof)
sapply(ll, mode)
stopifnot(sapply(ll, is.language))
```

---

is.object	<i>Is an Object "internally classed"?</i>
-----------	---

---

### Description

A function rather for internal use. It returns TRUE if the object x has the R internal OBJECT attribute set, and FALSE otherwise.

### Usage

```
is.object(x)
```

**Arguments**

x                    object to be tested.

**Details**

is.object is generic: you can write methods to handle specific classes of objects, see [Internal-Methods](#).

**See Also**

[class](#), and [methods](#).

**Examples**

```
is.object(1) # FALSE
is.object(as.factor(1:3)) # TRUE
```

---

is.R

*Are we using R, rather than S?*

---

**Description**

Test if running under R.

**Usage**

```
is.R()
```

**Details**

The function has been written such as to correctly run in all versions of R, S and S-PLUS. In order for code to be runnable in both R and S dialects, either your the code must define is.R or use it as

```
if (exists("is.R") && is.function(is.R) && is.R()) {
  ## R-specific code
} else {
  ## S-version of code
}
```

**Value**

is.R returns TRUE if we are using R and FALSE otherwise.

**See Also**

[R.version](#), [system](#).

**Examples**

```
x <- runif(20); small <- x < 0.4
## 'which()' only exists in R:
if(is.R()) which(small) else seq(along=small)[small]
```

---

is.recursive	<i>Is an Object Atomic or Recursive?</i>
--------------	--

---

**Description**

`is.atomic` returns TRUE if `x` does not have a list structure and FALSE otherwise.

`is.recursive` returns TRUE if `x` has a recursive (list-like) structure and FALSE otherwise.

**Usage**

```
is.atomic(x)
is.recursive(x)
```

**Arguments**

`x` object to be tested.

**Details**

These are generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[is.list](#), [is.language](#), etc, and the `demo("is.things")`.

**Examples**

```
is.a.r <- function(x) c(is.atomic(x), is.recursive(x))

is.a.r(c(a=1,b=3))      # TRUE FALSE
is.a.r(list())         # FALSE TRUE ??
is.a.r(list(2))        # FALSE TRUE
is.a.r(lm)             # FALSE TRUE
is.a.r(y ~ x)          # FALSE TRUE
is.a.r(expression(x+1)) # FALSE TRUE (not in 0.62.3!)
```

---

<code>is.single</code>	<i>Is an Object of Single Precision Type?</i>
------------------------	---

---

**Description**

`is.single` reports an error. There are no single precision values in R.

**Usage**

```
is.single(x)
```

**Arguments**

`x`                    object to be tested.

**Details**

`is.single` is generic: you can write methods to handle specific classes of objects, see [Internal-  
Methods](#).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

<code>jitter</code>	<i>Add 'Jitter' (Noise) to Numbers</i>
---------------------	--

---

**Description**

Add a small amount of noise to a numeric vector.

**Usage**

```
jitter(x, factor=1, amount = NULL)
```

**Arguments**

`x`                    numeric to which *jitter* should be added.  
`factor`                numeric  
`amount`                numeric; if positive, used as *amount* (see below), otherwise, if = 0 the default is `factor * z/50`.  
Default (NULL): `factor * d/5` where `d` is about the smallest difference between `x` values.

**Details**

The result, say  $r$ , is  $r \leftarrow x + \text{runif}(n, -a, a)$  where  $n \leftarrow \text{length}(x)$  and  $a$  is the amount argument (if specified).

Let  $z \leftarrow \max(x) - \min(x)$  (assuming the usual case). The amount  $a$  to be added is either provided as *positive* argument `amount` or otherwise computed from  $z$ , as follows:

If `amount == 0`, we set  $a \leftarrow \text{factor} * z/50$  (same as `S`).

If `amount` is `NULL` (*default*), we set  $a \leftarrow \text{factor} * d/5$  where  $d$  is the smallest difference between adjacent unique (apart from fuzz)  $x$  values.

**Value**

`jitter(x, ...)` returns a numeric of the same length as  $x$ , but with an amount of noise added in order to break ties.

**Author(s)**

Werner Stahel and Martin Maechler, ETH Zurich

**References**

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P.A. (1983) *Graphical Methods for Data Analysis*. Wadsworth; figures 2.8, 4.22, 5.4.

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

[rug](#) which you may want to combine with `jitter`.

**Examples**

```
round(jitter(c(rep(1,3), rep(1.2, 4), rep(3,3))), 3)
## These two 'fail' with S-plus 3.x:
jitter(rep(0, 7))
jitter(rep(10000,5))
```

---

kappa

*Estimate the Condition Number*


---

**Description**

An estimate of the condition number of a matrix or of the  $R$  matrix of a  $QR$  decomposition, perhaps of a linear fit. The condition number is defined as the ratio of the largest to the smallest *non-zero* singular value of the matrix.

**Usage**

```

kappa(z, ...)
## S3 method for class 'lm':
kappa(z, ...)
## Default S3 method:
kappa(z, exact = FALSE, ...)
## S3 method for class 'qr':
kappa(z, ...)

kappa.tri(z, exact = FALSE, ...)

```

**Arguments**

<code>z</code>	A matrix or a the result of <code>qr</code> or a fit from a class inheriting from "lm".
<code>exact</code>	logical. Should the result be exact?
<code>...</code>	further arguments passed to or from other methods.

**Details**

If `exact = FALSE` (the default) the condition number is estimated by a cheap approximation. Following S, this uses the LINPACK routine 'dtrco.f'. However, in R (or S) the exact calculation is also likely to be quick enough.

`kappa.tri` is an internal function called by `kappa.qr`.

**Value**

The condition number, *kappa*, or an approximation if `exact = FALSE`.

**Author(s)**

The design was inspired by (but differs considerably from) the S function of the same name described in Chambers (1992).

**References**

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[svd](#) for the singular value decomposition and [qr](#) for the *QR* one.

**Examples**

```

kappa(x1 <- cbind(1,1:10))# 15.71
kappa(x1, exact = TRUE)      # 13.68
kappa(x2 <- cbind(x1,2:11))# high! [x2 is singular!]

hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }

```

```
sv9 <- svd(h9 <- hilbert(9))$ d
kappa(h9)# pretty high!
kappa(h9, exact = TRUE) == max(sv9) / min(sv9)
kappa(h9, exact = TRUE) / kappa(h9) # .677 (i.e., rel.error = 32%)
```

---

kronecker

*Kronecker products on arrays*


---

### Description

Computes the generalised kronecker product of two arrays, X and Y. `kronecker(X, Y)` returns an array A with dimensions `dim(X) * dim(Y)`.

### Usage

```
kronecker(X, Y, FUN = "*", make.dimnames = FALSE, ...)
X %x% Y
```

### Arguments

X	A vector or array.
Y	A vector or array.
FUN	a function; it may be a quoted string.
make.dimnames	Provide dimnames that are the product of the dimnames of X and Y.
...	optional arguments to be passed to FUN.

### Details

If X and Y do not have the same number of dimensions, the smaller array is padded with dimensions of size one. The returned array comprises submatrices constructed by taking X one term at a time and expanding that term as `FUN(x, Y, ...)`.

`%x%` is an alias for `kronecker` (where FUN is hardwired to `"*"`).

### Author(s)

Jonathan Rougier, [J.C.Rougier@durham.ac.uk](mailto:J.C.Rougier@durham.ac.uk)

### References

Shayle R. Searle (1982) *Matrix Algebra Useful for Statistics*. John Wiley and Sons.

### See Also

[outer](#), on which `kronecker` is built and `%*%` for usual matrix multiplication.

**Examples**

```
# simple scalar multiplication
( M <- matrix(1:6, ncol=2) )
kronecker(4, M)
# Block diagonal matrix:
kronecker(diag(1, 3), M)

# ask for dimnames

fred <- matrix(1:12, 3, 4, dimnames=list(LETTERS[1:3], LETTERS[4:7]))
bill <- c("happy" = 100, "sad" = 1000)
kronecker(fred, bill, make.dimnames = TRUE)

bill <- outer(bill, c("cat"=3, "dog"=4))
kronecker(fred, bill, make.dimnames = TRUE)
```

---

labels

*Find Labels from Object*


---

**Description**

Find a suitable set of labels from an object for use in printing or plotting, for example. A generic function.

**Usage**

```
labels(object, ...)
```

**Arguments**

object      Any R object: the function is generic.  
...          further arguments passed to or from other methods.

**Value**

A character vector or list of such vectors. For a vector the results is the names or `seq(along=x)`, for a data frame or array it is the dimnames (with NULL expanded to `seq(len=d[i])`), for a `terms` object it is the term labels and for an `lm` object it is the term labels for estimable terms.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

---

`lapply`*Apply a Function over a List or Vector*

---

### Description

`lapply` returns a list of the same length as `X`. Each element of which is the result of applying `FUN` to the corresponding element of `X`.

`sapply` is a “user-friendly” version of `lapply` also accepting vectors as `X`, and returning a vector or matrix with `dimnames` if appropriate.

`replicate` is a wrapper for the common use of `sapply` for repeated evaluation of an expression (which will usually involve random number generation).

### Usage

```
lapply(X, FUN, ...)  
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)  
  
replicate(n, expr, simplify = TRUE)
```

### Arguments

<code>X</code>	list or vector to be used.
<code>FUN</code>	the function to be applied. In the case of functions like <code>+</code> , <code>%*%</code> , etc., the function name must be quoted.
<code>...</code>	optional arguments to <code>FUN</code> .
<code>simplify</code>	logical; should the result be simplified to a vector or matrix if possible?
<code>USE.NAMES</code>	logical; if <code>TRUE</code> and if <code>X</code> is character, use <code>X</code> as <code>names</code> for the result unless it had names already.
<code>n</code>	Number of replications.
<code>expr</code>	Expression to evaluate repeatedly.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[apply](#), [tapply](#).

**Examples**

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
# compute the list mean for each list element
lapply(x,mean)
# median and quartiles for each list element
lapply(x, quantile, probs = 1:3/4)
sapply(x, quantile)
i39 <- sapply(3:9, seq) # list of vectors
sapply(i39, fivenum)

hist(replicate(100, mean(rexp(10))))
```

---

Last.value

*Value of Last Evaluated Expression*


---

**Description**

The value of the internal evaluation of a top-level R expression is always assigned to `.Last.value` (in `package:base`) before further processing (e.g., printing).

**Usage**

```
.Last.value
```

**Details**

The value of a top-level assignment *is* put in `.Last.value`, unlike `S`.

Do not assign to `.Last.value` in the workspace, because this will always mask the object of the same name in `package:base`.

**See Also**

[eval](#)

**Examples**

```
## These will not work correctly from example(),
## but they will in make check or if pasted in,
## as example() does not run them at the top level
gamma(1:15)          # think of some intensive calculation...
fac14 <- .Last.value # keep them

library("splines") # returns invisibly
.Last.value       # shows what library(.) above returned
```

---

length	<i>Length of an Object</i>
--------	----------------------------

---

### Description

Get or set the length of vectors (including lists) and factors, and of any other R object for which a method has been defined.

### Usage

```
length(x)
length(x) <- value
```

### Arguments

x	a vector or list or factor.
value	an integer.

### Details

Both functions are generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

The replacement form can be used to reset the length of a vector. If a vector is shortened, extra values are discarded and when a vector is lengthened, it is padded out to its new length with [NAs](#).

### Value

The length of x as an [integer](#) of length 1, if x is (or can be coerced to) a vector or list. Otherwise, length returns NA.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

nchar for counting the number of characters in character vectors.

### Examples

```
length(diag(4))# = 16 (4 x 4)
length(options())# 12 or more
length(y ~ x1 + x2 + x3)# 3
length(expression(x, {y <- x^2; y+2}, x^y)) # 3
```

---

levels

*Levels Attributes*

---

### Description

levels provides access to the levels attribute of a variable. The first form returns the value of the levels of its argument and the second sets the attribute.

The assignment form ("levels<-") of levels is a generic function and new methods can be written for it. The most important method is that for [factors](#):

### Usage

```
levels(x)
levels(x) <- value
```

### Arguments

x	an object, for example a factor.
value	A valid value for levels(x). For the default method, NULL or a character vector. For the <code>factor</code> method, a vector of character strings with length at least the number of levels of x, or a named list specifying how to rename the levels.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[nlevels](#).

### Examples

```
## assign individual levels
x <- gl(2, 4, 8)
levels(x)[1] <- "low"
levels(x)[2] <- "high"
x

## or as a group
y <- gl(2, 4, 8)
levels(y) <- c("low", "high")
y

## combine some levels
z <- gl(3, 2, 12)
levels(z) <- c("A", "B", "A")
```

```
z

## same, using a named list
z <- gl(3, 2, 12)
levels(z) <- list(A=c(1,3), B=2)
z

## we can add levels this way:
f <- factor(c("a","b"))
levels(f) <- c("c", "a", "b")
f

f <- factor(c("a","b"))
levels(f) <- list(C="C", A="a", B="b")
f
```

---

libPaths

*Search Paths for Packages*

---

## Description

`.libPaths` gets/sets the library trees within which packages are looked for.

## Usage

```
.libPaths(new)
```

```
.Library
```

## Arguments

`new` a character vector with the locations of R library trees.

## Details

`.Library` is a character string giving the location of the default library, the ‘library’ subdirectory of `R_HOME`.

`.libPaths` is used for getting or setting the library trees that R knows about (and hence uses when looking for packages). If called with argument `new`, the library search path is set to the existing files in `unique(new, .Library)` and this is returned. If given no argument, a character vector with the currently known library trees is returned.

The library search path is initialized at startup from the environment variable `R_LIBS` (which should be a colon-separated list of directories at which R library trees are rooted) by calling `.libPaths` with the directories specified in `R_LIBS`.

## Value

A character vector of file paths.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[library](#)

**Examples**

```
.libPaths()                # all library trees R knows about
```

---

library	<i>Loading and Listing of Packages</i>
---------	--

---

**Description**

library and require load add-on packages.

.First.lib is called when a package is loaded; .Last.lib is called when a package is detached.

**Usage**

```
library(package, help, pos = 2, lib.loc = NULL,
         character.only = FALSE, logical.return = FALSE,
         warn.conflicts = TRUE,
         keep.source = getOption("keep.source.pkgs"),
         verbose = getOption("verbose"),
         version)

require(package, quietly = FALSE, warn.conflicts = TRUE,
        keep.source = getOption("keep.source.pkgs"),
        character.only = FALSE, version, save = TRUE)

.First.lib(libname, pkgname)
.Last.lib(libpath)
```

**Arguments**

package, help  
the name of a package, given as a [name](#) or literal character string, or a character string, depending on whether character.only is FALSE (default) or TRUE.

pos  
the position on the search list at which to attach the loaded package. Note that .First.lib may attach other packages, and pos is computed *after* .First.lib has been run. Can also be the name of a position on the current search list as given by [search\(\)](#).

<code>lib.loc</code>	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.
<code>character.only</code>	a logical indicating whether <code>package</code> or <code>help</code> can be assumed to be character strings.
<code>version</code>	A character string denoting a version number of the package to be loaded. If no version is given, a suitable default is chosen.
<code>logical.return</code>	logical. If it is TRUE, FALSE or TRUE is returned to indicate success.
<code>warn.conflicts</code>	logical. If TRUE, warnings are printed about <code>conflicts</code> from attaching the new package, unless that package contains an object <code>.conflicts.OK</code> .
<code>keep.source</code>	logical. If TRUE, functions “keep their source” including comments, see argument <code>keep.source</code> to <code>options</code> .
<code>verbose</code>	a logical. If TRUE, additional diagnostics are printed.
<code>quietly</code>	a logical. If TRUE, no message confirming package loading is printed.
<code>save</code>	logical or environment. If TRUE, a call to <code>require</code> from the source for a package will save the name of the required package in the variable <code>".required"</code> , allowing function <code>detach</code> to warn if a required package is detached. See section ‘Packages that require other packages’ below.
<code>libname</code>	a character string giving the library directory where the package was found.
<code>pkgname</code>	a character string giving the name of the package, including the version number if the package was installed with <code>--with-package-versions</code> .
<code>libpath</code>	a character string giving the complete path to the package.

## Details

`library(package)` and `require(package)` both load the package with name `package`. `require` is designed for use inside other functions; it returns FALSE and gives a warning (rather than an error as `library()` does) if the package does not exist. Both functions check and update the list of currently loaded packages and do not reload code that is already loaded.

For large packages, setting `keep.source = FALSE` may save quite a bit of memory.

If `library` is called with no `package` or `help` argument, it lists all available packages in the libraries specified by `lib.loc`, and returns the corresponding information in an object of class `"libraryIQR"`. The structure of this class may change in future versions. In earlier versions of R, only the names of all available packages were returned; use `.packages(all = TRUE)` for obtaining these. Note that `installed.packages()` returns even more information.

`library(help = somename)` computes basic information about the package `somename`, and returns this in an object of class `"packageInfo"`. The structure of this class may change in future versions. When used with the default value (NULL) for `lib.loc`, the loaded packages are searched before the libraries.

`.First.lib` is called when a package without a namespace is loaded by `library`. (For packages with namespaces see `.onLoad`.) It is called with two arguments, the name of the library directory where the package was found (i.e., the corresponding element of `lib.loc`), and

the name of the package (in that order). It is a good place to put calls to `library.dynam` which are needed when loading a package into this function (don't call `library.dynam` directly, as this will not work if the package is not installed in a "standard" location). `.First.lib` is invoked after the search path interrogated by `search()` has been updated, so `as.environment(match("package:name", search()))` will return the environment in which the package is stored. If calling `.First.lib` gives an error the loading of the package is abandoned, and the package will be unavailable. Similarly, if the option `".First.lib"` has a list element with the package's name, this element is called in the same manner as `.First.lib` when the package is loaded. This mechanism allows the user to set package "load hooks" in addition to startup code as provided by the package maintainers, but `setHook` is preferred.

`.Last.lib` is called when a package is detached. Beware that it might be called if `.First.lib` has failed, so it should be written defensively. (It is called within `try`, so errors will not stop the package being detached.)

### Value

`library` returns the list of loaded (or available) packages (or `TRUE` if `logical.return` is `TRUE`). `require` returns a logical indicating whether the required package is available.

### Packages that require other packages

The source code for a package that requires one or more other packages should have a call to `require`, preferably near the beginning of the source, and of course before any code that uses functions, classes or methods from the other package. The default for argument `save` will save the names of all required packages in the environment of the new package. The saved package names are used by `detach` when a package is detached to warn if other packages still require the detached package. Also, if a package is installed with saved image (see `INSTALL`), the saved package names are used to require these packages when the new package is attached.

### Formal methods

`library` takes some further actions when package **methods** is attached (as it is by default). Packages may define formal generic functions as well as re-defining functions in other packages (notably **base**) to be generic, and this information is cached whenever such a package is loaded after **methods** and re-defined functions are excluded from the list of conflicts. The check requires looking for a pattern of objects; the pattern search may be avoided by defining an object `.noGenerics` (with any value) in the package. Naturally, if the package *does* have any such methods, this will prevent them from being used.

### Note

`library` and `require` can only load an *installed* package, and this is detected by having a 'DESCRIPTION' file containing a `Built:` field.

Under Unix-alikes, the code checks that the package was installed under a similar operating system as given by `R.version$platform` (the canonical name of the platform under which R was compiled), provided it contains compiled code. Packages which do not contain compiled code can be shared between Unix-alikes, but not to other OSes because of potential problems with line endings and OS-specific help files.

As of version 2.0.0, the package name given to `library` and `require` must match the name given in the package's DESCRIPTION file exactly, even on case-insensitive file systems such as MS Windows.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[.libPaths](#), [.packages](#).

[attach](#), [detach](#), [search](#), [objects](#), [autoload](#), [library.dynam](#), [data](#), [install.packages](#) and [installed.packages](#); [INSTALL](#), [REMOVE](#).

## Examples

```
library()                # list all available packages
library(lib = .Library) # list all packages in the default library
library(help = splines) # documentation on package 'splines'
library(splines)        # load package 'splines'
require(splines)        # the same
search()                # "splines", too
detach("package:splines")

# if the package name is in a character vector, use
pkg <- "splines"
library(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep=":"), search()))

require(pkg, character.only = TRUE)
detach(pos = match(paste("package", pkg, sep=":"), search()))

require(nonexistent)    # FALSE
## Not run:
## Suppose a package needs to call a shared library named 'fooEXT',
## where 'EXT' is the system-specific extension. Then you should use
.First.lib <- function(lib, pkg) {
  library.dynam("foo", pkg, lib)
}

## if you want to mask as little as possible, use
library(mypkg, pos = "package:base")
## End(Not run)
```

**Description**

Load the specified file of compiled code if it has not been loaded already, or unloads it.

**Usage**

```
library.dynam(chname, package = NULL, lib.loc = NULL,
              verbose = getOption("verbose"),
              file.ext = .Platform$dynlib.ext, ...)
library.dynam.unload(chname, libpath,
                    verbose = getOption("verbose"),
                    file.ext = .Platform$dynlib.ext)
.dynLibs(new)
```

**Arguments**

<code>chname</code>	a character string naming a shared library to load.
<code>package</code>	a character vector with the names of packages to search through, or <code>NULL</code> . By default, all packages in the search path are used.
<code>lib.loc</code>	a character vector describing the location of R library trees to search through, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known.
<code>libpath</code>	the path to the loaded package whose shared library is to be unloaded.
<code>verbose</code>	a logical value indicating whether an announcement is printed on the console before loading the shared library. The default value is taken from the <code>verbose</code> entry in the system options.
<code>file.ext</code>	the extension to append to the file name to specify the library to be loaded. This defaults to the appropriate value for the operating system.
<code>...</code>	additional arguments needed by some libraries that are passed to the call to <code>dyn.load</code> to control how the library is loaded.
<code>new</code>	a character vector of packages which have loaded shared libraries.

**Details**

`library.dynam` is designed to be used inside a package rather than at the command line, and should really only be used inside `.First.lib` on `.onLoad`. The system-specific extension for shared libraries (e.g., `.so` or `.sl` on Unix systems) should not be added.

`library.dynam.unload` is designed for use in `.Last.lib` or `.onUnload`.

`.dynLibs` is used for getting or setting the packages that have loaded shared libraries (using `library.dynam`). Versions of R prior to 1.6.0 used an internal global variable `.Dyn.libs` for storing this information: this variable is now defunct.

**Value**

If `chname` is not specified, `library.dynam` returns a character vector with the names of packages which have used it in the current R session to load shared libraries. This vector is returned as `invisible`, unless the `chname` argument is missing.

If `chname` is specified, the return value is an object of class `DLLInfo` that identifies the DLL and can be used in future calls. For packages that have namespaces, a list of these objects is stored in the namespace's environment for use at run-time.

`library.dynam.unload` returns the updated character vector, invisibly.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[.First.lib](#), [library](#), [dyn.load](#), [.packages](#), [.libPaths](#)

[SHLIB](#) for how to create suitable shared libraries.

## Examples

```
library.dynam() # which packages have been "dynamically loaded"
```

---

license                      *The R License Terms*

---

## Description

The license terms under which R is distributed.

## Usage

```
license()  
licence()
```

## Details

R is distributed under the terms of the GNU GENERAL PUBLIC LICENSE Version 2, June 1991. A copy of this license is in `'$R_HOME/COPYING'`.

A small number of files (the API header files and import library) are distributed under the LESSER GNU GENERAL PUBLIC LICENSE version 2.1. A copy of this license is in `'$R_HOME/COPYING.LIB'`.

list

Lists – Generic and Dotted Pairs

**Description**

Functions to construct, coerce and check for all kinds of R lists.

**Usage**

```
list(...)
pairlist(...)

as.list(x, ...)
as.pairlist(x)
as.list.environment(x, all.names=FALSE, ...)

is.list(x)
is.pairlist(x)

alist(...)
```

**Arguments**

...	objects.
x	object to be coerced or tested.
all.names	a logical indicating whether to copy all values in <code>as.list.environment/</code>

**Details**

Most lists in R internally are *Generic Vectors*, whereas traditional *dotted pair* lists (as in LISP) are still available.

The arguments to `list` or `pairlist` are of the form `value` or `tag=value`. The functions return a list composed of its arguments with each value either tagged or untagged, depending on how the argument was specified.

`alist` is like `list`, except in the handling of tagged arguments with no value. These are handled as if they described function arguments with no default (cf. [formals](#)), whereas `list` simply ignores them.

`as.list` attempts to coerce its argument to list type. For functions, this returns the concatenation of the list of formals arguments and the function body. For expressions, the list of constituent calls is returned.

`is.list` returns TRUE iff its argument is a list *or* a pairlist of length > 0, whereas `is.pairlist` only returns TRUE in the latter case.

`is.list` and `is.pairlist` are generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

`as.list.environment` copies the named values from an environment to a list. The user can request that all named objects are copied (normally names that begin with a dot are not). The output is not sorted and no parent environments are searched.

An empty pairlist, `pairlist()` is the same as `NULL`. This is different from `list()`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

`vector(., mode="list")`, `c`, for concatenation; `formals`.

## Examples

```
# create a plotting structure
pts <- list(x=cars[,1], y=cars[,2])
plot(pts)

# Argument lists
f <- function()x
# Note the specification of a "... " argument:
formals(f) <- al <- alist(x=, y=2, ...=)
f
al

##environment->list coercion

e1 <- new.env()
e1$a <- 10
e1$b <- 20
as.list(e1)
```

---

list.files

*List the Files in a Directory/Folder*

---

## Description

This function produces a list containing the names of files in the named directory. `dir` is an alias.

## Usage

```
list.files(path = ".", pattern = NULL, all.files = FALSE,
           full.names = FALSE, recursive = FALSE)
dir(path = ".", pattern = NULL, all.files = FALSE,
    full.names = FALSE, recursive = FALSE)
```

**Arguments**

<code>path</code>	a character vector of full path names; the default corresponds to the working directory <code>getwd()</code> .
<code>pattern</code>	an optional <a href="#">regular expression</a> . Only file names which match the regular expression will be returned.
<code>all.files</code>	a logical value. If <code>FALSE</code> , only the names of visible files are returned. If <code>TRUE</code> , all file names will be returned.
<code>full.names</code>	a logical value. If <code>TRUE</code> , the directory path is prepended to the file names. If <code>FALSE</code> , only the file names are returned.
<code>recursive</code>	logical. Should the listing recurse into directories?

**Value**

A character vector containing the names of the files in the specified directories, or "" if there were no files. If a path does not exist or is not a directory or is unreadable it is skipped, with a warning.

The files are sorted in alphabetical order, on the full path if `full.names = TRUE`.

**Note**

File naming conventions are very platform dependent.

`recursive = TRUE` is not supported on all platforms, and may be ignored, with a warning.

**Author(s)**

Ross Ihaka, Brian Ripley

**See Also**

[file.info](#), [file.access](#) and [files](#) for many more file handling functions and [file.choose](#) for interactive selection.

**Examples**

```
list.files(R.home())
## Only files starting with a-l or r (*including* uppercase):
dir("../..", pattern = "[a-lr]", full.names=TRUE)
```

---

load

*Reload Saved Datasets*

---

**Description**

Reload datasets written with the function `save`.

**Usage**

```
load(file, envir = parent.frame())
loadURL(url, envir = parent.frame(), quiet = TRUE, ...)
```

**Arguments**

file	a connection or a character string giving the name of the file to load.
envir	the environment where the data should be loaded.
url	a character string naming a URL.
quiet, ...	additional arguments to <a href="#">download.file</a> .

**Details**

load can load R objects saved in the current or any earlier format. It can read a compressed file (see [save](#)) directly from a file or from a suitable connection (including a call to [url](#)).

loadURL is a convenience wrapper which downloads a file, loads it and deletes the downloaded copy.

Only R objects saved in the current format (used since R 1.4.0) can be read from a connection. If no input is available on a connection a warning will be given, but any input not in the current format will result in an error.

**Value**

A character vector of the names of objects created, invisibly.

**Warning**

Saved R objects are binary files, even those saved with `ascii = TRUE`, so ensure that they are transferred without conversion of end of line markers. load tries to detect this case and give an informative error message.

**See Also**

[save](#), [download.file](#).

**Examples**

```
## save all data
save(list = ls(all=TRUE), file= "all.Rdata")

## restore the saved values to the current environment
load("all.Rdata")

## restore the saved values to the user's workspace
load("all.Rdata", .GlobalEnv)

## Not run:
## print the value to see what objects were created.
print(loadURL("http://some.where.net/R/data/kprats.rda"))
```

```
# or, avoiding making a local copy,
print(load(url("http://some.where.net/R/data/kprats.rda")))
## End(Not run)
```

---

 localeconv

*Find Details of the Numerical Representations in the Current Locale*


---

## Description

Get details of the numerical representations in the current locale.

## Usage

```
Sys.localeconv()
```

## Value

A character vector with 18 named components. See your ISO C documentation for details of the meaning.

It is possible to compile R without support for locales, in which case the value will be `NULL`.

## See Also

[Sys.setlocale](#) for ways to set locales: by default R uses the C clocal for "LC\_NUMERIC" and "LC\_MONETARY".

## Examples

```
Sys.localeconv()
## The results in the default C locale are
##   decimal_point      thousands_sep      grouping      int_curr_symbol
##   "."                ""                ""                ""
##   currency_symbol  mon_decimal_point  mon_thousands_sep  mon_grouping
##   ""                ""                ""                ""
##   positive_sign     negative_sign      int_frac_digits     frac_digits
##   ""                ""                "127"             "127"
##   p_cs_precedes     p_sep_by_space     n_cs_precedes       n_sep_by_space
##   "127"             "127"             "127"             "127"
##   p_sign_posn       n_sign_posn
##   "127"             "127"

## Now try your default locale (which might be "C").
## Not run:
old <- Sys.getlocale()
Sys.setlocale(locale = "")
Sys.localeconv()
Sys.setlocale(locale = old)
## End(Not run)

## Not run: read.table("foo", dec=Sys.localeconv()["decimal_point"])
```

---

locales

*Query or Set Aspects of the Locale*

---

## Description

Get details of or set aspects of the locale for the R process.

## Usage

```
Sys.getlocale(category = "LC_ALL")
Sys.setlocale(category = "LC_ALL", locale = "")
```

## Arguments

category	character string. Must be one of "LC_ALL", "LC_COLLATE", "LC_CTYPE", "LC_MONETARY", "LC_NUMERIC" or "LC_TIME".
locale	character string. A valid locale name on the system in use. Normally "" (the default) will pick up the default locale for the system.

## Details

The locale describes aspects of the internationalization of a program. Initially most aspects of the locale of R are set to "C" (which is the default for the C language and reflects North-American usage). R does set "LC\_CTYPE" and "LC\_COLLATE", which allow the use of a different character set (typically ISO Latin 1) and alphabetic comparisons in that character set (including the use of `sort`) and "LC\_TIME" may affect the behaviour of `as.POSIXlt` and `strptime` and functions which use them (but not `date`).

R can be built with no support for locales, but it is normally available on Unix and is available on Windows.

Some systems will have other locale categories, but the six described here are those specified by POSIX.

## Value

A character string of length one describing the locale in use (after setting for `Sys.setlocale`), or an empty character string if the locale is invalid (with a warning) or NULL if locale information is unavailable.

For `category = "LC_ALL"` the details of the string are system-specific: it might be a single locale or a set of locales separated by "/" (Solaris) or ";" (Windows). For portability, it is best to query categories individually. It is guaranteed that the result of `foo <- Sys.getlocale()` can be used in `Sys.setlocale("LC_ALL", locale = foo)` on the same machine.

## Warning

Setting "LC\_NUMERIC" can produce output that R cannot then read by `scan` or `read.table` with their default arguments, which are not locale-specific.

**See Also**

[strptime](#) for uses of category = "LC\_TIME". [Sys.localeconv](#) for details of numerical representations.

**Examples**

```

Sys.getlocale()
Sys.getlocale("LC_TIME")
## Not run:
Sys.setlocale("LC_TIME", "de")      # Solaris: details are OS-dependent
Sys.setlocale("LC_TIME", "German") # Windows
## End(Not run)

Sys.setlocale("LC_COLLATE", "C") # turn off locale-specific sorting

```

---

log

---

*Logarithms and Exponentials*


---

**Description**

`log` computes natural logarithms, `log10` computes common (i.e., base 10) logarithms, and `log2` computes binary (i.e., base 2) logarithms. The general form `logb(x, base)` computes logarithms with base `base` (`log10` and `log2` are only special cases).

`log1p(x)` computes  $\log(1+x)$  accurately also for  $|x| \ll 1$  (and less accurately when  $x \approx -1$ ).

`exp` computes the exponential function.

`expm1(x)` computes  $\exp(x) - 1$  accurately also for  $|x| \ll 1$ .

**Usage**

```

log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)
exp(x)
expm1(x)
log1p(x)

```

**Arguments**

<code>x</code>	a numeric or complex vector.
<code>base</code>	positive number. The base with respect to which logarithms are computed. Defaults to <code>e=exp(1)</code> .

**Details**

`exp` and `log` are generic functions: methods can be defined for them individually or via the [Math](#) group generic.

**Value**

A vector of the same length as `x` containing the transformed values. `log(0)` gives `-Inf` (when available).

**Note**

`log` and `logb` are the same thing in R, but `logb` is preferred if `base` is specified, for S-PLUS compatibility.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (for `log`, `log10` and `exp`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (for `logb`.)

**See Also**

[Trig](#), [sqrt](#), [Arithmetic](#).

**Examples**

```
log(exp(3))
log10(1e7)# = 7

x <- 10^-(1+2*1:9)
cbind(x, log(1+x), log1p(x), exp(x)-1, expm1(x))
```

---

Logic

*Logical Operators*

---

**Description**

These operators act on logical vectors.

**Usage**

```
! x
x & y
x && y
x | y
x || y
xor(x, y)
```

**Arguments**

`x`, `y`      logical vectors, or objects which can be coerced to such or for which methods have been written.

## Details

`!` indicates logical negation (NOT).

`&` and `&&` indicate logical AND and `|` and `||` indicate logical OR. The shorter form performs elementwise comparisons in much the same way as arithmetic operators. The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined. The longer form is appropriate for programming control-flow and typically preferred in `if` clauses.

`xor` indicates elementwise exclusive OR.

Numeric and complex vectors will be coerced to logical values, with zero being false and all non-zero values being true. Raw vectors are handled without any coercion for `!`, `&` and `|`, with these operators being applied bitwise (so `!` is the 1-complement).

The operators `!`, `&` and `|` are generic functions: methods can be written for them individually or via the `Ops` group generic function.

`NA` is a valid logical object. Where a component of `x` or `y` is `NA`, the result will be `NA` if the outcome is ambiguous. In other words `NA & TRUE` evaluates to `NA`, but `NA & FALSE` evaluates to `FALSE`. See the examples below.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[TRUE](#) or [logical](#).

[any](#) and [all](#) for OR and AND on many scalar arguments.

[Syntax](#) for operator precedence.

## Examples

```
y <- 1 + (x <- rpois(50, lambda=1.5) / 4 - 1)
x[(x > 0) & (x < 1)] # all x values between 0 and 1
if (any(x == 0) || any(y == 0)) "zero encountered"

## construct truth tables :

x <- c(NA, FALSE, TRUE)
names(x) <- as.character(x)
outer(x, x, "&")## AND table
outer(x, x, "|")## OR table
```

---

`logical`*Logical Vectors*

---

### Description

Create or test for objects of type "logical", and the basic logical "constants".

### Usage

```
TRUE
FALSE
T; F
```

```
logical(length = 0)
as.logical(x, ...)
is.logical(x)
```

### Arguments

<code>length</code>	desired length.
<code>x</code>	object to be coerced or tested.
<code>...</code>	further arguments passed to or from other methods.

### Details

TRUE and FALSE are part of the R language, where T and F are global variables set to these. All four are `logical(1)` vectors.

`is.logical` is generic: you can write methods to handle specific classes of objects, see [Internal-Methods](#).

### Value

`logical` creates a logical vector of the specified length. Each element of the vector is equal to FALSE.

`as.logical` attempts to coerce its argument to be of logical type. For [factors](#), this uses the [levels](#) (labels). Like [as.vector](#) it strips attributes including names.

`is.logical` returns TRUE or FALSE depending on whether its argument is of logical type or not.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

lower.tri

*Lower and Upper Triangular Part of a Matrix*


---

**Description**

Returns a matrix of logicals the same size of a given matrix with entries TRUE in the lower or upper triangle.

**Usage**

```
lower.tri(x, diag = FALSE)
upper.tri(x, diag = FALSE)
```

**Arguments**

`x` a matrix.  
`diag` logical. Should the diagonal be included?

**See Also**

[diag](#), [matrix](#).

**Examples**

```
(m2 <- matrix(1:20, 4, 5))
lower.tri(m2)
m2[lower.tri(m2)] <- NA
m2
```

---

ls

*List Objects*


---

**Description**

`ls` and `objects` return a vector of character strings giving the names of the objects in the specified environment. When invoked with no argument at the top level prompt, `ls` shows what data sets and functions a user has defined. When invoked with no argument inside a function, `ls` returns the names of the functions local variables. This is useful in conjunction with `browser`.

**Usage**

```
ls(name, pos = -1, envir = as.environment(pos),
   all.names = FALSE, pattern)
objects(name, pos = -1, envir = as.environment(pos),
        all.names = FALSE, pattern)
```

**Arguments**

<code>name</code>	which environment to use in listing the available objects. Defaults to the <i>current</i> environment. Although called <code>name</code> for back compatibility, in fact this argument can specify the environment in any form; see the details section.
<code>pos</code>	An alternative argument to <code>name</code> for specifying the environment as a position in the search list. Mostly there for back compatibility.
<code>envir</code>	an alternative argument to <code>name</code> for specifying the environment evaluation environment. Mostly there for back compatibility.
<code>all.names</code>	a logical value. If <code>TRUE</code> , all object names are returned. If <code>FALSE</code> , names which begin with a <code>'.'</code> are omitted.
<code>pattern</code>	an optional <a href="#">regular expression</a> . Only names matching <code>pattern</code> are returned.

**Details**

The `name` argument can specify the environment from which object names are taken in one of several forms: as an integer (the position in the [search](#) list); as the character string name of an element in the search list; or as an explicit [environment](#) (including using `sys.frame` to access the currently active function calls). By default, the environment of the call to `ls` or `objects` is used. The `pos` and `envir` arguments are an alternative way to specify an environment, but are primarily there for back compatibility.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`ls.str` for a long listing based on `str`. `apropos` (or `find`) for finding objects in the whole search path; `grep` for more details on “regular expressions”; `class`, `methods`, etc., for object-oriented programming.

**Examples**

```
.Ob <- 1
ls(pat="O")
ls(pat="O", all = TRUE)    # also shows ".[foo]"

# shows an empty list because inside myfunc no variables are defined
myfunc <- function() {ls()}
myfunc()

# define a local variable inside myfunc
myfunc <- function() {y <- 1; ls()}
myfunc()    # shows "y"
```

---

`make.names`*Make Syntactically Valid Names*

---

**Description**

Make syntactically valid names out of character vectors.

**Usage**

```
make.names(names, unique = FALSE, allow_ = TRUE)
```

**Arguments**

<code>names</code>	character vector to be coerced to syntactically valid names. This is coerced to character if necessary.
<code>unique</code>	logical; if TRUE, the resulting elements are unique. This may be desired for, e.g., column names.
<code>allow_</code>	logical. For compatibility with R prior to 1.9.0.

**Details**

A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as ".2way" are not valid, and neither are the reserved words.

The character "X" is prepended if necessary. All invalid characters are translated to ".". A missing value is translated to "NA". Names which match R keywords have a dot appended to them. Duplicated values are altered by [make.unique](#).

**Value**

A character vector of same length as `names` with each changed to a syntactically valid name.

**Note**

Prior to R version 1.9.0, underscores were not valid in variable names, and code that relies on them being converted to dots will no longer work. Use `allow_ = FALSE` for back-compatibility.

`allow_ = FALSE` is also useful when creating names for export to applications which do not allow underline in names (for example, S-PLUS and some DBMSs).

**See Also**

[make.unique](#), [names](#), [character](#), [data.frame](#).

**Examples**

```

make.names(c("a and b", "a-and-b"), unique=TRUE)
# "a.and.b" "a.and.b.1"
make.names(c("a and b", "a_and_b"), unique=TRUE)
# "a.and.b" "a_and_b"
make.names(c("a and b", "a_and_b"), unique=TRUE, allow_=FALSE)
# "a.and.b" "a.and.b.1"

state.name[make.names(state.name) != state.name] # those 10 with a space

```

---

make.unique	<i>Make Character Strings Unique</i>
-------------	--------------------------------------

---

**Description**

Makes the elements of a character vector unique by appending sequence numbers to duplicates.

**Usage**

```
make.unique(names, sep = ".")
```

**Arguments**

names	a character vector
sep	a character string used to separate a duplicate name from its sequence number.

**Details**

The algorithm used by `make.unique` has the property that `make.unique(c(A, B)) == make.unique(c(make.unique(A), B))`.

In other words, you can append one string at a time to a vector, making it unique each time, and get the same result as applying `make.unique` to all of the strings at once.

If character vector `A` is already unique, then `make.unique(c(A, B))` preserves `A`.

**Value**

A character vector of same length as `names` with duplicates changed.

**Author(s)**

Thomas P Minka

**See Also**

[make.names](#)

**Examples**

```

make.unique(c("a", "a", "a"))
make.unique(c(make.unique(c("a", "a")), "a"))

make.unique(c("a", "a", "a.2", "a"))
make.unique(c(make.unique(c("a", "a")), "a.2", "a"))

rbind(data.frame(x=1), data.frame(x=2), data.frame(x=3))
rbind(rbind(data.frame(x=1), data.frame(x=2)), data.frame(x=3))

```

---

manglePackageName *Mangle the Package Name*

---

**Description**

This function takes the package name and the package version number and pastes them together with a separating underscore.

**Usage**

```
manglePackageName(pkgName, pkgVersion)
```

**Arguments**

pkgName        The package name, as a character string.  
pkgVersion     The package version, as a character string.

**Value**

A character string with the two inputs pasted together.

**Examples**

```
manglePackageName("foo", "1.2.3")
```

---

mapply *Apply a function to multiple list or vector arguments*

---

**Description**

A multivariate version of [sapply](#). `mapply` applies FUN to the first elements of each ... argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.

**Usage**

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

**Arguments**

FUN	Function to apply
...	Arguments to vectorise over (list or vector)
MoreArgs	A list of other arguments to FUN
SIMPLIFY	Attempt to reduce the result to a vector or matrix?
USE.NAMES	If the first ... argument is character and the result doesn't already have names, use it as the names

**Value**

A list, vector, or matrix.

**See Also**

[sapply](#), [outer](#)

**Examples**

```
mapply(rep, 1:4, 4:1)
mapply(rep, times=1:4, x=4:1)
mapply(rep, times=1:4, MoreArgs=list(x=42))
```

---

margin.table	<i>Compute table margin</i>
--------------	-----------------------------

---

**Description**

For a contingency table in array form, compute the sum of table entries for a given index.

**Usage**

```
margin.table(x, margin=NULL)
```

**Arguments**

x	an array
margin	index number (1 for rows, etc.)

**Details**

This is really just `apply(x, margin, sum)` packaged up for newbies, except that if `margin` has length zero you get `sum(x)`.

**Value**

The relevant marginal table. The class of `x` is copied to the output table, except in the summation case.

**Author(s)**

Peter Dalgaard

**Examples**

```
m <- matrix(1:4,2)
margin.table(m,1)
margin.table(m,2)
```

---

mat.or.vec

*Create a Matrix or a Vector*

---

**Description**

`mat.or.vec` creates an `nr` by `nc` zero matrix if `nc` is greater than 1, and a zero vector of length `nr` if `nc` equals 1.

**Usage**

```
mat.or.vec(nr, nc)
```

**Arguments**

`nr`, `nc`      numbers of rows and columns.

**Examples**

```
mat.or.vec(3, 1)
mat.or.vec(3, 2)
```

---

match	<i>Value Matching</i>
-------	-----------------------

---

**Description**

`match` returns a vector of the positions of (first) matches of its first argument in its second.

`%in%` is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.

**Usage**

```
match(x, table, nomatch = NA, incomparables = FALSE)
```

```
x %in% table
```

**Arguments**

<code>x</code>	the values to be matched.
<code>table</code>	the values to be matched against.
<code>nomatch</code>	the value to be returned in the case when no match is found. Note that it is coerced to integer.
<code>incomparables</code>	a vector of values that cannot be matched. Any value in <code>x</code> matching a value in this vector is assigned the <code>nomatch</code> value. Currently, <code>FALSE</code> is the only possible value, meaning that all values can be matched.

**Details**

`%in%` is currently defined as

```
"%in%" <- function(x, table) match(x, table, nomatch = 0) > 0
```

Factors are converted to character vectors, and then `x` and `table` are coerced to a common type (the later of the two types in R's ordering, logical < integer < numeric < complex < character) before matching.

**Value**

In both cases, a vector of the same length as `x`.

`match`: An integer vector giving the position in `table` of the first match if there is a match, otherwise `nomatch`.

If `x[i]` is found to equal `table[j]` then the value returned in the *i*-th position of the return value is *j*, for the smallest possible *j*. If no match is found, the value is `nomatch`.

`%in%`: A logical vector, indicating if a match was located for each element of `x`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[pmatch](#) and [charmatch](#) for (*partial*) string matching, [match.arg](#), etc for function argument matching.

[is.element](#) for an S-compatible equivalent of `%in%`.

## Examples

```
## The intersection of two sets :
intersect <- function(x, y) y[match(x, y, nomatch = 0)]
intersect(1:10,7:20)

1:10 %in% c(1,3,5,9)
sstr <- c("c", "ab", "B", "bba", "c", "@", "bla", "a", "Ba", "%")
sstr[sstr %in% c(letters,LETTERS)]

"%w/o%" <- function(x,y) x[!x %in% y] #-- x without y
(1:10) %w/o% c(3,7,12)
```

---

match.arg

*Argument Verification Using Partial Matching*

---

## Description

`match.arg` matches `arg` against a table of candidate values as specified by `choices`.

## Usage

```
match.arg(arg, choices)
```

## Arguments

<code>arg</code>	a character string
<code>choices</code>	a character vector of candidate values

## Details

In the one-argument form `match.arg(arg)`, the `choices` are obtained from a default setting for the formal argument `arg` of the function from which `match.arg` was called.

Matching is done using [pmatch](#), so `arg` may be abbreviated.

**Value**

The unabbreviated version of the unique partial match if there is one; otherwise, an error is signalled.

**See Also**

[pmatch](#), [match.fun](#), [match.call](#).

**Examples**

```
require(stats)
## Extends the example for 'switch'
center <- function(x, type = c("mean", "median", "trimmed")) {
  type <- match.arg(type)
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
center(x, "t")      # Works
center(x, "med")    # Works
try(center(x, "m")) # Error
```

---

match.call

*Argument Matching*

---

**Description**

`match.call` returns a call in which all of the arguments are specified by their names. The most common use is to get the call of the current function, with all arguments named.

**Usage**

```
match.call(definition = NULL, call = sys.call(sys.parent()),
           expand.dots = TRUE)
```

**Arguments**

<code>definition</code>	a function, by default the function from which <code>match.call</code> is called.
<code>call</code>	an unevaluated call to the function specified by <code>definition</code> , as generated by <a href="#">call</a> .
<code>expand.dots</code>	logical. Should arguments matching <code>...</code> in the call be included or left as a <code>...</code> argument?

**Value**

An object of class `call`.

## References

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

## See Also

[call](#), [pmatch](#), [match.arg](#), [match.fun](#).

## Examples

```
match.call(get, call("get", "abc", i = FALSE, p = 3))
## -> get(x = "abc", pos = 3, inherits = FALSE)
fun <- function(x, lower = 0, upper = 1) {
  structure((x - lower) / (upper - lower), CALL = match.call())
}
fun(4 * atan(1), u = pi)
```

---

match.fun

*Function Verification for “Function Variables”*

---

## Description

When called inside functions that take a function as argument, extract the desired function object while avoiding undesired matching to objects of other types.

## Usage

```
match.fun(FUN, descend = TRUE)
```

## Arguments

FUN	item to match as function.
descend	logical; control whether to search past non-function objects.

## Details

`match.fun` is not intended to be used at the top level since it will perform matching in the *parent* of the caller.

If FUN is a function, it is returned. If it is a symbol or a character vector of length one, it will be looked up using `get` in the environment of the parent of the caller. If it is of any other mode, it is attempted first to get the argument to the caller as a symbol (using `substitute` twice), and if that fails, an error is declared.

If `descend = TRUE`, `match.fun` will look past non-function objects with the given name; otherwise if FUN points to a non-function object then an error is generated.

This is now used in base functions such as [apply](#), [lapply](#), [outer](#), and [sweep](#).

**Value**

A function matching FUN or an error is generated.

**Bugs**

The `descend` argument is a bit of misnomer and probably not actually needed by anything. It may go away in the future.

It is impossible to fully foolproof this. If one attaches a list or data frame containing a character object with the same name of a system function, it will be used.

**Author(s)**

Peter Dalgaard and Robert Gentleman, based on an earlier version by Jonathan Rougier.

**See Also**

[match.arg](#), [get](#)

**Examples**

```
# Same as get("*"):
match.fun("*")
# Overwrite outer with a vector
outer <- 1:5
## Not run:
match.fun(outer, descend = FALSE) #-> Error: not a function
## End(Not run)
match.fun(outer) # finds it anyway
is.function(match.fun("outer")) # as well
```

---

matmult

*Matrix Multiplication*

---

**Description**

Multiplies two matrices, if they are conformable. If one argument is a vector, it will be coerced to either a row or column matrix to make the two arguments conformable. If both are vectors it will return the inner product.

**Usage**

```
a %*% b
```

**Arguments**

a, b            numeric or complex matrices or vectors.

**Value**

The matrix product. Use [drop](#) to get rid of dimensions which have only one level.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[matrix](#), [Arithmetic](#), [diag](#).

**Examples**

```
x <- 1:4
(z <- x %*% x)      # scalar ("inner") product (1 x 1 matrix)
drop(z)            # as scalar

y <- diag(x)
z <- matrix(1:12, ncol = 3, nrow = 4)
y %*% z
y %*% x
x %*% z
```

---

matrix

*Matrices*

---

**Description**

`matrix` creates a matrix from the given set of values.

`as.matrix` attempts to turn its argument into a matrix.

`is.matrix` tests if its argument is a (strict) matrix. It is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

**Usage**

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
as.matrix(x)
is.matrix(x)
```

**Arguments**

<code>data</code>	an optional data vector.
<code>nrow</code>	the desired number of rows
<code>ncol</code>	the desired number of columns
<code>byrow</code>	logical. If <code>FALSE</code> (the default) the matrix is filled by columns, otherwise the matrix is filled by rows.

`dimnames`      A `dimnames` attribute for the matrix: a list of length 2.  
`x`                an R object.

### Details

If either of `nrow` or `ncol` is not given, an attempt is made to infer it from the length of `data` and the other parameter.

If there are too few elements in `data` to fill the array, then the elements in `data` are recycled. If `data` has length zero, NA of an appropriate type is used for atomic vectors (0 for raw vectors) and NULL for lists.

`is.matrix` returns TRUE if `x` is a matrix (i.e., it is *not* a `data.frame` and has a `dim` attribute of length 2) and FALSE otherwise.

`as.matrix` is a generic function. The method for data frames will convert any non-numeric/complex column into a character vector using `format` and so return a character matrix, except that all-logical data frames will be coerced to a logical matrix.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`data.matrix`, which attempts to convert to a numeric matrix.

### Examples

```
is.matrix(as.matrix(1:10))
!is.matrix(warpbreaks)# data.frame, NOT matrix!
warpbreaks[1:10,]
as.matrix(warpbreaks[1:10,]) #using as.matrix.data.frame(.) method
```

---

maxCol

*Find Maximum Position in Matrix*

---

### Description

Find the maximum position for each row of a matrix, breaking ties at random.

### Usage

```
max.col(m)
```

### Arguments

`m`                numerical matrix

**Details**

Ties are broken at random. The determination of “tie” assumes that the entries are probabilities: there is a relative tolerance of  $10^{-5}$ , relative to the largest entry in the row.

**Value**

index of a maximal value for each row, an integer vector of length `nrow(m)`.

**References**

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer (4th ed).

**See Also**

`which.max` for vectors.

**Examples**

```
table(mc <- max.col(swiss))# mostly "1" and "5", 5 x "2" and once "4"
swiss[unique(print(mr <- max.col(t(swiss)))) , ] # 3 33 45 45 33 6
```

---

mean

*Arithmetic Mean*

---

**Description**

Generic function for the (trimmed) arithmetic mean.

**Usage**

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

**Arguments**

<code>x</code>	An R object. Currently there are methods for numeric data frames, numeric vectors and dates. A complex vector is allowed for <code>trim = 0</code> , only.
<code>trim</code>	the fraction (0 to 0.5) of observations to be trimmed from each end of <code>x</code> before the mean is computed.
<code>na.rm</code>	a logical value indicating whether NA values should be stripped before the computation proceeds.
<code>...</code>	further arguments passed to or from other methods.

**Value**

For a data frame, a named vector with the appropriate method being applied column by column.

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[weighted.mean](#), [mean.POSIXct](#)

**Examples**

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))

mean(USArrests, trim = 0.2)
```

---

Memory

*Memory Available for Data Storage*

---

**Description**

Use command line options to control the memory available for R.

**Usage**

```
R --min-vsize=v1 --max-vsize=vu --min-nsz=nl --max-nsz=nu
mem.limits(nsize = NA, vsize = NA)
```

**Arguments**

```
v1, vu, vsize
      Heap memory in bytes.
nl, nu, nsize
      Number of cons cells.
```

## Details

R has a variable-sized workspace (from version 1.2.0). There is now much less need to set memory options than previously, and most users will never need to set these. They are provided both as a way to control the overall memory usage (which can also be done by operating-system facilities such as `limit` on Unix), and since setting larger values of the minima will make R slightly more efficient on large tasks.

To understand the options, one needs to know that R maintains separate areas for fixed and variable sized objects. The first of these is allocated as an array of “*cons cells*” (Lisp programmers will know what they are, others may think of them as the building blocks of the language itself, parse trees, etc.), and the second are thrown on a “*heap*” of “*Vcells*” of 8 bytes each. Effectively, the input `v` is rounded up to the nearest multiple of 8.

Each cons cell occupies 28 bytes on a 32-bit machine, (usually) 56 bytes on a 64-bit machine.

The ‘`--*nsize`’ options can be used to specify the number of cons cells and the ‘`--*vsize`’ options specify the size of the vector heap in bytes. Both options must be integers or integers followed by G, M, K, or k meaning Giga ( $2^{30} = 1073741824$ ) Mega ( $2^{20} = 1048576$ ), (computer) Kilo ( $2^{10} = 1024$ ), or regular kilo (1000).

The ‘`--min-*`’ options set the minimal sizes for the number of cons cells and for the vector heap. These values are also the initial values, but thereafter R will grow or shrink the areas depending on usage, but never exceeding the limits set by the ‘`--max-*`’ options nor decreasing below the initial values.

The default values are currently minima of 350k cons cells, 6Mb of vector heap and no maxima (other than machine resources). The maxima can be changed during an R session by calling `mem.limits`. (If this is called with the default values, it reports the current settings.)

You can find out the current memory consumption (the heap and cons cells used as numbers and megabytes) by typing `gc()` at the R prompt. Note that following `gcinfo(TRUE)`, automatic garbage collection always prints memory use statistics. Maxima will never be reduced below the current values for triggering garbage collection, and attempts to do so will be silently ignored.

When using `read.table`, the memory requirements are in fact higher than anticipated, because the file is first read in as one long string which is then split again. Use `scan` if possible in case you run out of memory when reading in a large table.

## Value

`mem.limits()` returns an integer vector giving the current settings of the maxima, possibly NA.

## Note

For backwards compatibility, options ‘`--nsize`’ and ‘`--vsize`’ are equivalent to ‘`--min-nsize`’ and ‘`--min-vsize`’.

## See Also

[Memory-limits](#) for the design limitations.

`gc` for information on the garbage collector, `object.size(a)` for the (approximate) size of R object `a`. [memory.profile](#) for profiling the usage of cons cells.

## Examples

```
# Start R with 10MB of heap memory and 500k cons cells, limit to
# 100Mb and 1M cells
## Not run:
## Unix
R --min-vsize=10M --max-vsize=100M --min-nsize=500k --max-nsize=1M
## End(Not run)
```

---

Memory-limits

*Memory Limits in R*

---

## Description

R holds objects it is using in memory. This help file documents the current design limitations on large objects: these differ between 32-bit and 64-bit builds of R.

## Details

R holds all objects in memory, and there are limits based on the amount of memory that can be used by all objects:

- There may be limits on the size of the heap and the number of cons cells allowed – see [Memory](#) – but these are usually not imposed.
- There is a limit on the address space of a single process such as the R executable. This is system-specific, but for 32-bit OSes imposes a limit of no more than 4Gb. It is often rather less.
- The environment may impose limitations on the resources available to a single process – see the OS/shell's help on commands such as `limit` or `ulimit`.

Error messages beginning `cannot allocate vector of size` indicate a failure to obtain memory, either because the size exceeded the address-space limit for a process or, more likely, because the system was unable to provide the memory.

There are also limits on individual objects. On all versions of R, the maximum length (number of elements) of a vector is  $2^{31} - 1 \approx 2 \cdot 10^9$ , as lengths are stored as signed integers. In addition, the storage space cannot exceed the address limit, and if you try to exceed that limit, the error message begins `cannot allocate vector of length`. The number of characters in a character string is in theory only limited by the address space.

## See Also

[object.size\(a\)](#) for the (approximate) size of R object `a`.

---

memory.profile	<i>Profile the Usage of Cons Cells</i>
----------------	--

---

**Description**

Lists the usage of the cons cells by SEXPREC type.

**Usage**

```
memory.profile()
```

**Details**

The current types and their uses are listed in the include file ‘Rinternals.h’. There will be blanks in the list corresponding to types that are no longer in use (types 11 and 12 at the time of writing). Also FUNSXP is not included.

**Value**

A vector of counts, named by the types.

**See Also**

[gc](#) for the overall usage of cons cells.

**Examples**

```
memory.profile()
```

---

merge	<i>Merge Two Data Frames</i>
-------	------------------------------

---

**Description**

Merge two data frames by common columns or row names, or do other versions of database “join” operations.

**Usage**

```
merge(x, y, ...)

## Default S3 method:
merge(x, y, ...)

## S3 method for class 'data.frame':
merge(x, y, by = intersect(names(x), names(y)),
      by.x = by, by.y = by, all = FALSE, all.x = all, all.y = all,
      sort = TRUE, suffixes = c(".x", ".y"), ...)
```

**Arguments**

<code>x, y</code>	data frames, or objects to be coerced to one
<code>by, by.x, by.y</code>	specifications of the common columns. See Details.
<code>all</code>	logical; <code>all=L</code> is shorthand for <code>all.x=L</code> and <code>all.y=L</code> .
<code>all.x</code>	logical; if TRUE, then extra rows will be added to the output, one for each row in <code>x</code> that has no matching row in <code>y</code> . These rows will have NAs in those columns that are usually filled with values from <code>y</code> . The default is FALSE, so that only rows with data from both <code>x</code> and <code>y</code> are included in the output.
<code>all.y</code>	logical; analogous to <code>all.x</code> above.
<code>sort</code>	logical. Should the results be sorted on the <code>by</code> columns?
<code>suffixes</code>	character(2) specifying the suffixes to be used for making non- <code>by</code> names() unique.
<code>...</code>	arguments to be passed to or from methods.

**Details**

By default the data frames are merged on the columns with names they both have, but separate specifications of the columns can be given by `by.x` and `by.y`. Columns can be specified by name, number or by a logical vector: the name "row.names" or the number 0 specifies the row names. The rows in the two data frames that match on the specified columns are extracted, and joined together. If there is more than one match, all possible matches contribute one row each.

If the `by.*` vectors are of length 0, the result, `r`, is the "Cartesian product" of `x` and `y`, i.e.,  $\text{dim}(r) = c(\text{nrow}(x) * \text{nrow}(y), \text{ncol}(x) + \text{ncol}(y))$ .

If `all.x` is true, all the non matching cases of `x` are appended to the result as well, with NA filled in the corresponding columns of `y`; analogously for `all.y`.

If the remaining columns in the data frames have any common names, these have `suffixes` ("`.x`" and "`.y`" by default) appended to make the names of the result unique.

**Value**

A data frame. The rows are by default lexicographically sorted on the common columns, but are otherwise in the order in which they occurred in `y`. The columns are the common columns followed by the remaining columns in `x` and then those in `y`. If the matching involved row names, an extra column `Row.names` is added at the left, and in all cases the result has no special row names.

**See Also**

[data.frame](#), [by](#), [cbind](#)

**Examples**

```
authors <- data.frame(
  surname = c("Tukey", "Venables", "Tierney", "Ripley", "McNeil"),
  nationality = c("US", "Australia", "US", "UK", "Australia"),
  deceased = c("yes", rep("no", 4)))
```

```

books <- data.frame(
  name = c("Tukey", "Venables", "Tierney",
           "Ripley", "Ripley", "McNeil", "R Core"),
  title = c("Exploratory Data Analysis",
            "Modern Applied Statistics ...",
            "LISP-STAT",
            "Spatial Statistics", "Stochastic Simulation",
            "Interactive Data Analysis",
            "An Introduction to R"),
  other.author = c(NA, "Ripley", NA, NA, NA, NA,
                  "Venables & Smith"))

(m1 <- merge(authors, books, by.x = "surname", by.y = "name"))
(m2 <- merge(books, authors, by.x = "name", by.y = "surname"))
stopifnot(as.character(m1[,1]) == as.character(m2[,1]),
          all.equal(m1[, -1], m2[, -1][ names(m1)[-1] ]),
          dim(merge(m1, m2, by = integer(0))) == c(36, 10))

## "R core" is missing from authors and appears only here :
merge(authors, books, by.x = "surname", by.y = "name", all = TRUE)

```

---

missing

*Does a Formal Argument have a Value?*

---

## Description

`missing` can be used to test whether a value was specified as an argument to a function.

## Usage

```
missing(x)
```

## Arguments

`x` a formal argument.

## Details

`missing(x)` is only reliable if `x` has not been altered since entering the function: in particular it will *always* be false after `x <- match.arg(x)`.

The example shows how a plotting function can be written to work with either a pair of vectors giving `x` and `y` coordinates of points to be plotted or a single vector giving `y` values to be plotted against their indexes.

Currently `missing` can only be used in the immediate body of the function that defines the argument, not in the body of a nested function or a `local` call. This may change in the future.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

**See Also**

[substitute](#) for argument expression; [NA](#) for “missing values” in data.

**Examples**

```
myplot <- function(x,y) {
  if(missing(y)) {
    y <- x
    x <- 1:length(y)
  }
  plot(x,y)
}
```

---

 mode

*The (Storage) Mode of an Object*


---

**Description**

Get or set the type or storage mode of an object.

**Usage**

```
mode(x)
mode(x) <- value
storage.mode(x)
storage.mode(x) <- value
```

**Arguments**

`x` any R object.  
`value` a character string giving the desired (storage) mode of the object.

**Details**

Both `mode` and `storage.mode` return a character string giving the (storage) mode of the object — often the same — both relying on the output of `typeof(x)`, see the example below.

The two assignment versions are currently identical. Both `mode(x) <- newmode` and `storage.mode(x) <- newmode` change the mode or `storage.mode` of object `x` to `newmode`.

As storage mode "single" is only a pseudo-mode in R, it will not be reported by `mode` or `storage.mode`: use `attr(object, "Csingle")` to examine this. However, the assignment versions can be used to set the mode to "single", which sets the real mode to "double" and the "Csingle" attribute to TRUE. Setting any other mode will remove this attribute.

Note (in the examples below) that some `calls` have mode "( " which is S compatible.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

`typeof` for the R-internal "mode", `attributes`.

## Examples

```
sapply(options(),mode)

cex3 <- c("NULL","1","1:1","1i","list(1)","data.frame(x=1)", "pairlist(pi)",
  "c", "lm", "formals(lm)[[1]]", "formals(lm)[[2]]",
  "y~x","expression(1)[[1]]", "y~x)[[1]]", "expression(x <- pi)[[1]][[1]]")
lex3 <- sapply(cex3, function(x) eval(parse(text=x)))
mex3 <- t(sapply(lex3, function(x) c(typeof(x), storage.mode(x), mode(x))))
dimnames(mex3) <- list(cex3, c("typeof(.)", "storage.mode(.)", "mode(.)))
mex3

## This also makes a local copy of 'pi':
storage.mode(pi) <- "complex"
storage.mode(pi)
rm(pi)
```

---

NA

*Not Available / "Missing" Values*

---

## Description

NA is a logical constant of length 1 which contains a missing value indicator. NA can be freely coerced to any other vector type.

The generic function `is.na` indicates which elements are missing.

The generic function `is.na<-` sets elements to NA.

**Usage**

```

NA
is.na(x)
## S3 method for class 'data.frame':
is.na(x)

is.na(x) <- value

```

**Arguments**

`x` an R object to be tested.

`value` a suitable index vector for use with `x`.

**Details**

The NA of character type is as from R 1.5.0 distinct from the string "NA". Programmers who need to specify an explicit string NA should use `as.character(NA)` rather than "NA", or set elements to NA using `is.na<-`.

`is.na(x)` works elementwise when `x` is a [list](#). The method dispatching is C-internal, rather than via [UseMethod](#).

Function `is.na<-` may provide a safer way to set missingness. It behaves differently for factors, for example.

**Value**

The default method for `is.na` returns a logical vector of the same “form” as its argument `x`, containing TRUE for those elements marked NA or NaN (!) and FALSE otherwise. `dim`, `dimnames` and `names` attributes are preserved.

The method `is.na.data.frame` returns a logical matrix with the same dimensions as the data frame, and with `dimnames` taken from the row and column names of the data frame.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

**See Also**

[NaN](#), [is.nan](#), etc., and the utility function [complete.cases](#).

[na.action](#), [na.omit](#), [na.fail](#) on how methods can be tuned to deal with missing values.

**Examples**

```

is.na(c(1, NA))      #> FALSE TRUE
is.na(paste(c(1, NA))) #> FALSE FALSE

```

---

name	<i>Variable Names or Symbols, respectively</i>
------	--

---

### Description

`as.symbol` coerces its argument to be a *symbol*, or equivalently, a *name*. The argument must be of mode "character". `as.name` is an alias for `as.symbol`.

`is.symbol` (and `is.name` equivalently) returns TRUE or FALSE depending on whether its argument is a symbol (i.e., name) or not.

### Usage

```
as.symbol(x)
is.symbol(y)
```

```
as.name(x)
is.name(y)
```

### Arguments

`x`, `y`            objects to be coerced or tested.

### Details

`is.symbol` is generic: you can write methods to handle specific classes of objects, see [Internal-Methods](#).

### Note

The term “symbol” is from the LISP background of R, whereas “name” has been the standard S term for this.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[call](#), [is.language](#). For the internal object mode, [typeof](#).

### Examples

```
an <- as.name("arrg")
is.name(an) # TRUE
mode(an)    # name
typeof(an) # symbol
```

---

names

*The Names Attribute of an Object*

---

## Description

Functions to get or set the names of an object.

## Usage

```
names(x)
names(x) <- value
```

## Arguments

`x` an R object.  
`value` a character vector of up to the same length as `x`, or `NULL`.

## Details

`names` is a generic accessor function, and `names<-` is a generic replacement function. The default methods get and set the "names" attribute of a vector or list.

If `value` is shorter than `x`, it is extended by character NAs to the length of `x`.

It is possible to update just part of the names attribute via the general rules: see the examples. This works because the expression there is evaluated as `z <- "names<-"(z, "[<-"(names(z), 3, "c2"))`.

## Value

For `names`, `NULL` or a character vector of the same length as `x`.

For `names<-`, the updated object. (Note that the value of `names(x) <- value` is that of the assignment, `value`, not the return value from the left-hand side.)

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## Examples

```
# print the names attribute of the islands data set
names(islands)

# remove the names attribute
names(islands) <- NULL
islands
rm(islands) # remove the copy made
```

```

z <- list(a=1, b="c", c=1:3)
names(z)
# change just the name of the third element.
names(z)[3] <- "c2"
z

z <- 1:3
names(z)
## assign just one name
names(z)[2] <- "b"
z

```

---

nargs

*The Number of Arguments to a Function*


---

### Description

When used inside a function body, `nargs` returns the number of arguments supplied to that function, *including* positional arguments left blank.

### Usage

```
nargs()
```

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[args](#), [formals](#) and [sys.call](#).

### Examples

```

tst <- function(a, b = 3, ...) {nargs()}
tst() # 0
tst(clicketyclack) # 1 (even non-existing)
tst(c1, a2, rr3) # 3

foo <- function(x, y, z, w) {
  cat("call was", deparse(match.call()), "\n")
  nargs()
}
foo() # 0
foo(, , 3) # 3
foo(z=3) # 1, even though this is the same call

nargs()# not really meaningful

```

---

`nchar`*Count the Number of Characters*

---

**Description**

`nchar` takes a character vector as an argument and returns a vector whose elements contain the number of characters in the corresponding element of `x`.

**Usage**

```
nchar(x)
```

**Arguments**

`x` character vector, or a vector to be coerced to a character vector.

**Details**

The internal equivalent of `as.character` is performed on `x`. If you want to operate on non-vector objects passing them through `deparse` first will be required.

**Value**

The number of characters as the string will be printed (integer 2 for a missing string).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`strwidth` giving width of strings for plotting; `paste`, `substr`, `strsplit`

**Examples**

```
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
nchar(x)
# 5 6 6 1 15

nchar(deparse(mean))
# 23 1 16 45 11 64 2 17 50 43 2 17 1
```

---

nlevels	<i>The Number of Levels of a Factor</i>
---------	---

---

**Description**

Return the number of levels which its argument has.

**Usage**

```
nlevels(x)
```

**Arguments**

`x` an object, usually a factor.

**Details**

If the argument is not a `factor`, NA is returned.

The actual factor levels (if they exist) can be obtained with the `levels` function.

**Examples**

```
nlevels(gl(3,7)) # = 3
```

---

noquote	<i>Class for “no quote” Printing of Character Strings</i>
---------	---

---

**Description**

Print character strings without quotes.

**Usage**

```
noquote(obj)

## S3 method for class 'noquote':
print(x, ...)

## S3 method for class 'noquote':
c(..., recursive = FALSE)
```

**Arguments**

`obj` any R object, typically a vector of `character` strings.  
`x` an object of class "noquote".  
`...` further options passed to next methods, such as `print`.  
`recursive` for compatibility with the generic `c` function.

## Details

`noquote` returns its argument as an object of class `"noquote"`. There is a method for `c()` and subscript method (`"[.noquote"`) which ensures that the class is not lost by subsetting. The print method (`print.noquote`) prints character strings *without* quotes (`"..."`).

These functions exist both as utilities and as an example of using (S3) `class` and object orientation.

## Author(s)

Martin Maechler [⟨maechler@stat.math.ethz.ch⟩](mailto:maechler@stat.math.ethz.ch)

## See Also

[methods](#), [class](#), [print](#).

## Examples

```
letters
nql <- noquote(letters)
nql
nql[1:4] <- "oh"
nql[1:12]

cmp.logical <- function(log.v)
{
  ## Purpose: compact printing of logicals
  log.v <- as.logical(log.v)
  noquote(if(length(log.v)==0)"()" else c(".", "|")[1+log.v])
}
cmp.logical(runif(20) > 0.8)
```

## Description

In order to pinpoint missing functionality, the R core team uses these functions for missing R functions and not yet used arguments of existing R functions (which are typically there for compatibility purposes).

You are very welcome to contribute your code ...

## Usage

```
.NotYetImplemented()
.NotYetUsed(arg, error = TRUE)
```

**Arguments**

`arg` an argument of a function that is not yet used.  
`error` a logical. If TRUE, an error is signalled; if FALSE, only a warning is given.

**See Also**

the contrary, [Deprecated](#) and [Defunct](#) for outdated code.

**Examples**

```
require(graphics)
require(stats)
plot.mlm          # to see how the "NotYetImplemented"
                  # reference is made automagically
try(plot.mlm())

barplot(1:5, inside = TRUE) # 'inside' is not yet used
```

---

nrow

*The Number of Rows/Columns of an Array*


---

**Description**

`nrow` and `ncol` return the number of rows or columns present in `x`. `NCOL` and `NROW` do the same treating a vector as 1-column matrix.

**Usage**

```
nrow(x)
ncol(x)
NCOL(x)
NROW(x)
```

**Arguments**

`x` a vector, array or data frame

**Value**

an [integer](#) of length 1 or [NULL](#).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (`ncol` and `nrow`.)

**See Also**

`dim` which returns *all* dimensions; `array`, `matrix`.

**Examples**

```
ma <- matrix(1:12, 3, 4)
nrow(ma) # 3
ncol(ma) # 4

ncol(array(1:24, dim = 2:4)) # 3, the second dimension
NCOL(1:12) # 1
NROW(1:12) # 12
```

---

`ns-dblcolon`*Double Colon and Triple Colon Operators*

---

**Description**

Accessing exported and internal variables in a name space.

**Usage**

```
pkg::name
pkg>:::name
```

**Arguments**

<code>pkg</code>	package name symbol or literal character string.
<code>name</code>	variable name symbol or literal character string.

**Details**

The expression `pkg::name` returns the value of the exported variable `name` in package `pkg` if the package has a name space. The expression `pkg>:::name` returns the value of the internal variable `name` in package `pkg` if the package has a name space. The package will be loaded if it was not loaded already before the call. Assignment into name spaces is not supported.

**Examples**

```
base::log
base::"+"
```

---

 ns-hooks

*Hooks for Name Space events*


---

## Description

Packages with name spaces can supply functions to be called when loaded, attached or unloaded.

## Usage

```
.onLoad(libname, pkgname)
.onAttach(libname, pkgname)
.onUnload(libpath)
```

## Arguments

libname	a character string giving the library directory where the package defining the namespace was found.
pkgname	a character string giving the name of the package, including the version number if the package was installed with <code>--with-package-versions</code> .
libpath	a character string giving the complete path to the package.

## Details

These functions apply only to packages with name spaces.

After loading, `loadNamespace` looks for a hook function named `.onLoad` and runs it before sealing the namespace and processing exports.

If a name space is unloaded (via `unloadNamespace`), a hook function `.onUnload` is run before final unloading.

Note that the code in `.onLoad` and `.onUnload` is run without the package being on the search path, but (unless circumvented) lexical scope will make objects in the namespace and its imports visible. (Do not use the double colon operator in `.onLoad` as exports have not been processed at the point it is run.)

When the package is attached (via `library`), the hook function `.onAttach` is called after the exported functions are attached. This is less likely to be useful than `.onLoad`, which should be seen as the analogue of `.First.lib` (which is only used for packages without a name space).

`.onLoad`, `.onUnload` and `.onAttach` are looked for as internal variables in the name space and should not be exported.

If a function `.Last.lib` is visible in the package, it will be called when the package is detached: this does need to be exported.

Anything needed for the functioning of the name space should be handled at load/unload times by the `.onLoad` and `.onUnload` hooks. For example, shared libraries can be loaded (unless done by a `useDynlib` directive in the `NAMESPACE` file) and initialized in `.onLoad` and unloaded in `.onUnload`. Use `.onAttach` only for actions that are needed only when the package becomes visible to the user, for example a start-up message.

If a package was installed with `--with-package-versions`, the `pkgname` supplied will be something like `tree_1.0-16`.

### See Also

[setHook](#) shows how users can set hooks on the same events.

---

 ns-load

*Loading and Unloading Name Spaces*


---

### Description

Functions to load and unload namespaces.

### Usage

```
attachNamespace(ns, pos = 2, dataPath = NULL)
loadNamespace(package, lib.loc = NULL,
              keep.source = getOption("keep.source.pkgs"),
              partial = FALSE, declarativeOnly = FALSE)
loadedNamespaces()
unloadNamespace(ns)
```

### Arguments

<code>ns</code>	string or namespace object.
<code>pos</code>	integer specifying position to attach.
<code>dataPath</code>	path to directory containing a database of datasets to be lazy-loaded into the attached environment.
<code>package</code>	string naming the package/name space to load.
<code>lib.loc</code>	character vector specifying library search path.
<code>keep.source</code>	logical specifying whether to retain source.
<code>partial</code>	logical; if true, stop just after loading code.
<code>declarativeOnly</code>	logical; disables <code>.Import</code> , etc, if true.

### Details

The functions `loadNamespace` and `attachNamespace` are usually called implicitly when `library` is used to load a name space and any imports needed. However it may be useful to call these functions directly at times.

`loadNamespace` loads the specified name space and registers it in an internal data base. A request to load a name space that is already loaded has no effect. The arguments have the same meaning as the corresponding arguments to `library`. After loading, `loadNamespace` looks for a hook function named `.onLoad` as an internal variable in the name space (it should not be

exported). This function is called with the same arguments as `.First.lib`. Partial loading is used to support installation with the `'--save'` and `'--lazy'` options.

`loadNamespace` does not attach the name space it loads to the search path. `attachNamespace` can be used to attach a frame containing the exported values of a name space to the search path. The hook function `.onAttach` is run after the name space exports are attached.

`loadedNamespaces` returns a character vector of the names of the loaded name spaces.

`unloadNamespace` can be used to force a name space to be unloaded. An error is signaled if the name space is imported by other loaded name spaces. If defined, a hook function `code{.onUnload}` is run before removing the name space from the internal registry. `unloadNamespace` will first `detach` a package of the same name if one is on the path, thereby running a `.Last.lib` function in the package if one is exported.

### Author(s)

Luke Tierney

---

ns-topenv

*Top Level Environment*

---

### Description

Finding the top level environment.

### Usage

```
topenv(envir = parent.frame(),
       matchThisEnv = getOption("topLevelEnvironment"))
```

### Arguments

`envir` environment.

`matchThisEnv` return this environment, if it matches before any other criterion is satisfied. The default, the option "topLevelEnvironment", is set by `sys.source`, which treats a specific environment as the top level environment. Supplying the argument as NULL means it will never match.

### Details

`topenv` returns the first top level environment found when searching `envir` and its parent environments. An environment is considered top level if it is the internal environment of a name space, a package environment in the search path, or `.GlobalEnv`.

### Examples

```
topenv(.GlobalEnv)
topenv(new.env())
```

---

NULL

*The Null Object*

---

### Description

NULL represents the null object in R. NULL is used mainly to represent the lists with zero length, and is often returned by expressions and functions whose value is undefined.

`as.null` ignores its argument and returns the value NULL.

`is.null` returns TRUE if its argument is NULL and FALSE otherwise.

### Usage

```
NULL
as.null(x, ...)
is.null(x)
```

### Arguments

<code>x</code>	an object to be tested or coerced.
<code>...</code>	ignored.

### Details

`is.null` is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### Examples

```
is.null(list())      # FALSE (on purpose!)
is.null(integer(0)) # F
is.null(logical(0)) # F
as.null(list(a=1,b='c'))
```

---

`numeric`*Numeric Vectors*

---

### Description

Creates or tests for objects of type "numeric".

### Usage

```
numeric(length = 0)
as.numeric(x, ...)
is.numeric(x)
```

### Arguments

<code>length</code>	desired length.
<code>x</code>	object to be coerced or tested.
<code>...</code>	further arguments passed to or from other methods.

### Details

`as.numeric` is a generic function, but methods must be written for [as.double](#), which it calls. `is.numeric` is generic: you can write methods to handle specific classes of objects, see [Internal-Methods](#).

Note that factors are false for `is.numeric` since there is a "factor" method.

### Value

`numeric` creates a real vector of the specified length. The elements of the vector are all equal to 0.

`as.numeric` attempts to coerce its argument to numeric type (either integer or real). `as.numeric` for factors yields the codes underlying the factor levels, not the numeric representation of the labels.

`is.numeric` returns TRUE if its argument is of type numeric or type integer and FALSE otherwise.

### Note

*R has no single precision data type. All real numbers are stored in double precision format.*

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```
as.numeric(c("-.1", " 2.7 ", "B")) # (-0.1, 2.7, NA) + warning
as.numeric(factor(5:10))
```

---

octmode

*Display Numbers in Octal*

---

**Description**

Convert or print integers in octal format, with as many digits as are needed to display the largest, using leading zeroes as necessary.

**Usage**

```
## S3 method for class 'octmode':
as.character(x, ...)

## S3 method for class 'octmode':
format(x, ...)

## S3 method for class 'octmode':
print(x, ...)
```

**Arguments**

`x` An object inheriting from class "octmode".  
`...` further arguments passed to or from other methods.

**Details**

Class "octmode" consists of integer vectors with that class attribute, used merely to ensure that they are printed in octal notation, specifically for Unix-like file permissions such as 755.

**See Also**

These are auxiliary functions for [file.info](#)

---

`on.exit`*Function Exit Code*

---

**Description**

`on.exit` records the expression given as its argument as needing to be executed when the current function exits (either naturally or as the result of an error). This is useful for resetting graphical parameters or performing other cleanup actions.

If no expression is provided, i.e., the call is `on.exit()`, then the current `on.exit` code is removed.

**Usage**

```
on.exit(expr, add = FALSE)
```

**Arguments**

`expr` an expression to be executed.  
`add` if TRUE, add `expr` to be executed after any previously set expressions.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[sys.on.exit](#) to see the current expression.

**Examples**

```
opar <- par(mai = c(1,1,1,1))
on.exit(par(opar))
```

---

`options`*Options Settings*

---

**Description**

Allow the user to set and examine a variety of global “options” which affect the way in which R computes and displays its results.

**Usage**

```
options(...)  
  
getOption(x)  
  
.Options
```

**Arguments**

... any options can be defined, using `name = value` or by passing a list of such tagged values. However, only the ones below are used in “base R”. Further, `options('name') == options()[ 'name' ]`, see the example.

x a character string holding an option name.

**Details**

Invoking `options()` with no arguments returns a list with the current values of the options. Note that not all options listed below are set initially. To access the value of a single option, one should use `getOption("width")`, e.g., rather than `options("width")` which is a *list* of length one.

`.Options` also always contains the `options()` list (as a pairlist), for S compatibility. You must use it “read only” however.

**Value**

For `options`, a list (in any case) with the previous values of the options changed, or all options when no arguments were given.

**Options used in base R**

**prompt:** a string, used for R’s prompt; should usually end in a blank (" ").

**continue:** a string setting the prompt used for lines which continue over one line.

**width:** controls the number of characters on a line. You may want to change this if you re-size the window that R is running in. Valid values are 10...10000 with default normally 80. (The valid values are in file ‘Print.h’ and can be changed by re-compiling R.)

**digits:** controls the number of digits to print when printing numeric values. It is a suggestion only. Valid values are 1...22 with default 7. See [print.default](#).

**editor:** sets the default text editor, e.g., for [edit](#). Set from the environment variable VISUAL on UNIX.

**pager:** the (stand-alone) program used for displaying ASCII files on R’s console, also used by [file.show](#) and sometimes [help](#). Defaults to ‘\$R\_HOME/bin/pager’.

**browser:** default HTML browser used by [help.start\(\)](#) on UNIX, or a non-default browser on Windows.

**pdfviewer:** default PDF viewer. Set from the environment variable R\_PDFVIEWER.

**mailer:** default mailer used by [bug.report\(\)](#). Can be "none".

- contrasts:** the default `contrasts` used in model fitting such as with `aov` or `lm`. A character vector of length two, the first giving the function to be used with unordered factors and the second the function to be used with ordered factors.
- defaultPackages:** the packages that are attached by default when R starts up. Initially set from value of the environment variables `R_DefaultPackages`, or if that is unset to `c("utils", "stats", "graphics", "methods")`. (Set `R_DEFAULT_PACKAGES` to `NULL` or a comma-separated list of package names.) A call to `options` should be in your `‘.Rprofile’` file to ensure that the change takes effect before the base package is initialized (see `Startup`).
- expressions:** sets a limit on the number of nested expressions that will be evaluated. Valid values are 25...100000 with default 500.
- keep.source:** When `TRUE`, the source code for functions (newly defined or loaded) is stored in their `"source"` attribute (see `attr`) allowing comments to be kept in the right places. The default is `interactive()`, i.e., `TRUE` for interactive use.
- keep.source.pkgs:** As for `keep.source`, for functions in packages loaded by `library` or `require`. Defaults to `FALSE` unless the environment variable `R_KEEP_PKG_SOURCE` is set to `yes`.
- na.action:** the name of a function for treating missing values (`NA`'s) for certain situations.
- papersize:** the default paper format used by `postscript`; set by environment variable `R_PAPERSIZE` when R is started and defaulting to `"a4"` if that is unset or invalid.
- printcmd:** the command used by `postscript` for printing; set by environment variable `R_PRINTCMD` when R is started. This should be a command that expects either input to be piped to `‘stdin’` or to be given a single filename argument.
- latexcmd, dvipscmd:** character strings giving commands to be used in off-line printing of help pages.
- show.signif.stars, show.coef.Pvalues:** logical, affecting P value printing, see `printCoefmat`.
- ts.eps:** the relative tolerance for certain time series (`ts`) computations.
- error:** either a function or an expression governing the handling of non-catastrophic errors such as those generated by `stop` as well as by signals and internally detected errors. If the option is a function, a call to that function, with no arguments, is generated as the expression. The default value is `NULL`: see `stop` for the behaviour in that case. The function `dump.frames` provides one alternative that allows post-mortem debugging.
- show.error.messages:** a logical. Should error messages be printed? Intended for use with `try` or a user-installed error handler.
- warn:** sets the handling of warning messages. If `warn` is negative all warnings are ignored. If `warn` is zero (the default) warnings are stored until the top-level function returns. If fewer than 10 warnings were signalled they will be printed otherwise a message saying how many (max 50) were signalled. A top-level variable called `last.warning` is created and can be viewed through the function `warnings`. If `warn` is one, warnings are printed as they occur. If `warn` is two or larger all warnings are turned into errors.
- warning.length:** sets the truncation limit for error and warning messages. A non-negative integer, with allowed values 100–8192, default 1000.

- warning.expression:** an R code expression to be called if a warning is generated, replacing the standard message. If non-null it is called irrespective of the value of option `warn`.
- check.bounds:** logical, defaulting to `FALSE`. If true, a `warning` is produced whenever a “generalized vector” (atomic or `list`) is extended, by something like `x <- 1:3; x[5] <- 6`.
- echo:** logical. Only used in non-interactive mode, when it controls whether input is echoed. Command-line option `‘-slave’` sets this initially to `FALSE`.
- verbose:** logical. Should R report extra information on progress? Set to `TRUE` by the command-line option `‘-verbose’`.
- device:** a character string giving the default device for that session. This defaults to the normal screen device (e.g., `x11`, `windows` or `gtk`) for an interactive session, and `postscript` in batch use or if a screen is not available.
- X11colortype:** The default colour type for `X11` devices.
- CRAN:** The URL of the preferred CRAN node for use by `update.packages`. Defaults to <http://cran.r-project.org>.
- repositories:** A function that when evaluated returns a named vector of URLs which can be used to specify the set and order of any online repositories the user would like used in functions such as `pkgDepends`. The default return value for the function is a vector with two components, one the value of `contrib.url(getOption("CRAN"))` and the second `contrib.url(getOption("BIOC"))`.
- download.file.method:** Method to be used for `download.file`. Currently download methods `"internal"`, `"wget"` and `"lynx"` are available. There is no default for this option, when `method = "auto"` is chosen: see [download.file](#).
- unzip:** the command used for unzipping help files. Defaults to the value of `R_UNZIPCMD`, which is set in `‘etc/Renviron’` if an `unzip` command was found during configuration.
- de.cellwidth:** integer: the cell widths (number of characters) to be used in the data editor [dataentry](#). If this is unset, 0, negative or `NA`, variable cell widths are used.
- encoding:** An integer vector of length 256 holding an input encoding. Defaults to `native.enc` (`= 0:255`). See [connections](#).
- timeout:** integer. The timeout for some Internet operations, in seconds. Default 60 seconds. See [download.file](#) and [connections](#).
- internet.info:** The minimum level of information to be printed on URL downloads etc. Default is 2, for failure causes. Set to 1 or 0 to get more information.
- scipen:** integer. A penalty to be applied when deciding to print numeric values in fixed or exponential notation. Positive values bias towards fixed and negative towards scientific notation: fixed notation will be preferred unless it is more than `scipen` digits wider.
- locatorBell:** logical. Should selection in `locator` and `identify` be confirmed by a bell. Default `TRUE`. Honoured at least on `X11` and `windows` devices.
- X11fonts:** character vector of length 2. See [X11](#).

The default settings of some of these options are

<code>prompt</code>	<code>"&gt; "</code>	<code>continue</code>	<code>" + "</code>
<code>width</code>	80	<code>digits</code>	7

expressions	500	keep.source	TRUE
show.signif.stars	TRUE	show.coef.Pvalues	TRUE
na.action	na.omit	ts.eps	1e-5
error	NULL	show.error.messages	TRUE
warn	0	warning.length	1000
echo	TRUE	verbose	FALSE
scipen	0	locatorBell	TRUE

Others are set from environment variables or are platform-dependent.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## Examples

```
options() # printing all current options
op <- options(); str(op) # nicer printing

# .Options is the same:
all(sapply(1:length(op), function(i) if(is.atomic(op[[i]]))
      {all(.Options[[i]] == op[[i]])} else TRUE))

options('width')[[1]] == options()$width # the latter needs more memory
options(digits=20)
pi

# set the editor, and save previous value
old.o <- options(editor="nedit")
old.o

options(check.bounds = TRUE)
x <- NULL; x[4] <- "yes" # gives a warning

options(digits=5)
print(1e5)
options(scipen=3); print(1e5)

options(op) # reset (all) initial options
options('digits')

## Not run:
## set contrast handling to be like S
options(contrasts=c("contr.helmert", "contr.poly"))
## End(Not run)
## Not run:
## on error, terminate the R session with error status 66
options(error=quote(q("no", status=66, runLast=FALSE)))
stop("test it")
## End(Not run)
```

```
## Not run:
## set an error action for debugging: see ?debugger.
options(error=dump.frames)
## A possible setting for non-interactive sessions
options(error=quote({dump.frames(to.file=TRUE); q()}))
## End(Not run)
```

---

order

*Ordering Permutation*


---

### Description

`order` returns a permutation which rearranges its first argument into ascending or descending order, breaking ties by further arguments. `sort.list` is the same, using only one argument.

### Usage

```
order(..., na.last = TRUE, decreasing = FALSE)

sort.list(x, partial = NULL, na.last = TRUE, decreasing = FALSE,
          method = c("shell", "quick", "radix"))
```

### Arguments

<code>...</code>	a sequence of vectors, all of the same length.
<code>x</code>	a vector.
<code>partial</code>	vector of indices for partial sorting.
<code>decreasing</code>	logical. Should the sort order be increasing or decreasing?
<code>na.last</code>	for controlling the treatment of NAs. If <code>TRUE</code> , missing values in the data are put last; if <code>FALSE</code> , they are put first; if <code>NA</code> , they are removed.
<code>method</code>	the method to be used: partial matches are allowed.

### Details

In the case of ties in the first vector, values in the second are used to break the ties. If the values are still tied, values in the later arguments are used to break the tie (see the first example). The sort used is *stable* (except for `method = "quick"`), so any unresolved ties will be left in their original ordering.

The default method for `sort.list` is a good compromise. Method `"quick"` is only supported for numeric `x` with `na.last=NA`, and is not stable, but will be faster for long vectors. Method `"radix"` is only implemented for integer `x` with a range of less than 100,000. For such `x` it is very fast (and stable), and hence is ideal for sorting factors.

`partial` is supplied for compatibility with other implementations of `S`, but no other values are accepted and ordering is always complete.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[sort](#) and [rank](#).

## Examples

```
(ii <- order(x <- c(1,1,3:1,1:4,3), y <- c(9,9:1), z <-c(2,1:9)))
## 6 5 2 1 7 4 10 8 3 9
rbind(x,y,z)[,ii] # shows the reordering (ties via 2nd & 3rd arg)

## Suppose we wanted descending order on y. A simple solution is
rbind(x,y,z)[, order(x, -y, z)]
## For character vectors we can make use of rank:
cy <- as.character(y)
rbind(x,y,z)[, order(x, -rank(y), z)]

## rearrange matched vectors so that the first is in ascending order
x <- c(5:1, 6:8, 12:9)
y <- (x - 5)^2
o <- order(x)
rbind(x[o], y[o])

## tests of na.last
a <- c(4, 3, 2, NA, 1)
b <- c(4, NA, 2, 7, 1)
z <- cbind(a, b)
(o <- order(a, b)); z[o, ]
(o <- order(a, b, na.last = FALSE)); z[o, ]
(o <- order(a, b, na.last = NA)); z[o, ]

## Not run:
## speed examples for long vectors: timings are immediately after gc()
x <- factor(sample(letters, 1e6, replace=TRUE))
system.time(o <- sort.list(x)) ## 4 secs
stopifnot(!is.unsorted(x[o]))
system.time(o <- sort.list(x, method="quick", na.last=NA)) # 0.4 sec
stopifnot(!is.unsorted(x[o]))
system.time(o <- sort.list(x, method="radix")) # 0.04 sec
stopifnot(!is.unsorted(x[o]))
xx <- sample(1:26, 1e7, replace=TRUE)
system.time(o <- sort.list(xx, method="radix")) # 0.4 sec
xx <- sample(1:100000, 1e7, replace=TRUE)
system.time(o <- sort.list(xx, method="radix")) # 4 sec
## End(Not run)
```

---

`outer`*Outer Product of Arrays*

---

**Description**

The outer product of the arrays `X` and `Y` is the array `A` with dimension `c(dim(X), dim(Y))` where element `A[c(arrayindex.x, arrayindex.y)] = FUN(X[arrayindex.x], Y[arrayindex.y], ...)`.

**Usage**

```
outer(X, Y, FUN="*", ...)  
X %o% Y
```

**Arguments**

<code>X</code>	A vector or array.
<code>Y</code>	A vector or array.
<code>FUN</code>	a function to use on the outer products, it may be a quoted string.
<code>...</code>	optional arguments to be passed to <code>FUN</code> .

**Details**

`FUN` must be a function (or the name of it) which expects at least two arguments and which operates elementwise on arrays.

Where they exist, the `[dim]names` of `X` and `Y` will be preserved.

`%o%` is an alias for `outer` (where `FUN` cannot be changed from `"*"`).

**Author(s)**

Jonathan Rougier

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`%*%` for usual (*inner*) matrix vector multiplication; `kron` which is based on `outer`.

**Examples**

```
x <- 1:9; names(x) <- x
# Multiplication & Power Tables
x %o% x
y <- 2:8; names(y) <- paste(y, ":", sep=" ")
outer(y, x, "^")

outer(month.abb, 1999:2003, FUN = "paste")

## three way multiplication table:
x %o% x %o% y[1:3]
```

---

package-version      *Package versions*

---

**Description**

A simple S3 class for representing package versions, and associated methods.

**Usage**

```
package_version(x, strict = TRUE)

getRversion()
```

**Arguments**

<code>x</code>	a character vector with package version strings.
<code>strict</code>	a logical indicating whether invalid package versions should results in an error (default) or not.

**Details**

R (package) versions are sequences of at least two non-negative integers, usually (e.g., in package ‘DESCRIPTION’ files) represented as character strings with the elements of the sequence concatenated and separated by single ‘.’ or ‘-’ characters.

`package_version` creates a representation from such strings which allows for coercion and testing, combination, comparison, summaries (min/max), inclusion in data frames, subscripting, and printing.

`getRversion` returns the version of the running R as an object of class "package\_version".

**See Also**

[compareVersion](#)

**Examples**

```
x <- package_version(c("1.2-4", "1.2-3", "2.1"))
x < "1.4-2.3"
c(min(x), max(x))
x[2, 2]
x$major
x$minor
```

---

Paren

*Parentheses and Braces*

---

**Description**

Open parenthesis, (, and open brace, {, are [.Primitive](#) functions in R.

Effectively, ( is semantically equivalent to the identity function(x) x, whereas { is slightly more interesting, see examples.

**Usage**

```
( ... )
{ ... }
```

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[if](#), [return](#), etc for other objects used in the R language itself.

[Syntax](#) for operator precedence.

**Examples**

```
f <- get("(")
e <- expression(3 + 2 * 4)
identical(f(e), e)

do <- get("{")
do(x <- 3, y <- 2*x-3, 6-x-y); x; y
```

---

parse

*Parse Expressions*

---

### Description

`parse` returns the parsed but unevaluated expressions in a list. Each element of the list is of mode `expression`.

### Usage

```
parse(file = "", n = NULL, text = NULL, prompt = "?")
```

### Arguments

<code>file</code>	a connection, or a character string giving the name of a file or a URL to read the expressions from. If <code>file</code> is "" and <code>text</code> is missing or <code>NULL</code> then input is taken from the console.
<code>n</code>	the number of statements to parse. If <code>n</code> is negative the file is parsed in its entirety.
<code>text</code>	character vector. The text to parse. Elements are treated as if they were lines of a file.
<code>prompt</code>	the prompt to print when parsing from the keyboard. <code>NULL</code> means to use R's prompt, <code>getOption("prompt")</code> .

### Details

All versions of R accept input from a connection with end of line marked by LF (as used on Unix), CRLF (as used on DOS/Windows) or CR (as used on classic MacOS). The final line can be incomplete, that is missing the final EOL marker.

See [source](#) for the limits on the size of functions that can be parsed (by default).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[scan](#), [source](#), [eval](#), [deparse](#).

### Examples

```
cat("x <- c(1,4)\n x ^ 3 -10 ; outer(1:7,5:9)\n", file="xyz.Rdmped")
# parse 3 statements from the file "xyz.Rdmped"
parse(file = "xyz.Rdmped", n = 3)
unlink("xyz.Rdmped")
```

---

paste	<i>Concatenate Strings</i>
-------	----------------------------

---

## Description

Concatenate vectors after converting to character.

## Usage

```
paste(..., sep = " ", collapse = NULL)
```

## Arguments

...	one or more R objects, to be coerced to character vectors.
sep	a character string to separate the terms.
collapse	an optional character string to separate the results.

## Details

`paste` converts its arguments to character strings, and concatenates them (separating them by the string given by `sep`). If the arguments are vectors, they are concatenated term-by-term to give a character vector result.

If a value is specified for `collapse`, the values in the result are then concatenated into a single string, with the elements being separated by the value of `collapse`.

## Value

A character vector of the concatenated values. This will be of length zero if all the objects are, unless `collapse` is non-NULL, in which case it is a single empty string.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

String manipulation with [as.character](#), [substr](#), [nchar](#), [strsplit](#); further, [cat](#) which concatenates and writes to a file, and [sprintf](#) for C like string construction.

## Examples

```
paste(1:12) # same as as.character(1:12)
paste("A", 1:6, sep = " ")
paste("Today is", date())
```

---

`path.expand`                    *Expand File Paths*

---

### Description

Expand a path name, for example by replacing a leading tilde by the user's home directory (if defined on that platform).

### Usage

```
path.expand(path)
```

### Arguments

`path`                    character vector containing one or more path names.

### Details

On *some Unix* versions, a leading `~user` will expand to the home directory of `user`, but not on Unix versions without `readline` installed.

### See Also

[basename](#)

### Examples

```
path.expand("~/foo")
```

---

`pmatch`                    *Partial String Matching*

---

### Description

`pmatch` seeks matches for the elements of its first argument among those of its second.

### Usage

```
pmatch(x, table, nomatch = NA, duplicates.ok = FALSE)
```

### Arguments

`x`                    the values to be matched.  
`table`                the values to be matched against.  
`nomatch`             the value returned at non-matching or multiply partially matching positions.  
`duplicates.ok`        should elements be in `table` be used more than once?

## Details

The behaviour differs by the value of `duplicates.ok`. Consider first the case if this is true. First exact matches are considered, and the positions of the first exact matches are recorded. Then unique partial matches are considered, and if found recorded. (A partial match occurs if the whole of the element of `x` matches the beginning of the element of `table`.) Finally, all remaining elements of `x` are regarded as unmatched. In addition, an empty string can match nothing, not even an exact match to an empty string. This is the appropriate behaviour for partial matching of character indices, for example.

If `duplicates.ok` is `FALSE`, values of `table` once matched are excluded from the search for subsequent matches. This behaviour is equivalent to the R algorithm for argument matching, except for the consideration of empty strings (which in argument matching are matched after exact and partial matching to any remaining arguments).

`charmatch` is similar to `pmatch` with `duplicates.ok` true, the differences being that it differentiates between no match and an ambiguous partial match, it does match empty strings, and it does not allow multiple exact matches.

## Value

A numeric vector of integers (including `NA` if `nomatch = NA`) of the same length as `x`, giving the indices of the elements in `table` which matched, or `nomatch`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

## See Also

`match`, `charmatch` and `match.arg`, `match.fun`, `match.call`, for function argument matching etc., `grep` etc for more general (regexp) matching of strings.

## Examples

```
pmatch("", "") # returns NA
pmatch("m", c("mean", "median", "mode")) # returns NA
pmatch("med", c("mean", "median", "mode")) # returns 2

pmatch(c("", "ab", "ab"), c("abc", "ab"), dup=FALSE)
pmatch(c("", "ab", "ab"), c("abc", "ab"), dup=TRUE)
## compare
charmatch(c("", "ab", "ab"), c("abc", "ab"))
```

---

`polyroot`*Find Zeros of a Real or Complex Polynomial*

---

**Description**

Find zeros of a real or complex polynomial.

**Usage**

```
polyroot(z)
```

**Arguments**

`z` the vector of polynomial coefficients in increasing order.

**Details**

A polynomial of degree  $n - 1$ ,

$$p(x) = z_1 + z_2x + \cdots + z_nx^{n-1}$$

is given by its coefficient vector `z[1:n]`. `polyroot` returns the  $n - 1$  complex zeros of  $p(x)$  using the Jenkins-Traub algorithm.

If the coefficient vector `z` has zeroes for the highest powers, these are discarded.

**Value**

A complex vector of length  $n - 1$ , where  $n$  is the position of the largest non-zero element of `z`.

**References**

Jenkins and Traub (1972) TOMS Algorithm 419. *Comm. ACM*, **15**, 97–99.

**See Also**

[uniroot](#) for numerical root finding of arbitrary functions; [complex](#) and the `zero` example in the demos directory.

**Examples**

```
polyroot(c(1, 2, 1))
round(polyroot(choose(8, 0:8)), 11) # guess what!
for (n1 in 1:4) print(polyroot(1:n1), digits = 4)
polyroot(c(1, 2, 1, 0, 0)) # same as the first
```

---

`pos.to.env`*Convert Positions in the Search Path to Environments*

---

**Description**

Returns the environment at a specified position in the search path.

**Usage**

```
pos.to.env(x)
```

**Arguments**

`x` an integer between 1 and `length(search())`, the length of the search path.

**Details**

Several R functions for manipulating objects in environments (such as `get` and `ls`) allow specifying environments via corresponding positions in the search path. `pos.to.env` is a convenience function for programmers which converts these positions to corresponding environments; users will typically have no need for it.

**Examples**

```
pos.to.env(1) # R_GlobalEnv
# the next returns NULL, which is how package:base is represented.
pos.to.env(length(search()))
```

---

`pretty`*Pretty Breakpoints*

---

**Description**

Compute a sequence of about `n+1` equally spaced nice values which cover the range of the values in `x`. The values are chosen so that they are 1, 2 or 5 times a power of 10.

**Usage**

```
pretty(x, n = 5, min.n = n %% 3, shrink.sml = 0.75,
       high.u.bias = 1.5, u5.bias = .5 + 1.5*high.u.bias,
       eps.correct = 0)
```

**Arguments**

<code>x</code>	numeric vector
<code>n</code>	integer giving the <i>desired</i> number of intervals. Non-integer values are rounded down.
<code>min.n</code>	nonnegative integer giving the <i>minimal</i> number of intervals. If <code>min.n == 0</code> , <code>pretty(.)</code> may return a single value.
<code>shrink.sml</code>	positive numeric by which a default scale is shrunk in the case when <code>range(x)</code> is “very small” (usually 0).
<code>high.u.bias</code>	non-negative numeric, typically $> 1$ . The interval unit is determined as $\{1,2,5,10\}$ times <code>b</code> , a power of 10. Larger <code>high.u.bias</code> values favor larger units.
<code>u5.bias</code>	non-negative numeric multiplier favoring factor 5 over 2. Default and “optimal”: <code>u5.bias = .5 + 1.5*high.u.bias</code> .
<code>eps.correct</code>	integer code, one of $\{0,1,2\}$ . If non-0, an “ <i>epsilon correction</i> ” is made at the boundaries such that the result boundaries will be outside <code>range(x)</code> ; in the “ <i>small</i> ” case, the correction is only done if <code>eps.correct &gt;=2</code> .

**Details**

As from R 2.0.0 `pretty` ignores non-finite values in `x`.

Let  $d \leftarrow \max(x) - \min(x) \geq 0$ . If  $d$  is not (very close) to 0, we let  $c \leftarrow d/n$ , otherwise more or less  $c \leftarrow \max(\text{abs}(\text{range}(x))) * \text{shrink.sml} / \text{min.n}$ . Then, the *10 base*  $b$  is  $10^{\lfloor \log_{10}(c) \rfloor}$  such that  $b \leq c < 10b$ .

Now determine the basic *unit*  $u$  as one of  $\{1, 2, 5, 10\}b$ , depending on  $c/b \in [1, 10)$  and the two “*bias*” coefficients,  $h = \text{high.u.bias}$  and  $f = \text{u5.bias}$ .

.....

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[axTicks](#) for the computation of `pretty` axis tick locations in plots, particularly on the log scale.

**Examples**

```
pretty(1:15)      # 0  2  4  6  8 10 12 14 16
pretty(1:15, h=2)# 0  5 10 15
pretty(1:15, n=4)# 0  5 10 15
pretty(1:15 * 2) # 0  5 10 15 20 25 30
pretty(1:20)     # 0  5 10 15 20
pretty(1:20, n=2) # 0 10 20
pretty(1:20, n=10)# 0  2  4 ... 20

for(k in 5:11) {
```

```

    cat("k=",k,": "); print(diff(range(pretty(100 + c(0, pi*10^-k))))))}

##-- more bizarre, when min(x) == max(x):
pretty(pi)

add.names <- function(v) { names(v) <- paste(v); v}
str(lapply(add.names(-10:20), pretty))
str(lapply(add.names(0:20), pretty, min = 0))
sapply( add.names(0:20), pretty, min = 4)

pretty(1.234e100)
pretty(1001.1001)
pretty(1001.1001, shrink = .2)
for(k in -7:3)
  cat("shrink=",formatC(2^k,wid=9),":",
      formatC(pretty(1001.1001, shrink = 2^k), wid=6),"\n")

```

---

 Primitive

 Call a “Primitive” Internal Function
 

---

## Description

`.Primitive` returns an entry point to a “primitive” (internally implemented) function.

The advantage of `.Primitive` over `.Internal` functions is the potential efficiency of argument passing.

## Usage

```
.Primitive(name)
```

## Arguments

`name`                    name of the R function.

## See Also

[.Internal](#).

## Examples

```

mysqrt <- .Primitive("sqrt")
c
.Internal # this one *must* be primitive!
get("if") # just 'if' or 'print(if)' are not syntactically ok.

```

---

print *Print Values*

---

### Description

print prints its argument and returns it *invisibly* (via `invisible(x)`). It is a generic function which means that new printing methods can be easily added for new `classes`.

### Usage

```
print(x, ...)

## S3 method for class 'factor':
print(x, quote = FALSE, max.levels = NULL,
      width = getOption("width"), ...)

## S3 method for class 'table':
print(x, digits = getOption("digits"), quote = FALSE,
      na.print = "", zero.print = "0", justify = "none", ...)
```

### Arguments

x	an object used to select a method.
...	further arguments passed to or from other methods.
quote	logical, indicating whether or not strings should be printed with surrounding quotes.
max.levels	integer, indicating how many levels should be printed for a factor; if 0, no extra "Levels" line will be printed. The default, NULL, entails choosing max.levels such that the levels print on one line of width width.
width	only used when max.levels is NULL, see above.
digits	minimal number of <i>significant</i> digits, see <code>print.default</code> .
na.print	character string (or NULL) indicating NA values in printed output, see <code>print.default</code> .
zero.print	character specifying how zeros (0) should be printed; for sparse tables, using "." can produce stronger results.
justify	character indicating if strings should left- or right-justified or left alone, passed to <code>format</code> .

### Details

The default method, `print.default` has its own help page. Use `methods("print")` to get all the methods for the `print` generic.

`print.factor` allows some customization and is used for printing `ordered` factors as well.

`print.table` for printing `tables` allows other customization.

See `noquote` as an example of a class whose main purpose is a specific `print` method.

## References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

## See Also

The default method `print.default`, and help for the methods above; further `options`, `noquote`.

For more customizable (but cumbersome) printing, see `cat`, `format` or also `write`.

## Examples

```
ts(1:20)#-- print is the "Default function" --> print.ts(.) is called
rr <- for(i in 1:3) print(1:i)
rr

## Printing of factors
attenu$station ## 117 levels -> `max.levels` depending on width

## ordered factors: levels "l1 < l2 < .."
esoph$agegp[1:12]
esoph$alcgp[1:12]

## Printing of sparse (contingency) tables
set.seed(521)
t1 <- round(abs(rt(200, df=1.8)))
t2 <- round(abs(rt(200, df=1.4)))
table(t1,t2) # simple
print(table(t1,t2), zero.print = ".")# nicer to read
```

---

`print.data.frame`    *Printing Data Frames*

---

## Description

Print a data frame.

## Usage

```
## S3 method for class 'data.frame':
print(x, ..., digits = NULL, quote = FALSE, right = TRUE)
```

## Arguments

<code>x</code>	object of class <code>data.frame</code> .
<code>...</code>	optional arguments to <code>print</code> or <code>plot</code> methods.
<code>digits</code>	the minimum number of significant digits to be used.
<code>quote</code>	logical, indicating whether or not entries should be printed with surrounding quotes.

`right` logical, indicating whether or not strings should be right-aligned. The default is left-alignment.

### Details

This calls `format` which formats the data frame column-by-column, then converts to a character matrix and dispatches to the `print` method for matrices.

When `quote = TRUE` only the entries are quoted not the row names nor the column names.

### See Also

[data.frame](#).

---

`print.default`      *Default Printing*

---

### Description

`print.default` is the *default* method of the generic `print` function which prints its argument. `print.matrix` is currently identical, but was not prior to 1.7.0.

### Usage

```
## Default S3 method:
print(x, digits = NULL, quote = TRUE, na.print = NULL,
      print.gap = NULL, right = FALSE, ...)
```

### Arguments

<code>x</code>	the object to be printed.
<code>digits</code>	a non-null value for <code>digits</code> specifies the minimum number of significant digits to be printed in values. If <code>digits</code> is <code>NULL</code> , the value of <code>digits</code> set by <code>options</code> is used.
<code>quote</code>	logical, indicating whether or not strings ( <a href="#">characters</a> ) should be printed with surrounding quotes.
<code>na.print</code>	a character string which is used to indicate <a href="#">NA</a> values in printed output, or <code>NULL</code> (see <a href="#">Details</a> )
<code>print.gap</code>	a non-negative integer $\leq 1024$ , giving the spacing between adjacent “columns” in printed vectors, matrices and arrays, or <code>NULL</code> meaning 1.
<code>right</code>	logical, indicating whether or not strings should be right-aligned. The default is left-alignment.
<code>...</code>	further arguments to be passed to or from other methods. They are ignored in these functions.

## Details

Prior to R 1.7.0, `print.matrix` did not print attributes and did not have a `digits` argument.

The default for printing NAs is to print NA (without quotes) unless this is a character NA *and* `quote = FALSE`, when `<NA>` is printed.

The same number of decimal places is used throughout a vector, This means that `digits` specifies the minimum number of significant digits to be used, and that at least one entry will be printed with that minimum number.

As from R 1.7.0 attributes are printed respecting their class(es), using the values of `digits` to `print.default`, but using the default values (for the methods called) of the other arguments.

When the **methods** package is attached, `print` will call `show` for methods with formal classes if called with no optional arguments.

## See Also

The generic `print`, `options`. The `"noquote"` class and `print` method.

## Examples

```
pi
print(pi, digits = 16)
LETTERS[1:16]
print(LETTERS, quote = FALSE)
```

---

prmatrix

*Print Matrices, Old-style*

---

## Description

An earlier method for printing matrices, provided for S compatibility.

## Usage

```
prmatrix(x, rowlab =, collab =,
         quote = TRUE, right = FALSE, na.print = NULL, ...)
```

## Arguments

<code>x</code>	numeric or character matrix.
<code>rowlab, collab</code>	(optional) character vectors giving row or column names respectively. By default, these are taken from <code>dimnames(x)</code> .
<code>quote</code>	logical; if TRUE and <code>x</code> is of mode "character", <i>quotes</i> (") are used.
<code>right</code>	if TRUE and <code>x</code> is of mode "character", the output columns are <i>right-justified</i> .
<code>na.print</code>	how NAs are printed. If this is non-null, its value is used to represent NA.
<code>...</code>	arguments for <code>print</code> methods.

**Details**

`prmatrix` is an earlier form of `print.matrix`, and is very similar to the S function of the same name.

**Value**

Invisibly returns its argument, `x`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`print.default`, and other `print` methods.

**Examples**

```
prmatrix(m6 <- diag(6), row = rep("",6), coll=rep("",6))

chm <- matrix(scan(system.file("help", "AnIndex", package = "splines"),
                           what = ""), , 2, byrow = TRUE)
chm # uses print.matrix()
prmatrix(chm, collab = paste("Column",1:3), right=TRUE, quote=FALSE)
```

---

`proc.time`

*Running Time of R*

---

**Description**

`proc.time` determines how much time (in seconds) the currently running R process already consumed.

**Usage**

```
proc.time()
```

**Value**

A numeric vector of length 5, containing the user, system, and total elapsed times for the currently running R process, and the cumulative sum of user and system times of any child processes spawned by it.

The resolution of the times will be system-specific; it is common for them to be recorded to of the order of 1/100 second, and elapsed time is rounded to the nearest 1/100.

It is most useful for “timing” the evaluation of R expressions, which can be done conveniently with `system.time`.

**Note**

It is possible to compile R without support for `proc.time`, when the function will not exist.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[system.time](#) for timing a valid R expression, [gc.time](#) for how much of the time was spent in garbage collection.

**Examples**

```
## Not run:
## a way to time an R expression: system.time is preferred
ptm <- proc.time()
for (i in 1:50) mad(runif(500))
proc.time() - ptm
## End(Not run)
```

---

prod

*Product of Vector Elements*

---

**Description**

`prod` returns the product of all the values present in its arguments.

**Usage**

```
prod(..., na.rm = FALSE)
```

**Arguments**

...            numeric vectors.  
na.rm         logical. Should missing values be removed?

**Details**

If `na.rm` is `FALSE` an NA value in any of the arguments will cause a value of NA to be returned, otherwise NA values are ignored.

This is a generic function: methods can be defined for it directly or via the [Summary](#) group generic.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[sum](#), [cumprod](#), [cumsum](#).

**Examples**

```
print(prod(1:7)) == print(gamma(8))
```

---

prop.table	<i>Express table entries as fraction of marginal table</i>
------------	--

---

**Description**

This is really `sweep(x, margin, margin.table(x, margin), "/")` for newbies, except that if `margin` has length zero, then one gets `x/sum(x)`.

**Usage**

```
prop.table(x, margin=NULL)
```

**Arguments**

<code>x</code>	table
<code>margin</code>	index, or vector of indices to generate margin for

**Value**

Table like `x` expressed relative to `margin`

**Author(s)**

Peter Dalgaard

**See Also**

[margin.table](#)

**Examples**

```
m <- matrix(1:4,2)
m
prop.table(m,1)
```

---

pushBack	<i>Push Text Back on to a Connection</i>
----------	--

---

### Description

Functions to push back text lines onto a connection, and to enquire how many lines are currently pushed back.

### Usage

```
pushBack(data, connection, newLine = TRUE)
pushBackLength(connection)
```

### Arguments

data	a character vector.
connection	A connection.
newLine	logical. If true, a newline is appended to each string pushed back.

### Details

Several character strings can be pushed back on one or more occasions. The occasions form a stack, so the first line to be retrieved will be the first string from the last call to `pushBack`. Lines which are pushed back are read prior to the normal input from the connection, by the normal text-reading functions such as `readLines` and `scan`.

Pushback is only allowed for readable connections.

Not all uses of connections respect pushbacks, in particular the input connection is still wired directly, so for example parsing commands from the console and `scan(" ")` ignore pushbacks on `stdin`.

### Value

`pushBack` returns nothing.

`pushBackLength` returns number of lines currently pushed back.

### See Also

[connections](#), [readLines](#).

### Examples

```
zz <- textConnection(LETTERS)
readLines(zz, 2)
pushBack(c("aa", "bb"), zz)
pushBackLength(zz)
readLines(zz, 1)
pushBackLength(zz)
```

```
readLines(zz, 1)
readLines(zz, 1)
close(zz)
```

---

qr

---

*The QR Decomposition of a Matrix*


---

### Description

qr computes the QR decomposition of a matrix. It provides an interface to the techniques used in the LINPACK routine DQRDC or the LAPACK routines DGEQP3 and (for complex matrices) ZGEQP3.

### Usage

```
qr(x, tol = 1e-07 , LAPACK = FALSE)
qr.coef(qr, y)
qr.qy(qr, y)
qr.qty(qr, y)
qr.resid(qr, y)
qr.fitted(qr, y, k = qr$rank)
qr.solve(a, b, tol = 1e-7)
## S3 method for class 'qr':
solve(a, b, ...)

is.qr(x)
as.qr(x)
```

### Arguments

x	a matrix whose QR decomposition is to be computed.
tol	the tolerance for detecting linear dependencies in the columns of x. Only used if LAPACK is false and x is real.
qr	a QR decomposition of the type computed by qr.
y, b	a vector or matrix of right-hand sides of equations.
a	A QR decomposition or (qr.solve only) a rectangular matrix.
k	effective rank.
LAPACK	logical. For real x, if true use LAPACK otherwise use LINPACK.
...	further arguments passed to or from other methods

## Details

The QR decomposition plays an important role in many statistical techniques. In particular it can be used to solve the equation  $Ax = b$  for given matrix  $A$ , and vector  $b$ . It is useful for computing regression coefficients and in applying the Newton-Raphson algorithm.

The functions `qr.coef`, `qr.resid`, and `qr.fitted` return the coefficients, residuals and fitted values obtained when fitting  $y$  to the matrix with QR decomposition `qr`. `qr.qy` and `qr.qty` return  $Q \%*\% y$  and  $t(Q) \%*\% y$ , where  $Q$  is the  $Q$  matrix.

All the above functions keep `dimnames` (and `names`) of  $x$  and  $y$  if there are.

`solve.qr` is the method for `solve` for `qr` objects. `qr.solve` solves systems of equations via the QR decomposition: if  $a$  is a QR decomposition it is the same as `solve.qr`, but if  $a$  is a rectangular matrix the QR decomposition is computed first. Either will handle over- and under-determined systems, providing a minimal-length solution or a least-squares fit if appropriate.

`is.qr` returns TRUE if  $x$  is a [list](#) with components named `qr`, `rank` and `qraux` and FALSE otherwise.

It is not possible to coerce objects to mode "qr". Objects either are QR decompositions or they are not.

## Value

The QR decomposition of the matrix as computed by LINPACK or LAPACK. The components in the returned value correspond directly to the values returned by DQRDC/DGEQP3/ZGEQP3.

<code>qr</code>	a matrix with the same dimensions as $x$ . The upper triangle contains the $R$ of the decomposition and the lower triangle contains information on the $Q$ of the decomposition (stored in compact form). Note that the storage used by DQRDC and DGEQP3 differs.
<code>qraux</code>	a vector of length <code>ncol(x)</code> which contains additional information on $Q$ .
<code>rank</code>	the rank of $x$ as computed by the decomposition: always full rank in the LAPACK case.
<code>pivot</code>	information on the pivoting strategy used during the decomposition.

Non-complex QR objects computed by LAPACK have the attribute "useLAPACK" with value TRUE.

## Note

To compute the determinant of a matrix (do you *really* need it?), the QR decomposition is much more efficient than using Eigen values ([eigen](#)). See [det](#).

Using LAPACK (including in the complex case) uses column pivoting and does not attempt to detect rank-deficient matrices.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.  
Available on-line at [http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html).

### See Also

`qr.Q`, `qr.R`, `qr.X` for reconstruction of the matrices. `solve.qr`, `lsfit`, `eigen`, `svd`.  
`det` (using `qr`) to compute the determinant of a matrix.

### Examples

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
h9 <- hilbert(9); h9
qr(h9)$rank          #--> only 7
qrh9 <- qr(h9, tol = 1e-10)
qrh9$rank            #--> 9
##-- Solve linear equation system H %**% x = y :
y <- 1:9/10
x <- qr.solve(h9, y, tol = 1e-10) # or equivalently :
x <- qr.coef(qrh9, y) #-- is == but much better than
                        #-- solve(h9) %**% y
h9 %**% x              # = y
```

---

QR.Auxiliaries

*Reconstruct the Q, R, or X Matrices from a QR Object*

---

### Description

Returns the original matrix from which the object was constructed or the components of the decomposition.

### Usage

```
qr.X(qr, complete = FALSE, ncol =)
qr.Q(qr, complete = FALSE, Dvec =)
qr.R(qr, complete = FALSE)
```

### Arguments

<code>qr</code>	object representing a QR decomposition. This will typically have come from a previous call to <code>qr</code> or <code>lsfit</code> .
<code>complete</code>	logical expression of length 1. Indicates whether an arbitrary orthogonal completion of the $Q$ or $X$ matrices is to be made, or whether the $R$ matrix is to be completed by binding zero-value rows beneath the square upper triangle.

ncol	integer in the range $1:nrow(qr\$qr)$ . The number of columns to be in the reconstructed $\mathbf{X}$ . The default when <code>complete</code> is <code>FALSE</code> is the first $\min(ncol(X), nrow(X))$ columns of the original $\mathbf{X}$ from which the <code>qr</code> object was constructed. The default when <code>complete</code> is <code>TRUE</code> is a square matrix with the original $\mathbf{X}$ in the first $ncol(X)$ columns and an arbitrary orthogonal completion (unitary completion in the complex case) in the remaining columns.
Dvec	vector (not matrix) of diagonal values. Each column of the returned $\mathbf{Q}$ will be multiplied by the corresponding diagonal value. Defaults to all 1s.

### Value

`qr.X` returns  $\mathbf{X}$ , the original matrix from which the `qr` object was constructed, provided  $ncol(X) \leq nrow(X)$ . If `complete` is `TRUE` or the argument `ncol` is greater than  $ncol(X)$ , additional columns from an arbitrary orthogonal (unitary) completion of  $\mathbf{X}$  are returned.

`qr.Q` returns  $\mathbf{Q}$ , the order- $nrow(X)$  orthogonal (unitary) transformation represented by `qr`. If `complete` is `TRUE`,  $\mathbf{Q}$  has  $nrow(X)$  columns. If `complete` is `FALSE`,  $\mathbf{Q}$  has  $ncol(X)$  columns. When `Dvec` is specified, each column of  $\mathbf{Q}$  is multiplied by the corresponding value in `Dvec`.

`qr.R` returns  $\mathbf{R}$ , the upper triangular matrix such that  $\mathbf{X} == \mathbf{Q} \%*\% \mathbf{R}$ . The number of rows of  $\mathbf{R}$  is  $nrow(X)$  or  $ncol(X)$ , depending on whether `complete` is `TRUE` or `FALSE`.

### See Also

[qr](#), [qr.qy](#).

### Examples

```
p <- ncol(x <- LifeCycleSavings[,-1]) # not the 'sr'
qrstr <- qr(x) # dim(x) == c(n,p)
qrstr $ rank # = 4 = p
Q <- qr.Q(qrstr) # dim(Q) == dim(x)
R <- qr.R(qrstr) # dim(R) == ncol(x)
X <- qr.X(qrstr) # X == x
range(X - as.matrix(x))# ~ < 6e-12
## X == Q %*% R :
Q %*% R
```

---

quit

*Terminate an R Session*


---

### Description

The function `quit` or its alias `q` terminate the current `R` session.

**Usage**

```
quit(save = "default", status = 0, runLast = TRUE)
  q(save = "default", status = 0, runLast = TRUE)
.Last <- function(x) { ..... }
```

**Arguments**

save	a character string indicating whether the environment (workspace) should be saved, one of "no", "yes", "ask" or "default".
status	the (numerical) error status to be returned to the operating system, where relevant. Conventionally 0 indicates successful completion.
runLast	should .Last() be executed?

**Details**

save must be one of "no", "yes", "ask" or "default". In the first case the workspace is not saved, in the second it is saved and in the third the user is prompted and can also decide *not* to quit. The default is to ask in interactive use but may be overridden by command-line arguments (which must be supplied in non-interactive use).

Immediately *before* terminating, the function .Last() is executed if it exists and runLast is true. If in interactive use there are errors in the .Last function, control will be returned to the command prompt, so do test the function thoroughly.

Some error statuses are used by R itself. The default error handler for non-interactive effectively calls q("no", 1, FALSE) and returns error code 1. Error status 2 is used for R 'suicide', that is a catastrophic failure, and other small numbers are used by specific ports for initialization failures. It is recommended that users choose statuses of 10 or more.

Valid values of status are system-dependent, but 0:255 are normally valid.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[.First](#) for setting things on startup.

**Examples**

```
## Not run:
## Unix-flavour example
.Last <- function() {
  cat("Now sending PostScript graphics to the printer:\n")
  system("lpr Rplots.ps")
  cat("bye bye...\n")
}
quit("yes")
## End(Not run)
```

---

R.home	<i>Return the R Home Directory</i>
--------	------------------------------------

---

**Description**

Return the R home directory.

**Usage**

```
R.home()
```

**Value**

A character string giving the current home directory.

---

R.Version	<i>Version Information</i>
-----------	----------------------------

---

**Description**

R.Version() provides detailed information about the version of R running.

R.version is a variable (a [list](#)) holding this information (and version is a copy of it for S compatibility), whereas R.version.string is a simple [character](#) string, useful for plotting, etc.

**Usage**

```
R.Version()
R.version
R.version.string
```

**Value**

R.Version returns a list with components

platform	the platform for which R was built. Under Unix, a triplet of the form CPU-VENDOR-OS, as determined by the configure script. E.g, "i586-unknown-linux".
arch	the architecture (CPU) R was built on/for.
os	the underlying operating system
system	CPU and OS.
status	the status of the version (e.g., "Alpha")
status.rev	the status revision level
major	the major version number

minor	the minor version number
year	the year the version was released
month	the month the version was released
day	the day the version was released
language	always "R".

**Note**

Do *not* use `R.version$os` to test the platform the code is running on: use `.Platform$OS.type` instead. Slightly different versions of the OS may report different values of `R.version$os`, as may different versions of R.

**See Also**

[getRversion](#) [.Platform](#).

**Examples**

```
R.version$os # to check how lucky you are ...
plot(0) # any plot
mtext(R.version.string, side=1,line=4,adj=1)# a useful bottom-right note
```

---

r2dtable

*Random 2-way Tables with Given Marginals*


---

**Description**

Generate random 2-way tables with given marginals using Patefield's algorithm.

**Usage**

```
r2dtable(n, r, c)
```

**Arguments**

n	a non-negative numeric giving the number of tables to be drawn.
r	a non-negative vector of length at least 2 giving the row totals, to be coerced to integer. Must sum to the same as c.
c	a non-negative vector of length at least 2 giving the column totals, to be coerced to integer.

**Value**

A list of length n containing the generated tables as its components.

## References

Patefield, W. M. (1981) Algorithm AS159. An efficient method of generating  $r \times c$  tables with given row and column totals. *Applied Statistics* **30**, 91–97.

## Examples

```
## Fisher's Tea Drinker data.
TeaTasting <-
matrix(c(3, 1, 1, 3),
       nr = 2,
       dimnames = list(Guess = c("Milk", "Tea"),
                       Truth = c("Milk", "Tea")))
## Simulate permutation test for independence based on the maximum
## Pearson residuals (rather than their sum).
rowTotals <- rowSums(TeaTasting)
colTotals <- colSums(TeaTasting)
nOfCases <- sum(rowTotals)
expected <- outer(rowTotals, colTotals, "*") / nOfCases
maxSqResid <- function(x) max((x - expected) ^ 2 / expected)
simMaxSqResid <-
  sapply(r2dtable(1000, rowTotals, colTotals), maxSqResid)
sum(simMaxSqResid >= maxSqResid(TeaTasting)) / 1000
## Fisher's exact test gives p = 0.4857 ...
```

---

 Random

*Random Number Generation*


---

## Description

`.Random.seed` is an integer vector, containing the random number generator (RNG) **state** for random number generation in R. It can be saved and restored, but should not be altered by the user.

`RNGkind` is a more friendly interface to query or set the kind of RNG in use.

`RNGversion` can be used to set the random generators as they were in an earlier R version (for reproducibility).

`set.seed` is the recommended way to specify seeds.

## Usage

```
.Random.seed <- c(rng.kind, n1, n2, ...)
save.seed <- .Random.seed

RNGkind(kind = NULL, normal.kind = NULL)
RNGversion(vstr)
set.seed(seed, kind = NULL)
```

**Arguments**

<code>kind</code>	character or NULL. If <code>kind</code> is a character string, set R's RNG to the kind desired. If it is NULL, return the currently used RNG. Use "default" to return to the R default.
<code>normal.kind</code>	character string or NULL. If it is a character string, set the method of Normal generation. Use "default" to return to the R default.
<code>seed</code>	a single value, interpreted as an integer.
<code>vstr</code>	a character string containing a version number, e.g., "1.6.2"
<code>rng.kind</code>	integer code in 0:k for the above <code>kind</code> .
<code>n1, n2, ...</code>	integers. See the details for how many are required (which depends on <code>rng.kind</code> ).

**Details**

The currently available RNG kinds are given below. `kind` is partially matched to this list. The default is "Mersenne-Twister".

**"Wichmann-Hill"** The seed, `.Random.seed[-1] == r[1:3]` is an integer vector of length 3, where each `r[i]` is in  $1:(p[i] - 1)$ , where `p` is the length 3 vector of primes, `p = (30269, 30307, 30323)`. The Wichmann-Hill generator has a cycle length of  $6.9536 \times 10^{12}$  ( $= \text{prod}(p-1)/4$ , see *Applied Statistics* (1984) **33**, 123 which corrects the original article).

**"Marsaglia-Multicarry"**: A *multiply-with-carry* RNG is used, as recommended by George Marsaglia in his post to the mailing list 'sci.stat.math'. It has a period of more than  $2^{60}$  and has passed all tests (according to Marsaglia). The seed is two integers (all values allowed).

**"Super-Duper"**: Marsaglia's famous Super-Duper from the 70's. This is the original version which does *not* pass the MTUPLE test of the Diehard battery. It has a period of  $\approx 4.6 \times 10^{18}$  for most initial seeds. The seed is two integers (all values allowed for the first seed: the second must be odd).

We use the implementation by Reeds et al. (1982-84).

The two seeds are the Tausworthe and congruence long integers, respectively. A one-to-one mapping to S's `.Random.seed[1:12]` is possible but we will not publish one, not least as this generator is **not** exactly the same as that in recent versions of S-PLUS.

**"Mersenne-Twister"**: From Matsumoto and Nishimura (1998). A twisted GFSR with period  $2^{19937} - 1$  and equidistribution in 623 consecutive dimensions (over the whole period). The "seed" is a 624-dimensional set of 32-bit integers plus a current position in that set.

**"Knuth-TAOCP"**: From Knuth (1997). A GFSR using lagged Fibonacci sequences with subtraction. That is, the recurrence used is

$$X_j = (X_{j-100} - X_{j-37}) \bmod 2^{30}$$

and the "seed" is the set of the 100 last numbers (actually recorded as 101 numbers, the last being a cyclic shift of the buffer). The period is around  $2^{129}$ .

**"Knuth-TAOCP-2002"**: The 2002 version which not backwards compatible with the earlier version: the initialization of the GFSR from the seed was altered. R did not allow you to choose consecutive seeds, the reported 'weakness', and already scrambled the seeds.

**"user-supplied"**: Use a user-supplied generator. See [Random.user](#) for details.

`normal.kind` can be "Kinderman-Ramage", "Buggy Kinderman-Ramage", "Ahrens-Dieter", "Box-Muller", "Inversion" (the default), or "user-supplied". (For inversion, see the reference in [qnorm](#).) The Kinderman-Ramage generator used in versions prior to 1.7.1 had several approximation errors and should only be used for reproduction of older results.

`set.seed` uses its single integer argument to set as many seeds as are required. It is intended as a simple way to get quite different seeds by specifying small integer arguments, and also as a way to get valid seed sets for the more complicated methods (especially "Mersenne-Twister" and "Knuth-TAOCP").

### Value

`.Random.seed` is an [integer](#) vector whose first element *codes* the kind of RNG and normal generator. The lowest two decimal digits are in  $0:(k-1)$  where  $k$  is the number of available RNGs. The hundreds represent the type of normal generator (starting at 0).

In the underlying C, `.Random.seed[-1]` is unsigned; therefore in R `.Random.seed[-1]` can be negative, due to the representation of an unsigned integer by a signed integer.

`RNGkind` returns a two-element character vector of the RNG and normal kinds in use *before* the call, invisibly if either argument is not NULL. `RNGversion` returns the same information.

`set.seed` returns NULL, invisibly.

### Note

Initially, there is no seed; a new one is created from the current time when one is required. Hence, different sessions will give different simulation results, by default.

`.Random.seed` saves the seed set for the uniform random-number generator, at least for the system generators. It does not necessarily save the state of other generators, and in particular does not save the state of the Box-Muller normal generator. If you want to reproduce work later, call `set.seed` rather than `set.Random.seed`.

As from R 1.8.0, `.Random.seed` is only looked for in the user's workspace.

All the supplied uniform generators return 32-bit integer values that are converted to doubles, so they take at most  $2^{32}$  distinct values and long runs will return duplicated values.

### Author(s)

of `RNGkind`: Martin Maechler. Current implementation, B. D. Ripley

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`set.seed`, storing in `.Random.seed`.)

Wichmann, B. A. and Hill, I. D. (1982) *Algorithm AS 183: An Efficient and Portable Pseudo-random Number Generator*, Applied Statistics, **31**, 188–190; Remarks: **34**, 198 and **35**, 89.

De Matteis, A. and Pagnutti, S. (1993) *Long-range Correlation Analysis of the Wichmann-Hill Random Number Generator*, Statist. Comput., **3**, 67–70.

Marsaglia, G. (1997) *A random number generator for C*. Discussion paper, posting on Usenet newsgroup `sci.stat.math` on September 29, 1997.

Reeds, J., Hubert, S. and Abrahams, M. (1982–4) C implementation of SuperDuper, University of California at Berkeley. (Personal communication from Jim Reeds to Ross Ihaka.)

Marsaglia, G. and Zaman, A. (1994) Some portable very-long-period random number generators. *Computers in Physics*, **8**, 117–121.

Matsumoto, M. and Nishimura, T. (1998) Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Transactions on Modeling and Computer Simulation*, **8**, 3–30.

Source code at <http://www.math.keio.ac.jp/~matumoto/emt.html>.

Knuth, D. E. (1997) *The Art of Computer Programming*. Volume 2, third edition.

Source code at <http://www-cs-faculty.stanford.edu/~knuth/taocp.html>.

Knuth, D. E. (2002) *The Art of Computer Programming*. Volume 2, third edition, ninth printing.

See <http://Sunburn.Stanford.EDU/~knuth/news02.html>.

Kinderman, A. J. and Ramage, J. G. (1976) Computer generation of normal random variables. *Journal of the American Statistical Association* **71**, 893–896.

Ahrens, J.H. and Dieter, U. (1973) Extensions of Forsythe's method for random sampling from the normal distribution. *Mathematics of Computation* **27**, 927–937.

Box, G.E.P. and Muller, M.E. (1958) A note on the generation of normal random deviates. *Annals of Mathematical Statistics* **29**, 610–611.

### See Also

[runif](#), [rnorm](#), ...

### Examples

```
## the default random seed is 626 integers, so only print a few
runif(1); .Random.seed[1:6]; runif(1); .Random.seed[1:6]
## If there is no seed, a "random" new one is created:
rm(.Random.seed); runif(1); .Random.seed[1:6]

RNGkind("Wich")# (partial string matching on 'kind')

## This shows how 'runif(.)' works for Wichmann-Hill,
## using only R functions:

p.WH <- c(30269, 30307, 30323)
a.WH <- c( 171,  172,  170)
next.WHseed <- function(i.seed = .Random.seed[-1])
  { (a.WH * i.seed) %% p.WH }
my.runif1 <- function(i.seed = .Random.seed)
  { ns <- next.WHseed(i.seed[-1]); sum(ns / p.WH) %% 1 }
rs <- .Random.seed
(WHs <- next.WHseed(rs[-1]))
u <- runif(1)
stopifnot(
  next.WHseed(rs[-1]) == .Random.seed[-1],
```

```

    all.equal(u, my.runif1(rs))
  )

## ----
.Random.seed
ok <- RNGkind()
RNGkind("Super")#matches "Super-Duper"
RNGkind()
.Random.seed # new, corresponding to Super-Duper

## Reset:
RNGkind(ok[1])

## ----
sum(duplicated(runif(1e6))) # around 110
## and we would expect about almost sure duplicates beyond about
qbirthday(1-1e-6, classes=2e9) # 235,000

```

---

Random.user

*User-supplied Random Number Generation*


---

## Description

Function `RNGkind` allows user-coded uniform and normal random number generators to be supplied. The details are given here.

## Details

A user-specified uniform RNG is called from entry points in dynamically-loaded compiled code. The user must supply the entry point `user_unif_rand`, which takes no arguments and returns a *pointer to a double*. The example below will show the general pattern.

Optionally, the user can supply the entry point `user_unif_init`, which is called with an `unsigned int` argument when `RNGkind` (or `set.seed`) is called, and is intended to be used to initialize the user's RNG code. The argument is intended to be used to set the “seeds”; it is the `seed` argument to `set.seed` or an essentially random seed if `RNGkind` is called.

If only these functions are supplied, no information about the generator's state is recorded in `.Random.seed`. Optionally, functions `user_unif_nseed` and `user_unif_seedloc` can be supplied which are called with no arguments and should return pointers to the number of “seeds” and to an integer array of “seeds”. Calls to `GetRNGstate` and `PutRNGstate` will then copy this array to and from `.Random.seed`.

A user-specified normal RNG is specified by a single entry point `user_norm_rand`, which takes no arguments and returns a *pointer to a double*.

## Warning

As with all compiled code, mis-specifying these functions can crash R. Do include the ‘`R_ext/Random.h`’ header file for type checking.

**Examples**

```

## Not run:
## Marsaglia's congruential PRNG
#include <R_ext/Random.h>

static Int32 seed;
static double res;
static int nseed = 1;

double * user_unif_rand()
{
  seed = 69069 * seed + 1;
  res = seed * 2.32830643653869e-10;
  return &res;
}

void user_unif_init(Int32 seed_in) { seed = seed_in; }
int * user_unif_nseed() { return &nseed; }
int * user_unif_seedloc() { return (int *) &seed; }

/* ratio-of-uniforms for normal */
#include <math.h>
static double x;

double * user_norm_rand()
{
  double u, v, z;
  do {
    u = unif_rand();
    v = 0.857764 * (2. * unif_rand() - 1);
    x = v/u; z = 0.25 * x * x;
    if (z < 1. - u) break;
    if (z > 0.259/u + 0.35) continue;
  } while (z > -log(u));
  return &x;
}

## Use under Unix:
R SHLIB urand.c
R
> dyn.load("urand.so")
> RNGkind("user")
> runif(10)
> .Random.seed
> RNGkind(, "user")
> rnorm(10)
> RNGkind()
[1] "user-supplied" "user-supplied"
## End(Not run)

```

---

range	<i>Range of Values</i>
-------	------------------------

---

### Description

range returns a vector containing the minimum and maximum of all the given arguments.

### Usage

```
range(..., na.rm = FALSE)

## Default S3 method:
range(..., na.rm = FALSE, finite = FALSE)
```

### Arguments

...	any <code>numeric</code> objects.
<code>na.rm</code>	logical, indicating if <code>NA</code> 's should be omitted.
<code>finite</code>	logical, indicating if all non-finite elements should be omitted.

### Details

range is a generic function: methods can be defined for it directly or via the [Summary](#) group generic.

If `na.rm` is `FALSE`, `NA` and `NaN` values in any of the arguments will cause `NA` values to be returned, otherwise `NA` values are ignored.

If `finite` is `TRUE`, the minimum and maximum of all finite values is computed, i.e., `finite=TRUE` *includes* `na.rm=TRUE`.

A special situation occurs when there is no (after omission of `NAs`) nonempty argument left, see [min](#).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[min](#), [max](#), [Methods](#).

**Examples**

```
(r.x <- range(rnorm(100)))
diff(r.x) # the SAMPLE range

x <- c(NA, 1:3, -1:1/0); x
range(x)
range(x, na.rm = TRUE)
range(x, finite = TRUE)
```

rank

*Sample Ranks***Description**

Returns the sample ranks of the values in a numeric vector. Ties, i.e., equal values, result in ranks being averaged, by default.

**Usage**

```
rank(x, na.last = TRUE,
     ties.method = c("average", "first", "random", "max", "min"))
```

**Arguments**

`x` a numeric vector.

`na.last` for controlling the treatment of *NAs*. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed; if `"keep"` they are kept.

`ties.method` a character string specifying how ties are treated, see below; can be abbreviated.

**Details**

If all components are different, the ranks are well defined, with values in  $1:n$  where  $n <- \text{length}(x)$  and we assume no *NAs* for the moment. Otherwise, with some values equal, called ‘ties’, the argument `ties.method` determines the result at the corresponding indices. The `"first"` method results in a permutation with increasing values at each index set of ties. The `"random"` method puts these in random order whereas the default, `"average"`, replaces them by their mean, and `"max"` and `"min"` replaces them by their maximum and minimum respectively, the latter being the typical “sports” ranking.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[order](#) and [sort](#).

**Examples**

```
(r1 <- rank(x1 <- c(3, 1, 4, 15, 92)))
x2 <- c(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)
names(x2) <- letters[1:11]
(r2 <- rank(x2)) # ties are averaged

## rank() is "idempotent": rank(rank(x)) == rank(x) :
stopifnot(rank(r1) == r1, rank(r2) == r2)

## ranks without averaging
rank(x2, ties.method= "first") # first occurrence wins
rank(x2, ties.method= "random") # ties broken at random
rank(x2, ties.method= "random") # and again

## keep ties ties, no average
(rma <- rank(x2, ties.method= "max")) # as used classically
(rmi <- rank(x2, ties.method= "min")) # as in Sports
stopifnot(rma + rmi == round(r2 + r2))
```

---

raw

*Raw Vectors*

---

**Description**

Creates or tests for objects of type "raw".

**Usage**

```
raw(length = 0)
as.raw(x)
```

**Arguments**

length	desired length.
x	object to be coerced.

**Details**

The raw type is intended to hold raw bytes. It is possible to extract subsequences of bytes, and to replace elements (but only by elements of a raw vector). The relational operators (see [Comparison](#)) work, as do the logical operators (see [Logic](#)) with a bitwise interpretation.

A raw vector is printed with each byte separately represented as a pair of hex digits. If you want to see a character representation (with escape sequences for non-printing characters) use [rawToChar](#).

**Value**

`raw` creates a raw vector of the specified length. Each element of the vector is equal to 0. Raw vectors are used to store fixed-length sequences of bytes.

`as.raw` attempts to coerce its argument to be of raw type. The (elementwise) answer will be 0 unless the coercion succeeds.

**Examples**

```
x <- "A test string"
(y <- charToRaw(x))
is.vector(y) # TRUE
rawToChar(y)
```

---

<code>rawConversion</code>	<i>Convert to or from Raw Vectors</i>
----------------------------	---------------------------------------

---

**Description**

Conversion and manipulation of objects of type "raw".

**Usage**

```
charToRaw(x)
rawToChar(x, multiple = FALSE)

rawShift(x, n)

rawToBits(x)
intToBits(x)
packBits(x, type = c("raw", "integer"))
```

**Arguments**

<code>x</code>	object to be converted or shifted.
<code>multiple</code>	logical: should the conversion be to a single character string or multiple individual characters?
<code>n</code>	the number of bits to shift. Positive numbers shift right and negative numbers shift left: allowed values are -8 ... 8.
<code>type</code>	the result type.

**Details**

`packBits` accepts raw, integer or logical inputs, the last two without an NAs.

**Value**

`charToRaw` converts a length-one character string to raw bytes.

`rawToChar` converts raw bytes either to a single character string or a character vector of single bytes. (Note that a single character string could contain embedded nuls.)

`rawToBits` returns a raw vector of 8 times the length of a raw vector with entries 0 or 1. `intToBits` returns a raw vector of 32 times the length of an integer vector with entries 0 or 1. In both cases the unpacking is least-significant bit first.

`packbits` packs its input (using only the lowest bit for raw or integer vectors) least-significant bit first to a raw or integer vector.

**Examples**

```
x <- "A test string"
(y <- charToRaw(x))
is.vector(y) # TRUE

rawToChar(y)
rawToChar(y, multiple = TRUE)
(xx <- c(y, as.raw(0), charToRaw("more")))
rawToChar(xx)
xxx <- xx
xxx[length(y)+1] <- charToRaw("&")
xxx
rawToChar(xxx)

rawShift(y, 1)
rawShift(y, -2)

rawToBits(y)
```

**Description**

Utilities for converting files in R documentation (Rd) format to other formats or create indices from them, and for converting documentation in other formats to Rd format.

**Usage**

```
R CMD Rdconv [options] file
R CMD Rd2dvi [options] files
R CMD Rd2txt [options] file
R CMD Sd2Rd [options] file
```

**Arguments**

<code>file</code>	the path to a file to be processed.
<code>files</code>	a list of file names specifying the R documentation sources to use, by either giving the paths to the files, or the path to a directory with the sources of a package.
<code>options</code>	further options to control the processing, or for obtaining information about usage and version of the utility.

**Details**

`Rdconv` converts Rd format to other formats. Currently, plain text, HTML, LaTeX, S version 3 (Sd), and S version 4 (.sgml) formats are supported. It can also extract the examples for run-time testing.

`Rd2dvi` and `Rd2txt` are user-level programs for producing DVI/PDF output or pretty text output from Rd sources.

`Sd2Rd` converts S (version 3 or 4) documentation formats to Rd format.

Use `R CMD foo --help` to obtain usage information on utility `foo`.

**Note**

Conversion to S version 3/4 formats is rough: there are some .Rd constructs for which there is no natural analogue. They are intended as a starting point for hand-tuning.

**See Also**

The chapter “Processing Rd format” in “Writing R Extensions” (see the ‘doc/manual’ subdirectory of the R source tree).

---

`read.table`

*Data Input*

---

**Description**

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

**Usage**

```
read.table(file, header = FALSE, sep = "", quote = "\"", dec = ".",
           row.names, col.names, as.is = FALSE, na.strings = "NA",
           colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#")
```

```

read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, ...)

read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=".",
         fill = TRUE, ...)

read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
         fill = TRUE, ...)

read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=".",
         fill = TRUE, ...)

```

### Arguments

file	<p>the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an <i>absolute</i> path, the file name is <i>relative</i> to the current working directory, <code>getwd()</code>. Tilde-expansion is performed where supported.</p> <p>Alternatively, <code>file</code> can be a <a href="#">connection</a>, which will be opened if necessary, and if so closed at the end of the function call. (If <code>stdin()</code> is used, the prompts for lines may be somewhat confusing. Terminate input with an EOF signal, <code>Ctrl-D</code> on Unix and <code>Ctrl-Z</code> on Windows.)</p> <p><code>file</code> can also be a complete URL.</p>
header	<p>a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: <code>header</code> is set to <code>TRUE</code> if and only if the first row contains one fewer field than the number of columns.</p>
sep	<p>the field separator character. Values on each line of the file are separated by this character. If <code>sep = ""</code> (the default for <code>read.table</code>) the separator is “white space”, that is one or more spaces, tabs or newlines.</p>
quote	<p>the set of quoting characters. To disable quoting altogether, use <code>quote = ""</code>. See <a href="#">scan</a> for the behaviour on quotes embedded in quotes.</p>
dec	<p>the character used in the file for decimal points.</p>
row.names	<p>a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names.</p> <p>If there is a header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if <code>row.names</code> is missing, the rows are numbered.</p> <p>Using <code>row.names = NULL</code> forces row numbering.</p>
col.names	<p>a vector of optional names for the variables. The default is to use "V" followed by the column number.</p>
as.is	<p>the default behavior of <code>read.table</code> is to convert character variables (which are not converted to logical, numeric or complex) to factors. The variable <code>as.is</code> controls the conversion of columns not otherwise specified by <code>colClasses</code>. Its value is either a vector of logicals (values are recycled if</p>

	necessary), or a vector of numeric or character indices which specify which columns should not be converted to factors.
	Note: to suppress all conversions including those of numeric columns, set <code>colClasses = "character"</code> .
	Note that <code>as.is</code> is specified per column (not per variable) and so includes the column of row names (if any) and any columns to be skipped.
<code>na.strings</code>	a vector of strings which are to be interpreted as <code>NA</code> values. Blank fields are also considered to be missing values.
<code>colClasses</code>	character. A vector of classes to be assumed for the columns. Recycled as necessary, or if the character vector is named, unspecified values are taken to be <code>NA</code> .  Possible values are <code>NA</code> (when <code>type.convert</code> is used), <code>NULL</code> (when the column is skipped), one of the atomic vector classes (logical, integer, numeric, complex, character, raw), or <code>"factor"</code> , <code>"Date"</code> or <code>"POSIXct"</code> . Otherwise there needs to be an <code>as</code> method (from package <b>methods</b> ) for conversion from <code>"character"</code> to the specified formal class.  Note that <code>colClasses</code> is specified per column (not per variable) and so includes the column of row names (if any).
<code>nrows</code>	the maximum number of rows to read in. Negative values are ignored.
<code>skip</code>	the number of lines of the data file to skip before beginning to read data.
<code>check.names</code>	logical. If <code>TRUE</code> then the names of the variables in the data frame are checked to ensure that they are syntactically valid variable names. If necessary they are adjusted (by <code>make.names</code> ) so that they are, and also to ensure that there are no duplicates.
<code>fill</code>	logical. If <code>TRUE</code> then in case the rows have unequal length, blank fields are implicitly added. See Details.
<code>strip.white</code>	logical. Used only when <code>sep</code> has been specified, and allows the stripping of leading and trailing white space from <code>character</code> fields (numeric fields are always stripped). See <code>scan</code> for further details, remembering that the columns may include the row names.
<code>blank.lines.skip</code>	logical: if <code>TRUE</code> blank lines in the input are ignored.
<code>comment.char</code>	character: a character vector of length one containing a single character or an empty string. Use <code>" "</code> to turn off the interpretation of comments altogether.
<code>...</code>	Further arguments to <code>read.table</code> .

### Details

If `row.names` is not specified and the header line has one less entry than the number of columns, the first column is taken to be the row names. This allows data frames to be read in from the format in which they are printed. If `row.names` is specified and does not refer to the first column, that column is discarded from such files.

The number of data columns is determined by looking at the first five lines of input (or the whole file if it has less than five lines), or from the length of `col.names` if it is specified and is longer. This

could conceivably be wrong if `fill` or `blank.lines.skip` are true, so specify `col.names` if necessary.

`read.csv` and `read.csv2` are identical to `read.table` except for the defaults. They are intended for reading “comma separated value” files (‘.CSV’) or the variant used in countries that use a comma as decimal point and a semicolon as field separator. Similarly, `read.delim` and `read.delim2` are for reading delimited files, defaulting to the TAB character for the delimiter. Notice that `header = TRUE` and `fill = TRUE` in these variants.

The rest of the line after a comment character is skipped; quotes are not processed in comments. Complete comment lines are allowed provided `blank.lines.skip = TRUE`; however, comment lines prior to the header must have the comment character in the first non-blank column.

As from R 1.9.0 quoted fields with embedded newlines are supported except after a comment character.

### Value

A data frame ([data.frame](#)) containing a representation of the data in the file. Empty input is an error unless `col.names` is specified, when a 0-row data frame is returned: similarly giving just a header line if `header = TRUE` results in a 0-row data frame.

This function is the principal means of reading tabular data into R.

### Note

The columns referred to in `as.is` and `colClasses` include the column of row names (if any).

Less memory will be used if `colClasses` is specified as one of the six atomic vector classes.

Using `nrows`, even as a mild over-estimate, will help memory usage.

Using `comment.char = ""` will be appreciably faster.

`read.table` is not the right tool for reading large matrices, especially those with many columns: it is designed to read *data frames* which may have columns of very different classes. Use [scan](#) instead.

Prior to version 1.9.0, underscores were not valid in variable names, and code that relies on them being converted to dots will no longer work. The simplest workaround is to use `names(d) <- gsub("_", ".", names(d))`, or, avoiding the (small) risk of creating duplicate names, `names(d) <- make.names(gsub("_", ".", names(d)), unique=TRUE)`.

### References

Chambers, J. M. (1992) *Data for models*. Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

### See Also

The *R Data Import/Export* manual.

[scan](#), [type.convert](#), [read.fwf](#) for reading *fixed width formatted* input; [write.table](#); [data.frame](#).

[count.fields](#) can be useful to determine problems with reading files which result in reports of incorrect record lengths.

readBin

*Transfer Binary Data To and From Connections***Description**

Read binary data from a connection, or write binary data to a connection.

**Usage**

```
readBin(con, what, n = 1, size = NA, signed = TRUE,
        endian = .Platform$endian)
writeBin(object, con, size = NA, endian = .Platform$endian)

readChar(con, nchars)
writeChar(object, con, nchars = nchar(object), eos = "")
```

**Arguments**

con	A connection object or a character string.
what	Either an object whose mode will give the mode of the vector to be read, or a character vector of length one describing the mode: one of "numeric", "double", "integer", "int", "logical", "complex", "character", "raw".
n	integer. The (maximal) number of records to be read. You can use an overestimate here, but not too large as storage is reserved for n items.
size	integer. The number of bytes per element in the byte stream. The default, NA, uses the natural size. Size changing is not supported for raw and complex vectors.
signed	logical. Only used for integers of sizes 1 and 2, when it determines if the quantity on file should be regarded as a signed or unsigned integer.
endian	The endian-ness ("big" or "little" of the target system for the file. Using "swap" will force swapping endian-ness.
object	An R object to be written to the connection.
nchars	integer, giving the lengths of (unterminated) character strings to be read or written.
eos	character. The terminator to be written after each string, followed by an ASCII nul; use NULL for no terminator at all.

**Details**

If the con is a character string, the functions call `file` to obtain an file connection which is opened for the duration of the function call.

If the connection is open it is read/written from its current position. If it is not open, it is opened for the duration of the call and then closed again.

If `size` is specified and not the natural size of the object, each element of the vector is coerced to an appropriate type before being written or as it is read. Possible sizes are 1, 2, 4 and possibly 8 for integer or logical vectors, and 4, 8 and possibly 12/16 for numeric vectors. (Note that coercion occurs as signed types except if `signed = FALSE` when reading integers of sizes 1 and 2.) Changing sizes is unlikely to preserve NAs, and the extended precision sizes are unlikely to be portable across platforms.

`readBin` and `writeBin` read and write C-style zero-terminated character strings. Input strings are limited to 10000 characters. `readChar` and `writeChar` allow more flexibility, and can also be used on text-mode connections.

Handling R's missing and special (`Inf`, `-Inf` and `NaN`) values is discussed in the *R Data Import/Export* manual.

### Value

For `readBin`, a vector of appropriate mode and length the number of items read (which might be less than `n`).

For `readChar`, a character vector of length the number of items read (which might be less than `length(nchars)`).

For `writeBin` and `writeChar`, none.

### Note

Integer read/writes of size 8 will be available if either C type `long` is of size 8 bytes or C type `long long` exists and is of size 8 bytes.

Real read/writes of size `sizeof(long double)` (usually 12 or 16 bytes) will be available only if that type is available and different from `double`.

As from R 2.0.0, character strings containing ASCII `\0` will be read correctly by `readChar` and appear as embedded nuls in the character vector returned. (In earlier versions of R they terminated the strings returned.)

If the character length requested for `readChar` is longer than the data available on the connection, what is available is returned. For `writeChar` if too many characters are requested the output is zero-padded, with a warning.

If `readBin(what=character())` is used incorrectly on a file which does not contain C-style character strings, warnings (usually many) are given as from version 1.6.2. The input will be broken into pieces of length 10000 with any final part being discarded.

### See Also

The *R Data Import/Export* manual.

[connections](#), [readLines](#), [writeLines](#).

[.Machine](#) for the sizes of `long`, `long long` and `long double`.

**Examples**

```

zz <- file("testbin", "wb")
writeBin(1:10, zz)
writeBin(pi, zz, endian="swap")
writeBin(pi, zz, size=4)
writeBin(pi^2, zz, size=4, endian="swap")
writeBin(pi+3i, zz)
writeBin("A test of a connection", zz)
z <- paste("A very long string", 1:100, collapse=" + ")
writeBin(z, zz)
if(.Machine$sizeof.long == 8 || .Machine$sizeof.longlong == 8)
  writeBin(as.integer(5^(1:10)), zz, size = 8)
if((s <- .Machine$sizeof.longdouble) > 8)
  writeBin((pi/3)^(1:10), zz, size = s)
close(zz)

zz <- file("testbin", "rb")
readBin(zz, integer(), 4)
readBin(zz, integer(), 6)
readBin(zz, numeric(), 1, endian="swap")
readBin(zz, numeric(), size=4)
readBin(zz, numeric(), size=4, endian="swap")
readBin(zz, complex(), 1)
readBin(zz, character(), 1)
z2 <- readBin(zz, character(), 1)
if(.Machine$sizeof.long == 8 || .Machine$sizeof.longlong == 8)
  readBin(zz, integer(), 10, size = 8)
if((s <- .Machine$sizeof.longdouble) > 8)
  readBin(zz, numeric(), 10, size = s)
close(zz)
unlink("testbin")
stopifnot(z2 == z)

## test fixed-length strings
zz <- file("testbin", "wb")
x <- c("a", "this will be truncated", "abc")
nc <- c(3, 10, 3)
writeChar(x, zz, nc, eos=NULL)
writeChar(x, zz, eos="\r\n")
close(zz)

zz <- file("testbin", "rb")
readChar(zz, nc)
readChar(zz, nchar(x)+3) # need to read the terminator explicitly
close(zz)
unlink("testbin")

## signed vs unsigned ints
zz <- file("testbin", "wb")
x <- as.integer(seq(0, 255, 32))
writeBin(x, zz, size=1)
writeBin(x, zz, size=1)

```

```
x <- as.integer(seq(0, 60000, 10000))
writeBin(x, zz, size=2)
writeBin(x, zz, size=2)
close(zz)
zz <- file("testbin", "rb")
readBin(zz, integer(), 8, size=1)
readBin(zz, integer(), 8, size=1, signed=FALSE)
readBin(zz, integer(), 7, size=2)
readBin(zz, integer(), 7, size=2, signed=FALSE)
close(zz)
unlink("testbin")
```

---

readline

*Read a Line from the Terminal*

---

### Description

readline reads a line from the terminal

### Usage

```
readline(prompt = " ")
```

### Arguments

**prompt**            the string printed when prompting the user for input. Should usually end with a space " ".

### Details

The prompt string will be truncated to a maximum allowed length, normally 256 chars (but can be changed in the source code).

### Value

A character vector of length one.

### Examples

```
fun <- function() {
  ANSWER <- readline("Are you a satisfied R user? ")
  if (substr(ANSWER, 1, 1) == "n")
    cat("This is impossible. YOU LIED!\n")
  else
    cat("I knew it.\n")
}
fun()
```

---

readLines	<i>Read Text Lines from a Connection</i>
-----------	--

---

### Description

Read text lines from a connection.

### Usage

```
readLines(con = stdin(), n = -1, ok = TRUE)
```

### Arguments

con	A connection object or a character string.
n	integer. The (maximal) number of lines to read. Negative values indicate that one should read up to the end of the connection.
ok	logical. Is it OK to reach the end of the connection before $n > 0$ lines are read? If not, an error will be generated.

### Details

If the `con` is a character string, the functions call `file` to obtain an file connection which is opened for the duration of the function call.

If the connection is open it is read from its current position. If it is not open, it is opened for the duration of the call and then closed again.

If the final line is incomplete (no final EOL marker) the behaviour depends on whether the connection is blocking or not. For a blocking text-mode connection (or a non-text-mode connection) the line will be accepted, with a warning. For a non-blocking text-mode connection the incomplete line is pushed back, silently.

Whatever mode the connection is opened in, any of LF, CRLF or CR will be accepted as the EOL marker for a line.

### Value

A character vector of length the number of lines read.

### See Also

[connections](#), [writeLines](#), [readBin](#), [scan](#)

**Examples**

```
cat("TITLE extra line", "2 3 5 7", "", "11 13 17", file="ex.data",
    sep="\n")
readLines("ex.data", n=-1)
unlink("ex.data") # tidy up

## difference in blocking
cat("123\nabc", file = "test1")
readLines("test1") # line with a warning

con <- file("test1", "r", blocking = FALSE)
readLines(con) # empty
cat(" def\n", file = "test1", append = TRUE)
readLines(con) # gets both
close(con)

unlink("test1") # tidy up
```

---

real

*Real Vectors*

---

**Description**

`real` creates a double precision vector of the specified length. Each element of the vector is equal to 0.

`as.real` attempts to coerce its argument to be of real type.

`is.real` returns TRUE or FALSE depending on whether its argument is of real type or not.

**Usage**

```
real(length = 0)
as.real(x, ...)
is.real(x)
```

**Arguments**

<code>length</code>	desired length.
<code>x</code>	object to be coerced or tested.
<code>...</code>	further arguments passed to or from other methods.

**Note**

R has no single precision data type. All real numbers are stored in double precision format.

---

`Recall`*Recursive Calling*

---

**Description**

`Recall` is used as a placeholder for the name of the function in which it is called. It allows the definition of recursive functions which still work after being renamed, see example below.

**Usage**

```
Recall(...)
```

**Arguments**

... all the arguments to be passed.

**See Also**

[do.call](#) and [call](#).

**Examples**

```
## A trivial (but inefficient!) example:
fib <- function(n) if(n<=2) {if(n>=0) 1 else 0} else Recall(n-1) + Recall(n-2)
fibonacci <- fib; rm(fib)
## renaming wouldn't work without Recall
fibonacci(10) # 55
```

---

`reg.finalizer`*Finalization of objects*

---

**Description**

Registers an R function to be called upon garbage collection of object.

**Usage**

```
reg.finalizer(e, f)
```

**Arguments**

`e` Object to finalize. Must be environment or external pointer.  
`f` Function to call on finalization. Must accept a single argument, which will be the object to finalize.

**Value**

NULL.

**Note**

The purpose of this function is mainly to allow objects that refer to external items (a temporary file, say) to perform cleanup actions when they are no longer referenced from within R. This only makes sense for objects that are never copied on assignment, hence the restriction to environments and external pointers.

**See Also**

[gc](#) and [Memory](#) for garbage collection and memory management.

**Examples**

```
f <- function(e) print("cleaning...")
g <- function(x){ e <- environment(); reg.finalizer(e,f) }
g()
invisible(gc()) # trigger cleanup
```

---

regex

*Regular Expressions as used in R*

---

**Description**

This help page documents the regular expression patterns supported by [grep](#) and related functions [regexr](#), [sub](#) and [gsub](#), as well as by [strsplit](#).

**Details**

A ‘regular expression’ is a pattern that describes a set of strings. Three types of regular expressions are used in R, *extended* regular expressions, used by `grep(extended = TRUE)` (its default), *basic* regular expressions, as used by `grep(extended = FALSE)`, and *Perl-like* regular expressions used by `grep(perl = TRUE)`.

Other functions which use regular expressions (often via the use of `grep`) include `apropos`, `browseEnv`, `help.search`, `list.files`, `ls` and `strsplit`. These will all use *extended* regular expressions, unless `strsplit` is called with argument `extended = FALSE` or `perl = TRUE`.

Patterns are described here as they would be printed by `cat`: do remember that backslashes need to be doubled in entering R character strings from the keyboard.

## Extended Regular Expressions

This section covers the regular expressions allowed if `extended = TRUE` in `grep`, `regexpr`, `sub`, `gsub` and `strsplit`. They use the GNU implementation of the POSIX 1003.2 standard.

Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash. The metacharacters are `.` `\` `|` `(` `)` `[` `{` `^` `$` `*` `+` `?`.

A *character class* is a list of characters enclosed by `[` and `]` matches any single character in that list; if the first character of the list is the caret `^`, then it matches any character *not* in the list. For example, the regular expression `[0123456789]` matches any single digit, and `[^abc]` matches anything except the characters `a`, `b` or `c`. A range of characters may be specified by giving the first and last characters, separated by a hyphen. (Character ranges are interpreted in the collation order of the current locale.)

Certain named classes of characters are predefined. Their interpretation depends on the *locale* (see [locales](#)); the interpretation below is that of the POSIX locale.

**[ :alnum: ]** Alphanumeric characters: `[ :alpha: ]` and `[ :digit: ]`.

**[ :alpha: ]** Alphabetic characters: `[ :lower: ]` and `[ :upper: ]`.

**[ :blank: ]** Blank characters: space and tab.

**[ :cntrl: ]** Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL). In another character set, these are the equivalent characters, if any.

**[ :digit: ]** Digits: 0 1 2 3 4 5 6 7 8 9.

**[ :graph: ]** Graphical characters: `[ :alnum: ]` and `[ :punct: ]`.

**[ :lower: ]** Lower-case letters in the current locale.

**[ :print: ]** Printable characters: `[ :alnum: ]`, `[ :punct: ]` and space.

**[ :punct: ]** Punctuation characters: `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~`.

**[ :space: ]** Space characters: tab, newline, vertical tab, form feed, carriage return, and space.

**[ :upper: ]** Upper-case letters in the current locale.

**[ :xdigit: ]** Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f.

For example, `[[:alnum:]]` means `[0-9A-Za-z]`, except the latter depends upon the locale and the character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal `]`, place it first in the list. Similarly, to include a literal `^`, place it anywhere but first. Finally, to include a literal `-`, place it first or last. (Only these and `\` remain special inside character classes.)

The period `.` matches any single character. The symbol `\w` is documented to be synonym for `[[:alnum:]]` and `\W` is its negation. However, `\w` also matches underscore in the GNU `grep` code used in **R**.

The caret `^` and the dollar sign `$` are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols `\<` and `\>` respectively match the empty string at the beginning and end of a word. The symbol `\b` matches the empty string at the edge of a word, and `\B` matches the empty string provided it is not at the edge of a word.

A regular expression may be followed by one of several repetition quantifiers:

- ? The preceding item is optional and will be matched at most once.
- \* The preceding item will be matched zero or more times.
- + The preceding item will be matched one or more times.
- {*n*} The preceding item is matched exactly *n* times.
- {*n*,} The preceding item is matched *n* or more times.
- {*n*,*m*} The preceding item is matched at least *n* times, but not more than *m* times.

Repetition is greedy, so the maximal possible number of repeats is used.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator `|`; the resulting regular expression matches any string matching either subexpression. For example, `abba|cde` matches either the string `abba` or the string `cde`. Note that alternation does not work inside character classes, where `|` has its literal meaning.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference `\N`, where *N* is a single digit, matches the substring previously matched by the *N*th parenthesized subexpression of the regular expression.

The current code attempts to support traditional usage by assuming that `{` is not special if it would be the start of an invalid interval specification. (POSIX allows this behaviour as an extension but we advise users not to rely on it.)

### Basic Regular Expressions

This section covers the regular expressions allowed if `extended = FALSE` in `grep`, `regexpr`, `sub`, `gsub` and `strsplit`.

In basic regular expressions the metacharacters `?`, `+`, `{`, `|`, `(`, and `)` lose their special meaning; instead use the backslashed versions `\?`, `\+`, `\{`, `\|`, `\(`, and `\)`. Thus the metacharacters are `. \ [ ^ $ *`.

### Perl Regular Expressions

The `perl = TRUE` argument to `grep`, `regexpr`, `sub`, `gsub` and `strsplit` switches to the PCRE library that ‘implements regular expression pattern matching using the same syntax and semantics as Perl 5.6 or later, with just a few differences’.

For complete details please consult the man pages for PCRE, especially `man pcrepattern` and `man pcreapi` on your system or from the sources at <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>. If PCRE support was compiled from the sources within R, the PCRE version is 4.5 as described here (version  $\geq$  4.0 is required even if R is configured to use the system’s PCRE library).

All the regular expressions described for extended regular expressions are accepted except `\<` and `\>`: in Perl all backslashed metacharacters are alphanumeric and backslashed symbols always are interpreted as a literal character. `{` is not special if it would be the start of an invalid interval specification. There can be more than 9 backreferences.

The construct `(? . . .)` is used for Perl extensions in a variety of ways depending on what immediately follows the `?`.

Perl-like matching can work in several modes, set by the options `(?i)` (caseless, equivalent to Perl's `/i`), `(?m)` (multiline, equivalent to Perl's `/m`), `(?s)` (single line, so a dot matches all characters, even new lines: equivalent to Perl's `/s`) and `(?x)` (extended, whitespace data characters are ignored unless escaped and comments are allowed: equivalent to Perl's `/x`). These can be concatenated, so for example, `(?im)` sets caseless multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and to combine setting and unsetting such as `(?im-sx)`. These settings can be applied within patterns, and then apply to the remainder of the pattern. Additional options not in Perl include `(?U)` to set 'ungreedy' mode (so matching is minimal unless `?` is used, when it is greedy). Initially none of these options are set.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between `\Q` and `\E`. This is different from Perl in that `$` and `@` are handled as literals in `\Q . . \E` sequences in PCRE, whereas in Perl, `$` and `@` cause variable interpolation.

The escape sequences `\d`, `\s` and `\w` represent any decimal digit, space character and 'word' character (letter, digit or underscore in the current locale) respectively, and their upper-case versions represent their negation. Unlike POSIX and earlier versions of Perl and PCRE, vertical tab is not regarded as a whitespace character.

Escape sequence `\a` is BEL, `\e` is ESC, `\f` is FF, `\n` is LF, `\r` is CR and `\t` is TAB. In addition `\cx` is `ctrl-x` for any `x`, `\ddd` is the octal character `ddd` (for up to three digits unless interpretable as a backreference), and `\xhh` specifies a character in hex.

Outside a character class, `\b` matches a word boundary, `\B` is its negation, `\A` matches at start of a subject (even in multiline mode, unlike `^`), `\Z` matches at end of a subject or before newline at end, `\z` matches at end of a subject. and `\G` matches at first matching position in a subject. `\C` matches a single byte. including a newline.

The same repetition quantifiers as extended POSIX are supported. However, if a quantifier is followed by `?`, the match is 'ungreedy', that is as short as possible rather than as long as possible (unless the meanings are reversed by the `(?U)` option.)

The sequence `(?#` marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part at all in the pattern matching.

If the extended option is set, an unescaped `#` character outside a character class introduces a comment that continues up to the next newline character in the pattern.

The pattern `(?: . . .)` groups characters just as parentheses do but does not make a backreference.

Patterns `(?= . . .)` and `(?! . . .)` are zero-width positive and negative lookahead *assertions*: they match if an attempt to match the `. . .` forward from the current position would succeed (or not), but use up no characters in the string being processed. Patterns `(?<= . . .)` and `(?<! . . .)` are the lookbehind equivalents: they do not allow repetition quantifiers nor `\C` in `. . .`.

Named subpatterns, atomic grouping, possessive qualifiers and conditional and recursive patterns are not covered here.

**Author(s)**

This help page is based on the documentation of GNU `grep` 2.4.2, from which the C code used by R has been taken, the `pcre` man page from PCRE 3.9 and the `pcrpattern` man page from PCRE 4.4.

**See Also**

[grep](#), [apropos](#), [browseEnv](#), [help.search](#), [list.files](#), [ls](#) and [strsplit](#).

---

 remove

*Remove Objects from a Specified Environment*


---

**Description**

`remove` and `rm` can be used to remove objects. These can be specified successively as character strings, or in the character vector `list`, or through a combination of both. All objects thus specified will be removed.

If `envir` is `NULL` then the currently active environment is searched first.

If `inherits` is `TRUE` then parents of the supplied directory are searched until a variable with the given name is encountered. A warning is printed for each variable that is not found.

**Usage**

```
remove(..., list = character(0), pos = -1, envir = as.environment(pos),
        inherits = FALSE)
rm      (... , list = character(0), pos = -1, envir = as.environment(pos),
        inherits = FALSE)
```

**Arguments**

<code>...</code>	the objects to be removed, supplied individually and/or as a character vector
<code>list</code>	a character vector naming objects to be removed.
<code>pos</code>	where to do the removal. By default, uses the current environment. See the details for other possibilities.
<code>envir</code>	the <a href="#">environment</a> to use. See the details section.
<code>inherits</code>	should the enclosing frames of the environment be inspected?

**Details**

The `pos` argument can specify the environment from which to remove the objects in any of several ways: as an integer (the position in the [search](#) list); as the character string name of an element in the search list; or as an [environment](#) (including using [sys.frame](#) to access the currently active function calls). The `envir` argument is an alternative way to specify an environment, but is primarily there for back compatibility.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[ls, objects](#)

**Examples**

```
tmp <- 1:4
## work with tmp and cleanup
rm(tmp)

## Not run:
## remove (almost) everything in the working environment.
## You will get no warning, so don't do this unless you are really sure.
rm(list = ls())
## End(Not run)
```

---

 rep

*Replicate Elements of Vectors and Lists*


---

**Description**

rep replicates the values in x. It is a generic function, and the default method is described here. rep.int is a faster simplified version for the commonest case.

**Usage**

```
rep(x, times, ...)

## Default S3 method:
rep(x, times, length.out, each, ...)

rep.int(x, times)
```

**Arguments**

x	a vector (of any mode including a list) or a pairlist or a POSIXct or POSIXlt or date object.
times	optional non-negative integer. A vector giving the number of times to repeat each element if of length length(x), or to repeat the whole vector if of length 1.
length.out	optional integer. The desired length of the output vector.
each	optional integer. Each element of x is repeated each times.
...	further arguments to be passed to or from other methods.

**Details**

A least one of `times`, `length.out` and `each` must be specified, and normally exactly one is. If `length.out` is given, `times` is ignored. If `each` is specified with either of the other two, its replication is performed first, and then that implied by `times` or `length.out`.

If `times` consists of a single integer, the result consists of the values in `x` repeated this many times. If `times` is a vector of the same length as `x`, the result consists of `x[1]` repeated `times[1]` times, `x[2]` repeated `times[2]` times and so on.

`length.out` may be given in place of `times`, in which case `x` is repeated as many times as is necessary to create a vector of this length.

Non-integer values of `times` will be truncated towards zero. If `times` is a computed quantity it is prudent to add a small fuzz.

If `x` has length zero and `length.out` is supplied and is positive, the values are filled in using the extraction rules, that is by an NA of the appropriate class for an atomic vector (0 for raw vectors) and NULL for a list.

**Value**

A vector of the same class as `x`.

**Note**

If the original vector has names, these are also replicated and so will almost always contain duplicates. (In contrast, `S` strips the names.)

Function `rep.int` is a simple case handled by internal code, and provided as a separate function purely for `S` compatibility.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[seq](#), [sequence](#).

**Examples**

```
rep(1:4, 2)
rep(1:4, each = 2)      # not the same.
rep(1:4, c(2,2,2,2))   # same as second.
rep(1:4, c(2,1,2,1))
rep(1:4, each = 2, len = 4)  # first 4 only.
rep(1:4, each = 2, len = 10) # 8 integers plus two recycled 1's.
rep(1:4, each = 2, times = 3) # length 24, 3 complete replications

rep(1, 40*(1-.8)) # length 7 on most platforms
rep(1, 40*(1-.8)+1e-7) # better
```

```
## replicate a list
fred <- list(happy = 1:10, name = "squash")
rep(fred, 5)

# date-time objects
x <- .leap.seconds[1:3]
rep(x, 2)
rep(as.POSIXlt(x), rep(2, 3))
```

---

replace

*Replace Values in a Vector*

---

### Description

replace replaces the values in `x` with indexes given in `list` by those given in `values`. If necessary, the values in `values` are recycled.

### Usage

```
replace(x, list, values)
```

### Arguments

<code>x</code>	vector
<code>list</code>	an index vector
<code>values</code>	replacement values

### Value

A vector with the values replaced.

### Note

`x` is unchanged: remember to assign the result.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

rev

*Reverse Elements*

---

### Description

rev provides a reversed version of its argument. It is generic function with a default method for vectors and one for [dendrograms](#).

Note that this is no longer needed (nor efficient) for obtaining vectors sorted into descending order, since that is now rather more directly achievable by `sort(x, decreasing = TRUE)`.

### Usage

```
rev(x)
```

### Arguments

x                    a vector or another object for which reversion is defined.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[seq](#), [sort](#).

### Examples

```
x <- c(1:5,5:3)
## sort into descending order; first more efficiently:
stopifnot(sort(x, decreasing = TRUE) == rev(sort(x)))
stopifnot(rev(1:7) == 7:1)#- don't need 'rev' here
```

---

rle

*Run Length Encoding*

---

### Description

Compute the lengths and values of runs of equal values in a vector – or the reverse operation.

### Usage

```
rle(x)
inverse.rle(x, ...)
```

**Arguments**

`x` a simple vector for `rle()` or an object of class "rle" for `inverse.rle()`.  
`...` further arguments which are ignored in R.

**Value**

`rle()` returns an object of class "rle" which is a list with components

`lengths` an integer vector containing the length of each run.

`values` a vector of the same length as `lengths` with the corresponding values.

`inverse.rle()` is the inverse function of `rle()`.

**Examples**

```
x <- rev(rep(6:10, 1:5))
rle(x)
## lengths [1:5]  5 4 3 2 1
## values  [1:5] 10 9 8 7 6

z <- c(TRUE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE, TRUE, TRUE)
rle(z)
rle(as.character(z))

stopifnot(x == inverse.rle(rle(x)),
          z == inverse.rle(rle(z)))
```

---

Round

*Rounding of Numbers*


---

**Description**

`ceiling` takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

`floor` takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the corresponding elements of `x`.

`round` rounds the values in its first argument to the specified number of decimal places (default 0). Note that for rounding off a 5, the IEEE standard is used, "go to the even digit". Therefore `round(0.5)` is 0 and `round(-1.5)` is -2.

`signif` rounds the values in its first argument to the specified number of significant digits.

`trunc` takes a single numeric argument `x` and returns a numeric vector containing the integers by truncating the values in `x` toward 0.

`zapsmall` determines a `digits` argument `dr` for calling `round(x, digits = dr)` such that values "close to zero" (compared with the maximal absolute one) are "zapped", i.e., treated as 0.

**Usage**

```
ceiling(x)
floor(x)
round(x, digits = 0)
signif(x, digits = 6)
trunc(x)
zapsmall(x, digits = getOption("digits"))
```

**Arguments**

`x` a numeric vector.

`digits` integer indicating the precision to be used.

**Details**

All but `zapsmall` are generic functions: methods can be defined for them individually or via the [Math](#) group generic.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (except `zapsmall`.)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. (`zapsmall`.)

**See Also**

[as.integer](#).

**Examples**

```
round(.5 + -2:4) # IEEE rounding: -2 0 0 2 2 4 4
( x1 <- seq(-2, 4, by = .5) )
round(x1)#-- IEEE rounding !
x1[trunc(x1) != floor(x1)]
x1[round(x1) != floor(x1 + .5)]
(non.int <- ceiling(x1) != floor(x1))

x2 <- pi * 100^(-1:3)
round(x2, 3)
signif(x2, 3)

print (x2 / 1000, digits=4)
zapsmall(x2 / 1000, digits=4)
zapsmall(exp(1i*0:4*pi/2))
```

---

round.POSIXt                      *Round / Truncate Data-Time Objects*

---

## Description

Round or truncate date-time objects.

## Usage

```
## S3 method for class 'POSIXt':
round(x, units = c("secs", "mins", "hours", "days"))
## S3 method for class 'POSIXt':
trunc(x, units = c("secs", "mins", "hours", "days"))

## S3 method for class 'Date':
round(x, ...)
## S3 method for class 'Date':
trunc(x)
```

## Arguments

`x`                      an object inheriting from "POSIXt" or "Date".  
`units`                    one of the units listed. Can be abbreviated.  
`...`                    arguments to be passed to or from other methods, notably `digits`.

## Details

The time is rounded or truncated to the second, minute, hour or day. Timezones are only relevant to days, when midnight in the current timezone is used.

The methods for class "Date" are of little use except to remove fractional days.

## Value

An object of class "POSIXlt".

## See Also

[round](#) for the generic function and default methods.

[DateTimeClasses](#), [Date](#)

## Examples

```
round(.leap.seconds + 1000, "hour")
trunc(Sys.time(), "day")
```

---

`row`*Row Indexes*

---

**Description**

Returns a matrix of integers indicating their row number in the matrix.

**Usage**

```
row(x, as.factor = FALSE)
```

**Arguments**

<code>x</code>	a matrix.
<code>as.factor</code>	a logical value indicating whether the value should be returned as a factor rather than as numeric.

**Value**

An integer matrix with the same dimensions as `x` and whose  $i$   $j$ -th element is equal to  $i$ .

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`col` to get columns.

**Examples**

```
x <- matrix(1:12, 3, 4)
# extract the diagonal of a matrix
dx <- x[row(x) == col(x)]
dx

# create an identity 5-by-5 matrix
x <- matrix(0, nr = 5, nc = 5)
x[row(x) == col(x)] <- 1
x
```

---

`row.names`*Get and Set Row Names for Data Frames*

---

**Description**

All data frames have a row names attribute, a character vector of length the number of rows with no duplicates nor missing values.

For convenience, these are generic functions for which users can write other methods, and there are default methods for arrays. The description here is for the `data.frame` method.

**Usage**

```
row.names(x)
row.names(x) <- value
```

**Arguments**

<code>x</code>	object of class "data.frame", or any other class for which a method has been defined.
<code>value</code>	a vector with the same length as the number of rows of <code>x</code> , to be coerced to character. Duplicated or missing values are not allowed.

**Value**

`row.names` returns a character vector.

`row.names<-` returns a data frame with the row names changed.

**Note**

`row.names` is similar to [rownames](#) for arrays, and it has a method that calls `rownames` for an array argument.

**References**

Chambers, J. M. (1992) *Data for models*. Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[data.frame](#), [rownames](#).

---

row/colnames	<i>Row and Column Names</i>
--------------	-----------------------------

---

## Description

Retrieve or set the row or column names of a matrix-like object.

## Usage

```
rownames(x, do.NULL = TRUE, prefix = "row")
rownames(x) <- value

colnames(x, do.NULL = TRUE, prefix = "col")
colnames(x) <- value
```

## Arguments

<code>x</code>	a matrix-like R object, with at least two dimensions for <code>colnames</code> .
<code>do.NULL</code>	logical. Should this create names if they are NULL?
<code>prefix</code>	for created names.
<code>value</code>	a valid value for that component of <code>dimnames(x)</code> . For a matrix or array this is either NULL or a character vector of length the appropriate dimension.

## Details

The extractor functions try to do something sensible for any matrix-like object `x`. If the object has `dimnames` the first component is used as the row names, and the second component (if any) is used for the col names. For a data frame, `rownames` and `colnames` are equivalent to `row.names` and `names` respectively.

If `do.NULL` is `FALSE`, a character vector (of length `NROW(x)` or `NCOL(x)`) is returned in any case, prepending `prefix` to simple numbers, if there are no `dimnames` or the corresponding component of the `dimnames` is NULL.

For a data frame, `value` for `rownames` should be a character vector of unique names, and for `colnames` a character vector of unique syntactically-valid names. (Note: uniqueness and validity are not enforced.)

## See Also

[dimnames](#), [case.names](#), [variable.names](#).

## Examples

```
m0 <- matrix(NA, 4, 0)
rownames(m0)

m2 <- cbind(1, 1:4)
```

```
colnames(m2, do.NULL = FALSE)
colnames(m2) <- c("x", "Y")
rownames(m2) <- rownames(m2, do.NULL = FALSE, prefix = "Obs.")
m2
```

---

rowsum

*Give row sums of a matrix or data frame, based on a grouping variable*


---

## Description

Compute sums across rows of a matrix-like object for each level of a grouping variable. `rowsum` is generic, with methods for matrices and data frames.

## Usage

```
rowsum(x, group, reorder = TRUE, ...)
```

## Arguments

<code>x</code>	a matrix, data frame or vector of numeric data. Missing values are allowed.
<code>group</code>	a vector giving the grouping, with one element per row of <code>x</code> . Missing values will be treated as another group and a warning will be given
<code>reorder</code>	if <code>TRUE</code> , then the result will be in order of <code>sort(unique(group))</code> , if <code>FALSE</code> , it will be in the order that rows were encountered.
<code>...</code>	other arguments for future methods

## Details

The default is to reorder the rows to agree with `tapply` as in the example below. Reordering should not add noticeably to the time except when there are very many distinct values of `group` and `x` has few columns.

The original function was written by Terry Therneau, but this is a new implementation using hashing that is much faster for large matrices.

To add all the rows of a matrix (ie, a single `group`) use `rowSums`, which should be even faster.

## Value

a matrix or data frame containing the sums. There will be one row per unique value of `group`.

## See Also

[tapply](#), [aggregate](#), [rowSums](#)

**Examples**

```
x <- matrix(runif(100), ncol=5)
group <- sample(1:8, 20, TRUE)
xsum <- rowsum(x, group)
## Slower versions
xsum2 <- tapply(x, list(group[row(x)], col(x)), sum)
xsum3 <- aggregate(x, list(group), sum)
```

sample

*Random Samples and Permutations***Description**

`sample` takes a sample of the specified size from the elements of `x` using either with or without replacement.

**Usage**

```
sample(x, size, replace = FALSE, prob = NULL)
```

**Arguments**

<code>x</code>	Either a (numeric, complex, character or logical) vector of more than one element from which to choose, or a positive integer.
<code>size</code>	non-negative integer giving the number of items to choose.
<code>replace</code>	Should sampling be with replacement?
<code>prob</code>	A vector of probability weights for obtaining the elements of the vector being sampled.

**Details**

If `x` has length 1, sampling takes place from `1:x`. *Note* that this convenience feature may lead to undesired behaviour when `x` is of varying length `sample(x)`. See the `resample()` example below.

By default `size` is equal to `length(x)` so that `sample(x)` generates a random permutation of the elements of `x` (or `1:x`).

The optional `prob` argument can be used to give a vector of weights for obtaining the elements of the vector being sampled. They need not sum to one, but they should be nonnegative and not all zero. If `replace` is false, these probabilities are applied sequentially, that is the probability of choosing the next item is proportional to the probabilities amongst the remaining items. The number of nonzero weights must be at least `size` in this case.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```

x <- 1:12
# a random permutation
sample(x)
# bootstrap sampling -- only if length(x) > 1 !
sample(x,replace=TRUE)

# 100 Bernoulli trials
sample(c(0,1), 100, replace = TRUE)

## More careful bootstrapping -- Consider this when using sample()
## programmatically (i.e., in your function or simulation)!

# sample()'s surprise -- example
x <- 1:10
  sample(x[x > 8]) # length 2
  sample(x[x > 9]) # oops -- length 10!
try(sample(x[x > 10]))# error!

## This is safer:
resample <- function(x, size, ...)
  if(length(x) <= 1) { if(!missing(size) && size == 0) x[FALSE] else x
  } else sample(x, size, ...)

resample(x[x > 8])# length 2
resample(x[x > 9])# length 1
resample(x[x > 10])# length 0

```

---

save

*Save R Objects*

---

**Description**

save writes an external representation of R objects to the specified file. The objects can be read back from the file at a later date by using the function load (or data in some cases).

save.image() is just a short-cut for “save my current environment”, i.e., save(list = ls(all=TRUE), file = ".RData"). It is what also happens with `q("yes")`.

**Usage**

```

save(..., list = character(0),
      file = stop("'file' must be specified"),
      ascii = FALSE, version = NULL, envir = parent.frame(),
      compress = FALSE)

save.image(file = ".RData", version = NULL, ascii = FALSE,
           compress = FALSE, safe = TRUE)

```

```
sys.load.image(name, quiet)
sys.save.image(name)
```

### Arguments

<code>...</code>	the names of the objects to be saved.
<code>list</code>	A character vector containing the names of objects to be saved.
<code>file</code>	a connection or the name of the file where the data will be saved. Must be a file name for workspace format version 1.
<code>ascii</code>	if TRUE, an ASCII representation of the data is written. This is useful for transporting data between machines of different types. The default value of <code>ascii</code> is FALSE which leads to a more compact binary file being written.
<code>version</code>	the workspace format version to use. NULL specifies the current default format. The version used from R 0.99.0 to R 1.3.1 was version 1. The default format as from R 1.4.0 is version 2.
<code>envir</code>	environment to search for objects to be saved.
<code>compress</code>	logical specifying whether saving to a named file is to use compression. Ignored when <code>file</code> is a connection and for workspace format version 1.
<code>safe</code>	logical. If TRUE, a temporary file is used for creating the saved workspace. The temporary file is renamed to <code>file</code> if the save succeeds. This preserves an existing workspace <code>file</code> if the save fails, but at the cost of using extra disk space during the save.
<code>name</code>	name of image file to save or load.
<code>quiet</code>	logical specifying whether a message should be printed.

### Details

All R platforms use the XDR representation of binary objects in binary save-d files, and these are portable across all R platforms.

Default values for the `ascii`, `compress`, `safe` and `version` arguments can be modified with the `save.defaults` option (used both by `save` and `save.image`). If a `save.image.defaults` option is set it overrides `save.defaults` for function `save.image` (which allows to have different defaults). This mechanism is experimental and subject to change.

`sys.save.image` is a system function that is called by `q()` and its GUI analogs; `sys.load.image` is called by the startup code. These functions should not be called directly and are subject to change.

`sys.save.image` closes all connections first, to ensure that it is able to open a connection to save the image. This is appropriate when called from `q()` and allies, but reinforces the warning that it should not be called directly.

### Warning

The `...` arguments only give the *names* of the objects to be saved: they are searched for in the environment given by the `envir` argument, and the actual objects given as arguments need not be those found.

Saved R objects are binary files, even those saved with `ascii=TRUE`, so ensure that they are transferred without conversion of end of line markers. The lines are delimited by LF on all platforms.

### See Also

[dput](#), [dump](#), [load](#), [data](#).

### Examples

```
x <- runif(20)
y <- list(a = 1, b = TRUE, c = "oops")
save(x, y, file = "xy.Rdata")
save.image()
unlink("xy.Rdata")
unlink(".RData")

# set save defaults using option:
options(save.defaults=list(ascii=TRUE, safe=FALSE))
save.image()
unlink(".RData")
```

---

scale

*Scaling and Centering of Matrix-like Objects*

---

### Description

`scale` is generic function whose default method centers and/or scales the columns of a numeric matrix.

### Usage

```
scale(x, center = TRUE, scale = TRUE)
```

### Arguments

<code>x</code>	a numeric matrix(like object).
<code>center</code>	either a logical value or a numeric vector of length equal to the number of columns of <code>x</code> .
<code>scale</code>	either a logical value or a numeric vector of length equal to the number of columns of <code>x</code> .

### Details

The value of `center` determines how column centering is performed. If `center` is a numeric vector with length equal to the number of columns of `x`, then each column of `x` has the corresponding value from `center` subtracted from it. If `center` is `TRUE` then centering is done by subtracting the column means (omitting NAs) of `x` from their corresponding columns, and if `center` is `FALSE`, no centering is done.

The value of `scale` determines how column scaling is performed (after centering). If `scale` is a numeric vector with length equal to the number of columns of `x`, then each column of `x` is divided by the corresponding value from `scale`. If `scale` is `TRUE` then scaling is done by dividing the (centered) columns of `x` by their root-mean-square, and if `scale` is `FALSE`, no scaling is done.

The root-mean-square for a column is obtained by computing the square-root of the sum-of-squares of the non-missing values in the column divided by the number of non-missing values minus one.

### Value

For `scale.default`, the centered, scaled matrix. The numeric centering and scalings used (if any) are returned as attributes `"scaled:center"` and `"scaled:scale"`

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[sweep](#) which allows centering (and scaling) with arbitrary statistics.

For working with the scale of a plot, see [par](#).

### Examples

```
require(stats)
x <- matrix(1:10, nc=2)
(centered.x <- scale(x, scale=FALSE))
cov(centered.scaled.x <- scale(x))# all 1
```

---

scan

*Read Data Values*

---

### Description

Read data into a vector or list from the console or file.

### Usage

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
      quote = if (sep=="\n") "" else "'\"'", dec = ".",
      skip = 0, nlines = 0, na.strings = "NA",
      flush = FALSE, fill = FALSE, strip.white = FALSE,
      quiet = FALSE, blank.lines.skip = TRUE, multiline = TRUE,
      comment.char = "")
```

**Arguments**

<code>file</code>	<p>the name of a file to read data values from. If the specified file is "", then input is taken from the keyboard (or <code>stdin</code> if input is redirected). (In this case input can be terminated by a blank line or an EOF signal, <code>Ctrl-D</code> on Unix and <code>Ctrl-Z</code> on Windows.)</p> <p>Otherwise, the file name is interpreted <i>relative</i> to the current working directory (given by <code>getwd()</code>), unless it specifies an <i>absolute</i> path. Tilde-expansion is performed where supported.</p> <p>Alternatively, <code>file</code> can be a <i>connection</i>, which will be opened if necessary, and if so closed at the end of the function call. Whatever mode the connection is opened in, any of LF, CRLF or CR will be accepted as the EOL marker for a line and so will match <code>sep = "\n"</code>.</p> <p><code>file</code> can also be a complete URL.</p>
<code>what</code>	<p>the type of <code>what</code> gives the type of data to be read. If <code>what</code> is a list, it is assumed that the lines of the data file are records each containing <code>length(what)</code> items ("fields"). The supported types are <code>logical</code>, <code>integer</code>, <code>numeric</code>, <code>complex</code>, <code>character</code>, <code>raw</code> and <code>list</code>: <code>list</code> values should have elements which are one of the first six types listed or <code>NULL</code>.</p>
<code>nmax</code>	<p>the maximum number of data values to be read, or if <code>what</code> is a list, the maximum number of records to be read. If omitted or not positive (and <code>nlines</code> is not set to a positive value), <code>scan</code> will read to the end of <code>file</code>.</p>
<code>n</code>	<p>the maximum number of data values to be read, defaulting to no limit.</p>
<code>sep</code>	<p>by default, <code>scan</code> expects to read white-space delimited input fields. Alternatively, <code>sep</code> can be used to specify a character which delimits fields. A field is always delimited by an end-of-line marker unless it is quoted.</p>
<code>quote</code>	<p>the set of quoting characters as a single character string.</p>
<code>dec</code>	<p>decimal point character.</p>
<code>skip</code>	<p>the number of lines of the input file to skip before beginning to read data values.</p>
<code>nlines</code>	<p>if positive, the maximum number of lines of data to be read.</p>
<code>na.strings</code>	<p>character vector. Elements of this vector are to be interpreted as missing (<code>NA</code>) values.</p>
<code>flush</code>	<p>logical: if <code>TRUE</code>, <code>scan</code> will flush to the end of the line after reading the last of the fields requested. This allows putting comments after the last field, but precludes putting more than one record on a line.</p>
<code>fill</code>	<p>logical: if <code>TRUE</code>, <code>scan</code> will implicitly add empty fields to any lines with fewer fields than implied by <code>what</code>.</p>
<code>strip.white</code>	<p>vector of logical value(s) corresponding to items in the <code>what</code> argument. It is used only when <code>sep</code> has been specified, and allows the stripping of leading and trailing white space from <code>character</code> fields (<code>numeric</code> fields are always stripped).</p> <p>If <code>strip.white</code> is of length 1, it applies to all fields; otherwise, if <code>strip.white[i]</code> is <code>TRUE</code> and the <i>i</i>-th field is of mode <code>character</code> (because <code>what[i]</code> is) then the leading and trailing white space from field <i>i</i> is stripped.</p>

<code>quiet</code>	logical: if <code>FALSE</code> (default), <code>scan()</code> will print a line, saying how many items have been read.
<code>blank.lines.skip</code>	logical: if <code>TRUE</code> blank lines in the input are ignored, except when counting <code>skip</code> and <code>nlines</code> .
<code>multi.line</code>	logical. Only used if <code>what</code> is a list. If <code>FALSE</code> , all of a record must appear on one line (but more than one record can appear on a single line). Note that using <code>fill = TRUE</code> implies that a record will terminated at the end of a line.
<code>comment.char</code>	character: a character vector of length one containing a single character or an empty string. Use <code>" "</code> to turn off the interpretation of comments altogether (the default).

### Details

The value of `what` can be a list of types, in which case `scan` returns a list of vectors with the types given by the types of the elements in `what`. This provides a way of reading columnar data. If any of the types is `NULL`, the corresponding field is skipped (but a `NULL` component appears in the result).

The type of `what` or its components can be one of the five atomic types or `NULL`,

Empty numeric fields are always regarded as missing values. Empty character fields are scanned as empty character vectors, unless `na.strings` contains `" "` when they are regarded as missing values.

If `sep` is the default (`" "`), the character `\` in a quoted string escapes the following character, so quotes may be included in the string by escaping them.

If `sep` is non-default, the fields may be quoted in the style of `' .csv '` files where separators inside quotes (`"` or `" "`) are ignored and quotes may be put inside strings by doubling them. However, if `sep = "\n"` it is assumed by default that one wants to read entire lines verbatim.

Quoting is only interpreted in character fields, and as from R 1.8.0 in `NULL` fields (which might be skipping character fields).

Note that since `sep` is a separator and not a terminator, reading a file by `scan("foo", sep="\n", blank.lines.skip=FALSE)` will give an empty file line if the file ends in a linefeed and not if it does not. This might not be what you expected; see also [readLines](#).

If `comment.char` occurs (except inside a quoted character field), it signals that the rest of the line should be regarded as a comment and be discarded. Lines beginning with a comment character (possibly after white space) are treated as blank lines.

### Value

if `what` is a list, a list of the same length and same names (as any) as `what`.

Otherwise, a vector of the type of `what`.

### Note

The default for `multi.line` differs from `S`. To read one record per line, use `flush = TRUE` and `multi.line = FALSE`.

If number of items is not specified, the internal mechanism re-allocates memory in powers of two and so could use up to three times as much memory as needed. (It needs both old and new copies.) If you can, specify either `n` or `nmax` whenever inputting a large vector, and `nmax` or `nlines` when inputting a large list.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[read.table](#) for more user-friendly reading of data matrices; [readLines](#) to read a file a line at a time. [write](#).

## Examples

```
cat("TITLE extra line", "2 3 5 7", "11 13 17", file="ex.data", sep="\n")
pp <- scan("ex.data", skip = 1, quiet= TRUE)
  scan("ex.data", skip = 1)
  scan("ex.data", skip = 1, nlines=1)# only 1 line after the skipped one
scan("ex.data", what = list("", "", "")) # flush is F -> read "7"
scan("ex.data", what = list("", "", ""), flush = TRUE)
unlink("ex.data") # tidy up
```

---

search

*Give Search Path for R Objects*

---

## Description

Gives a list of [attached packages](#) (see [library](#)), and R objects, usually [data.frames](#).

## Usage

```
search()
searchpaths()
```

## Value

A character vector, starting with `".GlobalEnv"`, and ending with `"package:base"` which is R's **base** package required always.

`searchpaths` gives a similar character vector, with the entries for packages being the path to the package used to load the code.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. ([search](#).)

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer. ([searchPaths](#).)

## See Also

[.packages](#) to list just the packages on search path.

[loadedNamespaces](#) to list loaded namespaces.

[attach](#) and [detach](#) to change the search “path”, [objects](#) to find R objects in there.

## Examples

```
search()  
searchpaths()
```

---

seek

*Functions to Reposition Connections*

---

## Description

Functions to re-position connections.

## Usage

```
seek(con, ...)  
## S3 method for class 'connection':  
seek(con, where = NA, origin = "start", rw = "", ...)  
  
isSeekable(con)  
  
truncate(con, ...)
```

## Arguments

<code>con</code>	a connection.
<code>where</code>	integer. A file position (relative to the origin specified by <code>origin</code> ), or NA.
<code>rw</code>	character. Empty or "read" or "write", partial matches allowed.
<code>origin</code>	character. One of "start", "current", "end".
<code>...</code>	further arguments passed to or from other methods.

**Details**

`seek` with `where = NA` returns the current byte offset of a connection (from the beginning), and with a non-missing `where` argument the connection is re-positioned (if possible) to the specified position. `isSeekable` returns whether the connection in principle supports `seek`: currently only (possibly gz-compressed) file connections do. `gzfile` connections do not support `origin = "end"`, and the file position they use is that of the uncompressed file.

File connections can be open for both writing/append, in which case R keeps separate positions for reading and writing. Which `seek` refers to can be set by its `rw` argument: the default is the last mode (reading or writing) which was used. Most files are only opened for reading or writing and so default to that state. If a file is open for reading and writing but has not been used, the default is to give the reading position (0).

The initial file position for reading is always at the beginning. The initial position for writing is at the beginning of the file for modes `"r+"` and `"r+b"`, otherwise at the end of the file. Some platforms only allow writing at the end of the file in the append modes.

`truncate` truncates a file opened for writing at its current position. It works only for `file` connections, and is not implemented on all platforms.

**Value**

`seek` returns the current position (before any move), as a byte offset, if relevant, or 0 if not.

`truncate` returns `NULL`: it stops with an error if it fails (or is not implemented).

`isSeekable` returns a logical value, whether the connection is support `seek`.

**See Also**

[connections](#)

---

seq

*Sequence Generation*

---

**Description**

Generate regular sequences.

**Usage**

```
from:to
  a:b
```

```
seq(from, to)
seq(from, to, by=)
seq(from, to, length=)
seq(along=)
seq(from)
```

**Arguments**

from	starting value of sequence.
to	(maximal) end value of the sequence.
by	increment of the sequence.
length	desired length of the sequence.
along	take the length from the length of this argument.
a,b	<a href="#">factors</a> of same length.

**Details**

The binary operator `:` has two meanings: for factors `a:b` is equivalent to [interaction](#)(a, b) (except for labelling by `1a:1b` not `1a.1b`). For numeric arguments `a:b` is equivalent to `seq(from=a, to=b)`.

The interpretation of the unnamed arguments of `seq` is *not* standard, and it is recommended always to name the arguments when programming.

Function `seq` is generic, and only the default method is described here.

The operator `:` and the `seq(from, to)` form generate the sequence `from, from+1, ..., to`.

The second form generates `from, from+by, ...,` up to the sequence value less than or equal to `to`.

The third generates a sequence of `length` equally spaced values from `from` to `to`.

The fourth form generates the sequence `1, 2, ..., length(along)`.

The last generates the sequence `1, 2, ..., length(from)` (as if argument `along` had been specified), *unless* the argument is numeric of length 1 when it is interpreted as `1:from` (even for `seq(0)` for compatibility with S).

If `from` and `to` are factors of the same length, then `from : to` returns the “cross” of the two.

Very small sequences (with `from - to` of the order of  $10^{-14}$  times the larger of the ends) will return `from`.

**Value**

The result is of mode `"integer"` if `from` is (numerically equal to an) integer and `by` is not specified.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

The method [seq.POSIXt](#).

[rep](#), [sequence](#), [row](#), [col](#).

As an alternative to using `:` for factors, [interaction](#).

**Examples**

```

1:4
pi:6 # float
6:pi # integer

seq(0,1, length=11)
seq(rnorm(20))
seq(1,9, by = 2) # match
seq(1,9, by = pi)# stay below
seq(1,6, by = 3)
seq(1.575, 5.125, by=0.05)
seq(17) # same as 1:17

for (x in list(NULL, letters[1:6], list(1,pi)))
  cat("x=",deparse(x),"; seq(along = x):",seq(along = x),"\n")

f1 <- gl(2,3); f1
f2 <- gl(3,2); f2
f1:f2 # a factor, the "cross" f1 x f2

```

seq.Date

*Generate Regular Sequences of Dates***Description**

The method for `seq` for date-time classes.

**Usage**

```

## S3 method for class 'Date':
seq(from, to, by, length.out = NULL, along.with = NULL, ...)

```

**Arguments**

<code>from</code>	starting date. Required
<code>to</code>	end date. Optional. If supplied must be after <code>from</code> .
<code>by</code>	increment of the sequence. Optional. See Details.
<code>length.out</code>	integer, optional. desired length of the sequence.
<code>along.with</code>	take the length from the length of this argument.
<code>...</code>	arguments passed to or from other methods.

**Details**

`by` can be specified in several ways.

- A number, taken to be in days.
- A object of class `difftime`
- A character string, containing one of "day", "week", "month" or "year". This can optionally be preceded by an integer and a space, or followed by "s".

**Value**

A vector of class "Date".

**See Also**

[Date](#)

**Examples**

```
## first days of years
seq(as.Date("1910/1/1"), as.Date("1999/1/1"), "years")
## by month
seq(as.Date("2000/1/1"), by="month", length=12)
## quarters
seq(as.Date("2000/1/1"), as.Date("2003/1/1"), by="3 months")
```

---

seq.POSIXt

*Generate Regular Sequences of Dates*


---

**Description**

The method for [seq](#) for date-time classes.

**Usage**

```
## S3 method for class 'POSIXt':
seq(from, to, by, length.out = NULL, along.with = NULL, ...)
```

**Arguments**

from	starting date. Required
to	end date. Optional. If supplied must be after from.
by	increment of the sequence. Optional. See Details.
length.out	integer, optional. desired length of the sequence.
along.with	take the length from the length of this argument.
...	arguments passed to or from other methods.

**Details**

by can be specified in several ways.

- A number, taken to be in seconds.
- A object of class [difftime](#)
- A character string, containing one of "sec", "min", "hour", "day", "DSTday", "week", "month" or "year". This can optionally be preceded by an integer and a space, or followed by "s".

The difference between "day" and "DSTday" is that the former ignores changes to/from daylight savings time and the latter takes the same clock time each day. ("week" ignores DST, but "7 DSTdays") can be used as an alternative. "month" and "year" allow for DST as from R 1.5.0.)

### Value

A vector of class "POSIXct".

### See Also

[DateTimeClasses](#)

### Examples

```
## first days of years
seq(ISOdate(1910,1,1), ISOdate(1999,1,1), "years")
## by month
seq(ISOdate(2000,1,1), by = "month", length = 12)
## quarters
seq(ISOdate(1990,1,1), ISOdate(2000,1,1), by = "3 months")
## days vs DSTdays
seq(ISOdate(2000,3,20), by = "day", length = 10)
seq(ISOdate(2000,3,20), by = "DSTday", length = 10)
seq(ISOdate(2000,3,20), by = "7 DSTdays", length = 4)
```

---

sequence

*Create A Vector of Sequences*

---

### Description

For each element of `nvec` the sequence `seq(nvec[i])` is created. These are appended and the result returned.

### Usage

```
sequence(nvec)
```

### Arguments

`nvec` an integer vector each element of which specifies the upper bound of a sequence.

### See Also

[gl](#), [seq](#), [rep](#).

### Examples

```
sequence(c(3,2))# the concatenated sequences 1:3 and 1:2.
#> [1] 1 2 3 1 2
```

**Description**

Performs **set** union, intersection, (asymmetric!) difference, equality and membership on two vectors.

**Usage**

```
union(x, y)
intersect(x, y)
setdiff(x, y)
setequal(x, y)
is.element(el, set)
```

**Arguments**

`x`, `y`, `el`, `set`  
vectors (of the same mode) containing a sequence of items (conceptually) with no duplicated values.

**Details**

Each of `union`, `intersect` and `setdiff` will remove any duplicated values in the arguments. `is.element(x, y)` is identical to `x %in% y`.

**Value**

A vector of the same `mode` as `x` or `y` for `setdiff` and `intersect`, respectively, and of a common mode for `union`.

A logical scalar for `setequal` and a logical of the same length as `x` for `is.element`.

**See Also**

`%in%`

**Examples**

```
(x <- c(sort(sample(1:20, 9)),NA))
(y <- c(sort(sample(3:23, 7)),NA))
union(x, y)
intersect(x, y)
setdiff(x, y)
setdiff(y, x)
setequal(x, y)

## True for all possible x & y :
```

```

setequal( union(x,y),
          c(setdiff(x,y), intersect(x,y), setdiff(y,x)))

is.element(x, y)# length 10
is.element(y, x)# length 8

```

---

showConnections      *Display Connections*

---

## Description

Display aspects of connections.

## Usage

```

showConnections(all = FALSE)
getConnection(what)
closeAllConnections()

stdin()
stdout()
stderr()

```

## Arguments

all	logical: if true all connections, including closed ones and the standard ones are displayed. If false only open user-created connections are included.
what	integer: a row number of the table given by showConnections.

## Details

`stdin()`, `stdout()` and `stderr()` are standard connections corresponding to input, output and error on the console respectively (and not necessarily to file streams). They are text-mode connections of class "terminal" which cannot be opened or closed, and are read-only, write-only and write-only respectively. The `stdout()` and `stderr()` connections can be re-directed by [sink](#).

`showConnections` returns a matrix of information. If a connection object has been lost or forgotten, `getConnection` will take a row number from the table and return a connection object for that connection, which can be used to close the connection, for example.

`closeAllConnections` closes (and destroys) all open user connections, restoring all [sink](#) diversions as it does so.

## Value

`stdin()`, `stdout()` and `stderr()` return connection objects.

`showConnections` returns a character matrix of information with a row for each connection, by default only for open non-standard connections.

`getConnection` returns a connection object, or NULL.

**See Also**[connections](#)**Examples**

```

showConnections(all = TRUE)

textConnection(letters)
# oops, I forgot to record that one
showConnections()
# class      description      mode text  isopen  can read can write
#3 "letters" "textConnection" "r"  "text" "opened" "yes"    "no"
## Not run: close(getConnection(3))

showConnections()

```

---

shQuote*Quote Strings for Use in OS Shells*

---

**Description**

Quote a string to be passed to an operating system shell.

**Usage**

```
shQuote(string, type = c("sh", "csh", "cmd"))
```

**Arguments**

<code>string</code>	a character vector, usually of length one.
<code>type</code>	character: the type of shell. Partial matching is supported. "cmd" refers to the Windows NT shell.

**Details**

The default type of quoting supported is that for the Bourne shell `sh`. If the string does not contain single quotes, we can just surround it with single quotes. Otherwise, the string is surrounded in double quotes, which suppresses all special meanings of metacharacters except dollar, backquote and backslash, so these (and of course double quote) are preceded by backslash. This type of quoting is also appropriate for `ksh` and `bash`.

The other type of quoting is for the C-shell (`csh` and `tcsh`). Once again, if the string does not contain single quotes, we can just surround it with single quotes. If it does contain single quotes, we can use double quotes provided it does not contain dollar or backquote (and we need to escape backslash, exclamation mark and double quote). As a last resort, we need to split the string into pieces not containing single quotes and surround each with single quotes, and the single quotes with double quotes.

The Windows shell supports only double quoting. All this implementation does is to surround by double quotes and escape internal double quotes by a backslash.

## References

Loukides, M. et al (2002) *Unix Power Tools* Third Edition. O'Reilly. Section 27.12.  
[http://www.mhuffman.com/notes/dos/bash\\_cmd.htm](http://www.mhuffman.com/notes/dos/bash_cmd.htm)

## See Also

[sQuote](#) for quoting in text.

## Examples

```
test <- "abc$def`gh`i`j"
cat(shQuote(test), "\n")
## Not run: system(paste("echo", shQuote(test)))
test <- "don't do it!"
cat(shQuote(test), "\n")

tryit <- "use the `-c' switch\nlike this"
cat(shQuote(tryit), "\n")
## Not run: system(paste("echo", shQuote(tryit)))
cat(shQuote(tryit, type="csh"), "\n")

## Windows-only example.
perlcmd <- 'print "Hello World\n";'
## Not run: shell(paste("perl -e", shQuote(perlcmd, type="cmd")))
```

---

sign

*Sign Function*

---

## Description

`sign` returns a vector with the signs of the corresponding elements of `x` (the sign of a real number is 1, 0, or  $-1$  if the number is positive, zero, or negative, respectively).

Note that `sign` does not operate on complex vectors.

## Usage

```
sign(x)
```

## Arguments

`x` a numeric vector

## Details

This is a generic function: methods can be defined for it directly or via the [Math](#) group generic.

## See Also

[abs](#)

**Examples**

```
sign(pi) # == 1
sign(-2:3)# -1 -1 0 1 1 1
```

---

Signals

*Interrupting Execution of R*

---

**Description**

On receiving SIGUSR1 R will save the workspace and quit. SIGUSR2 has the same result except that the `.Last` function and `on.exit` expressions will not be called.

**Usage**

```
kill -USR1 pid
kill -USR2 pid
```

**Arguments**

`pid`            The process ID of the R process

**Warning**

It is possible that one or more R objects will be undergoing modification at the time the signal is sent. These objects could be saved in a corrupted form.

---

sink

*Send R Output to a File*

---

**Description**

`sink` diverts R output to a connection.

`sink.number()` reports how many diversions are in use.

`sink.number(type = "message")` reports the number of the connection currently being used for error messages.

**Usage**

```
sink(file = NULL, append = FALSE, type = c("output", "message"),
      split = FALSE)
```

```
sink.number(type = c("output", "message"))
```

**Arguments**

<code>file</code>	a connection or a character string naming the file to write to, or <code>NULL</code> to stop sink-ing.
<code>append</code>	logical. If <code>TRUE</code> , output will be appended to <code>file</code> ; otherwise, it will overwrite the contents of <code>file</code> .
<code>type</code>	character. Either the output stream or the messages stream.
<code>split</code>	logical: if <code>TRUE</code> , output will be sent to the new sink and to the current output stream.

**Details**

`sink` diverts R output to a connection. If `file` is a character string, a file connection with that name will be established for the duration of the diversion.

Normal R output is diverted by the default `type = "output"`. Only prompts and warning/error messages continue to appear on the terminal. The latter can be diverted by `type = "message"` (see below).

`sink()` or `sink(file=NULL)` ends the last diversion (of the specified type). As from R version 1.3.0 there is a stack of diversions for normal output, so output reverts to the previous diversion (if there was one). The stack is of up to 21 connections (20 diversions).

If `file` is a connection it will be opened if necessary.

Sink-ing the messages stream should be done only with great care. For that stream `file` must be an already open connection, and there is no stack of connections.

**Value**

`sink` returns `NULL`.

For `sink.number()` the number (0, 1, 2, ...) of diversions of output in place.

For `sink.number("message")` the connection number used for messages, 2 if no diversion has been used.

**Warning**

Don't use a connection that is open for `sink` for any other purpose. The software will stop you closing one such inadvertently.

Do not sink the messages stream unless you understand the source code implementing it and hence the pitfalls.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M. (1998) *Programming with Data. A Guide to the S Language*. Springer.

**See Also**

[capture.output](#)

## Examples

```
sink("sink-examp.txt")
i <- 1:10
outer(i, i, "*")
sink()
unlink("sink-examp.txt")
## Not run:
## capture all the output to a file.
zz <- file("all.Rout", open="wt")
sink(zz)
sink(zz, type="message")
try(log("a"))
## back to the console
sink(type="message")
sink()
try(log("a"))
## End(Not run)
```

---

slice.index

*Slice Indexes in an Array*

---

## Description

Returns a matrix of integers indicating the number of their slice in a given array.

## Usage

```
slice.index(x, MARGIN)
```

## Arguments

x	an array. If x has no dimension attribute, it is considered a one-dimensional array.
MARGIN	an integer giving the dimension number to slice by.

## Value

An integer array y with dimensions corresponding to those of x such that all elements of slice number i with respect to dimension MARGIN have value i.

## See Also

`row` and `col` for determining row and column indexes; in fact, these are special cases of `slice.index` corresponding to MARGIN equal to 1 and 2, respectively when x is a matrix.

## Examples

```
x <- array(1 : 24, c(2, 3, 4))
slice.index(x, 2)
```

---

slotOp

*Extract Slots*


---

**Description**

Extract the contents of a slot in a object with a formal class structure.

**Usage**

```
object@name
```

**Arguments**

object	An object from a formally defined class.
name	The character-string name of the slot.

**Details**

These operators support the formal classes of package **methods**. See [slot](#) for further details. Currently there is no checking that the object is an instance of a class.

**See Also**

[Extract](#), [slot](#)

---

socketSelect

*Wait on Socket Connections*


---

**Description**

Waits for the first of several socket connections to become available.

**Usage**

```
socketSelect(socklist, write = FALSE, timeout = NULL)
```

**Arguments**

socklist	list of open socket connections
write	logical. If TRUE wait for corresponding socket to become available for writing; otherwise wait for it to become available for reading.
timeout	numeric or NULL. Time in seconds to wait for a socket to become available; NULL means wait indefinitely.

**Details**

The values in `write` are recycled if necessary to make up a logical vector the same length as `socklist`. Socket connections can appear more than once in `socklist`; this can be useful if you want to determine whether a socket is available for reading or writing.

**Value**

Logical the same length as `socklist` indicating whether the corresponding socket connection is available for output or input, depending on the corresponding value of `write`.

**Examples**

```
## Not run:
## test whether socket connection s is available for writing or reading
socketSelect(list(s,s),c(TRUE,FALSE),timeout=0)
## End(Not run)
```

---

solve

*Solve a System of Equations*

---

**Description**

This generic function solves the equation  $a \%*\% x = b$  for  $x$ , where  $b$  can be either a vector or a matrix.

**Usage**

```
solve(a, b, ...)
```

```
## Default S3 method:
solve(a, b, tol, LINPACK = FALSE, ...)
```

**Arguments**

<code>a</code>	a square numeric or complex matrix containing the coefficients of the linear system.
<code>b</code>	a numeric or complex vector or matrix giving the right-hand side(s) of the linear system. If missing, <code>b</code> is taken to be an identity matrix and <code>solve</code> will return the inverse of <code>a</code> .
<code>tol</code>	the tolerance for detecting linear dependencies in the columns of <code>a</code> . If <code>LINPACK</code> is <code>TRUE</code> the default is $1e-7$ , otherwise it is <code>.Machine\$double.eps</code> . Future versions of R may use a tighter tolerance. Not presently used with complex matrices <code>a</code> .
<code>LINPACK</code>	logical. Should <code>LINPACK</code> be used (for compatibility with R < 1.7.0)? Otherwise <code>LAPACK</code> is used.
<code>...</code>	further arguments passed to or from other methods

**Details**

`a` or `b` can be complex, but this uses double complex arithmetic which might not be available on all platforms and LAPACK will always be used.

The row and column names of the result are taken from the column names of `a` and of `b` respectively. As from R 1.7.0 if `b` is missing the column names of the result are the row names of `a`. No check is made that the column names of `a` and the row names of `b` are equal.

For back-compatibility `a` can be a (real) QR decomposition, although `qr.solve` should be called in that case. `qr.solve` can handle non-square systems.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`solve.qr` for the `qr` method, `chol2inv` for inverting from the Choleski factor `backsolve`, `qr.solve`.

**Examples**

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
h8 <- hilbert(8); h8
sh8 <- solve(h8)
round(sh8 %*% h8, 3)

A <- hilbert(4)
A[] <- as.complex(A)
## might not be supported on all platforms
try(solve(A))
```

---

 sort

*Sorting or Ordering Vectors*


---

**Description**

`Sort` (or *order*) a numeric or complex vector (partially) into ascending (or descending) order.

**Usage**

```
sort(x, partial = NULL, na.last = NA, decreasing = FALSE,
     method = c("shell", "quick"), index.return = FALSE)

is.unsorted(x, na.rm = FALSE)
```

**Arguments**

<code>x</code>	a numeric or complex vector.
<code>partial</code>	a vector of indices for partial sorting.
<code>na.last</code>	for controlling the treatment of NAs. If <code>TRUE</code> , missing values in the data are put last; if <code>FALSE</code> , they are put first; if <code>NA</code> , they are removed.
<code>decreasing</code>	logical. Should the sort be increasing or decreasing?
<code>method</code>	character specifying the algorithm used.
<code>index.return</code>	logical indicating if the ordering index vector should be returned as well; this is only available for the default <code>na.last = NA</code> .
<code>na.rm</code>	logical. Should missing values be removed?

**Details**

If `partial` is not `NULL`, it is taken to contain indices of elements of `x` which are to be placed in their correct positions by partial sorting. After the sort, the values specified in `partial` are in their correct position in the sorted array. Any values smaller than these values are guaranteed to have a smaller index in the sorted array and any values which are greater are guaranteed to have a bigger index in the sorted array.

The sort order for character vectors will depend on the collating sequence of the locale in use: see [Comparison](#).

`is.unsorted` returns a logical indicating if `x` is sorted increasingly, i.e., `is.unsorted(x)` is true if `any(x != sort(x))` (and there are no NAs).

`method = "shell"` uses Shellsort (an  $O(n^{4/3})$  variant from Sedgewick (1996)). If `x` has names a stable sort is used, so ties are not reordered. (This only matters if names are present.)

Method `"quick"` uses Singleton's Quicksort implementation and is only available when `x` is numeric (double or integer) and `partial` is `NULL`. It is normally somewhat faster than Shellsort (perhaps twice as fast on vectors of length a million) but has poor performance in the rare worst case. (Peto's modification using a pseudo-random midpoint is used to make the worst case rarer.) This is not a stable sort, and ties may be reordered.

**Value**

For `sort` the sorted vector unless `index.return` is true, when the result is a list with components named `x` and `ix` containing the sorted numbers and the ordering index vector. In the latter case, if `method == "quick"` ties may be reversed in the ordering, unlike `sort.list`, as quicksort is not stable.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Sedgewick, R. (1986) A new upper bound for Shell sort. *J. Algorithms* **7**, 159–173.

Singleton, R. C. (1969) An efficient algorithm for sorting with minimal storage: Algorithm 347. *Communications of the ACM* **12**, 185–187.

**See Also**

[order](#), [rank](#).

**Examples**

```
require(stats)
x <- swiss$Education[1:25]
x; sort(x); sort(x, partial = c(10, 15))
median # shows you another example for 'partial'

## illustrate 'stable' sorting (of ties):
sort(c(10:3,2:12), method = "sh", index=TRUE) # is stable
## $x : 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 12
## $ix: 9 8 10 7 11 6 12 5 13 4 14 3 15 2 16 1 17 18 19
sort(c(10:3,2:12), method = "qu", index=TRUE) # is not
## $x : 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 12
## $ix: 9 10 8 7 11 6 12 5 13 4 14 3 15 16 2 17 1 18 19
##          ^^^^^^

## Not run: ## Small speed comparison simulation:
N <- 2000
Sim <- 20
rep <- 50 # << adjust to your CPU
c1 <- c2 <- numeric(Sim)
for(is in 1:Sim){
  x <- rnorm(N)
  gc() ## sort should not have to pay for gc
  c1[is] <- system.time(for(i in 1:rep) sort(x, method = "shell"))[1]
  c2[is] <- system.time(for(i in 1:rep) sort(x, method = "quick"))[1]
  stopifnot(sort(x, meth = "s") == sort(x, meth = "q"))
}
100 * rbind(ShellSort = c1, QuickSort = c2)
cat("Speedup factor of quick sort():\n")
summary({qq <- c1 / c2; qq[is.finite(qq)]})

## A larger test
x <- rnorm(1e6)
gc()
system.time(x1 <- sort(x, method = "shell"))
gc()
system.time(x2 <- sort(x, method = "quick"))
stopifnot(identical(x1, x2))
## End(Not run)
```

## Description

`source` causes R to accept its input from the named file (the name must be quoted). Input is read from that file until the end of the file is reached. `parse` is used to scan the expressions in, they are then evaluated sequentially in the chosen environment.

## Usage

```
source(file, local = FALSE, echo = verbose, print.eval = echo,
       verbose = getOption("verbose"),
       prompt.echo = getOption("prompt"),
       max.deparse.length = 150, chdir = FALSE)
```

## Arguments

<code>file</code>	a connection or a character string giving the name of the file or URL to read from.
<code>local</code>	if <code>local</code> is <code>FALSE</code> , the statements scanned are evaluated in the user's workspace (the global environment), otherwise in the environment calling <code>source</code> .
<code>echo</code>	logical; if <code>TRUE</code> , each expression is printed after parsing, before evaluation.
<code>print.eval</code>	logical; if <code>TRUE</code> , the result of <code>eval(i)</code> is printed for each expression <code>i</code> ; defaults to <code>echo</code> .
<code>verbose</code>	if <code>TRUE</code> , more diagnostics (than just <code>echo = TRUE</code> ) are printed during parsing and evaluation of input, including extra info for <b>each</b> expression.
<code>prompt.echo</code>	character; gives the prompt to be used if <code>echo = TRUE</code> .
<code>max.deparse.length</code>	integer; is used only if <code>echo</code> is <code>TRUE</code> and gives the maximal length of the "echo" of a single expression.
<code>chdir</code>	logical; if <code>TRUE</code> , the R working directory is changed to the directory containing <code>file</code> for evaluating.

## Details

All versions of R accept input from a connection with end of line marked by LF (as used on Unix), CRLF (as used on DOS/Windows) or CR (as used on classic MacOS). The final line can be incomplete, that is missing the final EOL marker.

If `options("keep.source")` is true (the default), the source of functions is kept so they can be listed exactly as input. This imposes a limit of 128K chars on the function size and a nesting limit of 265. Use `option(keep.source = FALSE)` when these limits might take effect: if exceeded they generate an error.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[demo](#) which uses `source`; `eval`, `parse` and `scan`; `options("keep.source")`.

Special

*Special Functions of Mathematics***Description**

Special mathematical functions related to the beta and gamma functions.

**Usage**

```
beta(a, b)
lbeta(a, b)
gamma(x)
lgamma(x)
psigamma(x, deriv = 0)
digamma(x)
trigamma(x)
choose(n, k)
lchoose(n, k)
factorial(x)
lfactorial(x)
```

**Arguments**

`a`, `b`, `x`        numeric vectors.  
`n`, `k`, `deriv`    integer vectors.

**Details**

The functions `beta` and `lbeta` return the beta function and the natural logarithm of the beta function,

$$B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

The formal definition is

$$B(a, b) = \int_0^1 t^{a-1}(1-t)^{b-1} dt$$

(Abramowitz and Stegun (6.2.1), page 258).

The functions `gamma` and `lgamma` return the gamma function  $\Gamma(x)$  and the natural logarithm of the absolute value of the gamma function. The gamma function is defined by (Abramowitz and Stegun (6.1.1), page 255)

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

`factorial(x)` is  $x!$  and identical to `gamma(x+1)` and `lfactorial` is `lgamma(x+1)`.

The functions `digamma` and `trigamma` return the first and second derivatives of the logarithm of the gamma function. `psigamma(x, deriv)` (`deriv >= 0`) is more generally computing the `deriv`-th derivative of  $\psi(x)$ .

$$\text{digamma}(x) = \psi(x) = \frac{d}{dx} \ln \Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

The functions `choose` and `lchoose` return binomial coefficients and their logarithms.

All the `*gamma*` functions are generic: methods can be defined for them individually or via the [Math](#) group generic.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (for `gamma` and `lgamma`.)

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. Chapter 6: Gamma and Related Functions.

## See Also

[Arithmetic](#) for simple, [sqrt](#) for miscellaneous mathematical functions and [Bessel](#) for the real Bessel functions.

For the incomplete gamma function see [pgamma](#).

## Examples

```
choose(5, 2)
for (n in 0:10) print(choose(n, k = 0:n))

factorial(100)
lfactorial(10000)

## gamma has 1st order poles at 0, -1, -2, ...
x <- sort(c(seq(-3,4, length=201), outer(0:-3, (-1:1)*1e-6, "+")))
plot(x, gamma(x), ylim=c(-20,20), col="red", type="l", lwd=2,
      main=expression(Gamma(x)))
abline(h=0, v=-3:0, lty=3, col="midnightblue")

x <- seq(.1, 4, length = 201); dx <- diff(x)[1]
par(mfrow = c(2, 3))
for (ch in c("", "l", "di", "tri", "tetra", "penta")) {
  is.deriv <- nchar(ch) >= 2
  nm <- paste(ch, "gamma", sep = "")
  if (is.deriv) {
    dy <- diff(y) / dx # finite difference
    der <- which(ch == c("di", "tri", "tetra", "penta")) - 1
    nm2 <- paste("psigamma(*, deriv = ", der, ")", sep='')
    nm <- if(der >= 2) nm2 else paste(nm, nm2, sep = " ==\n")
    y <- psigamma(x, deriv=der)
  } else {
    y <- get(nm)(x)
  }
}
```

```

}
plot(x, y, type = "l", main = nm, col = "red")
abline(h = 0, col = "lightgray")
if (is.deriv) lines(x[-1], dy, col = "blue", lty = 2)
}

```

---

split

*Divide into Groups*


---

### Description

`split` divides the data in the vector `x` into the groups defined by `f`. The assignment forms replace values corresponding to such a division. `unsplit` reverses the effect of `split`.

### Usage

```

split(x, f)
split(x, f) <- value
unsplit(value, f)

```

### Arguments

<code>x</code>	vector or data frame containing values to be divided into groups.
<code>f</code>	a “factor” such that <code>factor(f)</code> defines the grouping, or a list of such factors in which case their interaction is used for the grouping.
<code>value</code>	a list of vectors or data frames compatible with a splitting of <code>x</code>

### Details

`split` and `split<-` are generic functions with default and `data.frame` methods.

`f` is recycled as necessary and if the length of `x` is not a multiple of the length of `f` a warning is printed. `unsplit` works only with lists of vectors. The data frame method can also be used to split a matrix into a list of matrices, and the assignment form likewise, provided they are invoked explicitly.

Any missing values in `f` are dropped together with the corresponding values of `x`.

### Value

The value returned from `split` is a list of vectors containing the values for the groups. The components of the list are named by the *used* factor levels given by `f`. (If `f` is longer than `x` then some of the components will be of zero length.)

The assignment forms return their right hand side. `unsplit` returns a vector for which `split(x, f)` equals `value`

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[cut](#)

**Examples**

```
require(stats)
n <- 10; mn <- 100
g <- factor(round(n * runif(n * mn)))
x <- rnorm(n * mn) + sqrt(as.numeric(g))
xg <- split(x, g)
boxplot(xg, col = "lavender", notch = TRUE, varwidth = TRUE)
sapply(xg, length)
sapply(xg, mean)

## Calculate z-scores by group

z <- unsplit(lapply(split(x, g), scale), g)
tapply(z, g, mean)

# or

z <- x
split(z, g) <- lapply(split(x, g), scale)
tapply(z, g, sd)

## Split a matrix into a list by columns
ma <- cbind(x = 1:10, y = (-4:5)^2)
split(ma, col(ma))

split(1:10, 1:2)
```

---

sprintf

*Use C-style String Formatting Commands*

---

**Description**

A wrapper for the C function `sprintf`, that returns a character vector of length one containing a formatted combination of text and variable values.

**Usage**

```
sprintf(fmt, ...)
```

**Arguments**

`fmt` a format string.  
`...` values to be passed into `fmt`. Only logical, integer, real and character vectors are accepted, and only the first value is read from each vector.

**Details**

This is a wrapper for the system `sprintf` C-library function. Attempts are made to check that the mode of the values passed match the format supplied, and R's special values (`NA`, `Inf`, `-Inf` and `NaN`) are handled correctly.

The following is abstracted from K&R (see References, below). The string `fmt` contains normal characters, which are passed through to the output string, and also special characters that operate on the arguments provided through `...`. Special characters start with a `%` and terminate with one of the letters in the set `difeEgGs%`. These letters denote the following types:

**d, i** Integer value

**f** Double precision value, in decimal notation of the form "`[-]mmm.ddd`". The number of decimal places is specified by the precision: the default is 6; a precision of 0 suppresses the decimal point.

**e, E** Double precision value, in decimal notation of the form `[-]m.ddde[+-]xx` or `[-]m.dddE[+-]xx`

**g, G** Double precision value, in `%e` or `%E` format if the exponent is less than -4 or greater than or equal to the precision, and `%f` format otherwise

**s** Character string

**%** Literal `%` (none of the formatting characters given below are permitted in this case)

In addition, between the initial `%` and the terminating conversion character there may be, in any order:

**m.n** Two numbers separated by a period, denoting the field width (`m`) and the precision (`n`)

**-** Left adjustment of converted argument in its field

**+** Always print number with sign

**a space** Prefix a space if the first number is not a sign

**0** For numbers, pad to the field width with leading zeros

**Value**

A character vector of length one. Character NAs are converted to `"NA"`.

**Author(s)**

Original code by Jonathan Rougier, `<J.C.Rougier@durham.ac.uk>`

**References**

Kernighan, B. W. and Ritchie, D. M. (1988) *The C Programming Language*. Second edition, Prentice Hall. describes the format options in table B-1 in the Appendix.

**See Also**

[formatC](#) for a way of formatting vectors of numbers in a similar fashion.

[paste](#) for another way of creating a vector combining text and values.

**Examples**

```
## be careful with the format: most things in R are floats

sprintf("%s is %f feet tall\n", "Sven", 7) # OK
try(sprintf("%s is %i feet tall\n", "Sven", 7)) # not OK
sprintf("%s is %i feet tall\n", "Sven", as.integer(7)) # OK again

## use a literal % :

sprintf("%.0f%% said yes (out of a sample of size %.0f)", 66.666, 3)

## various formats of pi :

sprintf("%f", pi)
sprintf("%.3f", pi)
sprintf("%1.0f", pi)
sprintf("%5.1f", pi)
sprintf("%05.1f", pi)
sprintf("%+f", pi)
sprintf("% f", pi)
sprintf("%-10f", pi)# left justified
sprintf("%e", pi)
sprintf("%E", pi)
sprintf("%g", pi)
sprintf("%g", 1e6 * pi) # -> exponential
sprintf("%.9g", 1e6 * pi) # -> "fixed"
sprintf("%G", 1e-6 * pi)

## no truncation:
sprintf("%1.f",101)

## More sophisticated:

lapply(c("a", "ABC", "and an even longer one"),
       function(ch) sprintf("10-string '%10s'", ch))

sapply(1:18, function(n)
       sprintf(paste("e with %2d digits = %.",n,"g",sep=""),
              n, exp(1)))
```

**Description**

Single or double quote text by combining with appropriate single or double left and right quotation marks.

**Usage**

```
sQuote(x)
dQuote(x)
```

**Arguments**

`x` an R object, to be coerced to a character vector.

**Details**

The purpose of the functions is to provide a simple means of markup for quoting text to be used in the R output, e.g., in warnings or error messages.

The choice of the appropriate quotation marks depends on both the locale and the available character sets. Older Unix/X11 fonts displayed the grave accent (0x60) and the apostrophe (0x27) in a way that they could also be used as matching open and close single quotation marks. Using modern fonts, or non-Unix systems, these characters no longer produce matching glyphs. Unicode provides left and right single quotation mark characters (U+2018 and U+2019); if Unicode cannot be assumed, it seems reasonable to use the apostrophe as an unidirectional single quotation mark.

Similarly, Unicode has left and right double quotation mark characters (U+201C and U+201D); if only ASCII's typewriter characteristics can be employed, then the ASCII quotation mark (0x22) should be used as both the left and right double quotation mark.

`sQuote` and `dQuote` currently only provide unidirectional ASCII quotation style, but may be enhanced in the future.

**References**

Markus Kuhn, "ASCII and Unicode quotation marks". <http://www.cl.cam.ac.uk/~mgk25/ucs/quotes.html>

**Examples**

```
paste("argument", sQuote("x"), "must be non-zero")
```

---

stack

*Stack or Unstack Vectors from a Data Frame or List*

---

**Description**

Stacking vectors concatenates multiple vectors into a single vector along with a factor indicating where each observation originated. Unstacking reverses this operation.

**Usage**

```
stack(x, ...)
## Default S3 method:
stack(x, ...)
## S3 method for class 'data.frame':
stack(x, select, ...)

unstack(x, ...)
## Default S3 method:
unstack(x, form, ...)
## S3 method for class 'data.frame':
unstack(x, form = formula(x), ...)
```

**Arguments**

<code>x</code>	object to be stacked or unstacked
<code>select</code>	expression, indicating variables to select from a data frame
<code>form</code>	a two-sided formula whose left side evaluates to the vector to be unstacked and whose right side evaluates to the indicator of the groups to create. Defaults to <code>formula(x)</code> in <code>unstack.data.frame</code> .
<code>...</code>	further arguments passed to or from other methods.

**Details**

The `stack` function is used to transform data available as separate columns in a data frame or list into a single column that can be used in an analysis of variance model or other linear model. The `unstack` function reverses this operation.

**Value**

`unstack` produces a list of columns according to the formula `form`. If all the columns have the same length, the resulting list is coerced to a data frame.

`stack` produces a data frame with two columns

<code>values</code>	the result of concatenating the selected vectors in <code>x</code>
<code>ind</code>	a factor indicating from which vector in <code>x</code> the observation originated

**Author(s)**

Douglas Bates

**See Also**

[lm](#), [reshape](#)

## Examples

```
require(stats)
formula(PlantGrowth)      # check the default formula
pg <- unstack(PlantGrowth) # unstack according to this formula
pg
stack(pg)                  # now put it back together
stack(pg, select = -ctrl) # omitting one vector
```

Startup

*Initialization at Start of an R Session*

## Description

In R, the startup mechanism is as follows.

Unless `--no-environ` was given on the command line, R searches for user and site files to process for setting environment variables. The name of the site file is the one pointed to by the environment variable `R_ENVIRON`; if this is unset or empty, `‘$R_HOME/etc/Renviron.site’` is used (if it exists, which it does not in a “factory-fresh” installation). The user files searched for are `‘.Renviron’` in the current or in the user’s home directory (in that order). See **Details** for how the files are read.

Then R searches for the site-wide startup profile unless the command line option `--no-site-file` was given. The name of this file is taken from the value of the `R_PROFILE` environment variable. If this variable is unset, the default is `‘$R_HOME/etc/Rprofile.site’`, which is used if it exists (which it does not in a “factory-fresh” installation). This code is loaded into package **base**. Users need to be careful not to unintentionally overwrite objects in **base**, and it is normally advisable to use `local` if code needs to be executed: see the examples.

Then, unless `--no-init-file` was given, R searches for a file called `‘.Rprofile’` in the current directory or in the user’s home directory (in that order) and sources it into the user workspace.

It then loads a saved image of the user workspace from `‘.RData’` if there is one (unless `--no-restore-data` was specified, or `--no-restore`, on the command line).

Next, if a function `.First` is found on the search path, it is executed as `.First()`. Finally, function `.First.sys()` in the **base** package is run. This calls `require` to attach the default packages specified by `options("defaultPackages")`.

A function `.First` (and `.Last`) can be defined in appropriate `‘.Rprofile’` or `‘Rprofile.site’` files or have been saved in `‘.RData’`. If you want a different set of packages than the default ones when you start, insert a call to `options` in the `‘.Rprofile’` or `‘Rprofile.site’` file. For example, `options(defaultPackages = character())` will attach no extra packages on startup. Alternatively, set `R_DEFAULT_PACKAGES=NULL` as an environment variable before running R. Using `options(defaultPackages = "")` or `R_DEFAULT_PACKAGES=""` enforces the R *system* default.

The commands history is read from the file specified by the environment variable `R_HISTFILE` (default `‘.Rhistory’`) unless `--no-restore-history` was specified (or `--no-restore`).

The command-line flag `--vanilla` implies `--no-site-file`, `--no-init-file`, `--no-restore` and `--no-environ`.

## Usage

```
.First <- function() { ..... }  
  
.Rprofile <startup file>
```

## Details

Note that there are two sorts of files used in startup: *environment files* which contain lists of environment variables to be set, and *profile files* which contain R code.

Lines in a site or user environment file should be either comment lines starting with #, or lines of the form `name=value`. The latter sets the environmental variable `name` to `value`, overriding an existing value. If `value` is of the form `${foo-bar}`, the value is that of the environmental variable `foo` if that exists and is set to a non-empty value, otherwise `bar`. This construction can be nested, so `bar` can be of the same form (as in `${foo-${bar-blah}}`).

Leading and trailing white space in `value` are stripped. `value` is processed in a similar way to a Unix shell. In particular quotes are stripped, and backslashes are removed except inside quotes.

## Historical notes

Prior to R version 1.4.0, the environment files searched were `‘.Renviron’` in the current directory, the file pointed to by `R_ENVIRON` if set, and `‘.Renviron’` in the user’s home directory.

Prior to R version 1.2.1, `‘.Rprofile’` was sourced after `‘.RData’` was loaded, although the documented order was as here.

The format for site and user environment files was changed in version 1.2.0. Older files are quite likely to work but may generate warnings on startup if they contained unnecessary `export` statements.

Values in environment files were not processed prior to version 1.4.0.

## Note

The file `‘$R_HOME/etc/Renviron’` is always read very early in the start-up processing. It contains environment variables set by R in the configure process. Values in that file can be overridden in site or user environment files: do not change `‘$R_HOME/etc/Renviron’` itself.

## See Also

[.Last](#) for final actions before termination.

For profiling code, see [Rprof](#).

## Examples

```
## Not run:  
# Example ~/.Renviron on Unix  
R_LIBS=~ /R/library  
PAGER=/usr/local/bin/less  
  
# Example .Renviron on Windows  
R_LIBS=C:/R/library
```

```

MY_TCLTK=yes
TCL_LIBRARY=c:/packages/Tcl/lib/tcl8.4

# Example of .Rprofile
options(width=65, digits=5)
options(show.signif.stars=FALSE)
ps.options(horizontal=FALSE)
set.seed(1234)
.First <- function() cat("\n Welcome to R!\n\n")
.Last <- function() cat("\n Goodbye!\n\n")

# Example of Rprofile.site
local({
  old <- getOption("defaultPackages")
  options(defaultPackages = c(old, "MASS"))
})

## if .Renviron contains
FOOBAR="coo\bar"doh\ex"abc\def' "

## then we get
> cat(Sys.getenv("FOOBAR"), "\n")
coo\bardoh\exabc"def'
## End(Not run)

```

---

stop

*Stop Function Execution*


---

## Description

`stop` stops execution of the current expression and executes an error action.

`geterrmessage` gives the last error message.

## Usage

```

stop(..., call. = TRUE)
geterrmessage()

```

## Arguments

`...` character vectors (which are pasted together with no separator), a condition object, or `NULL`.

`call.` logical, indicating if the call should become part of the error message.

## Details

The error action is controlled by error handlers established within the executing code and by the current default error handler set by `options(error=)`. The error is first signaled as if using `signalCondition()`. If there are no handlers or if all handlers return, then the error message is printed (if `options("show.error.messages")` is true) and the default error handler is used. The default behaviour (the NULL error-handler) in interactive use is to return to the top level prompt or the top level browser, and in non-interactive use to (effectively) call `q("no", status=1, runLast=FALSE)`. The default handler stores the error message in a buffer; it can be retrieved by `geterrmessage()`. It also stores a trace of the call stack that can be retrieved by `traceback()`.

Errors will be truncated to `getOption("warning.length")` characters, default 1000.

## Value

`geterrmessage` gives the last error message, as character string ending in `"\n"`.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

`warning`, `try` to catch errors and retry, and `options` for setting error handlers. `stopifnot` for validity testing. `tryCatch` and `withCallingHandlers` can be used to establish custom handlers while executing an expression.

## Examples

```
options(error = expression(NULL))
# don't stop on stop(.) << Use with CARE! >>

iter <- 12
if(iter > 10) stop("too many iterations")

tst1 <- function(...) stop("dummy error")
tst1(1:10, long, calling, expression)

tst2 <- function(...) stop("dummy error", call. = FALSE)
tst2(1:10, longcalling, expression, but.not.seen.in.Error)

options(error = NULL)# revert to default
```

---

 stopifnot

*Ensure the 'Truth' of R Expressions*


---

### Description

If any of the expressions in `...` are not `all` TRUE, `stop` is called, producing an error message indicating the *first* element of `...` which was not true.

### Usage

```
stopifnot(...)
```

### Arguments

`...` any number of (`logical`) R expressions which should evaluate to `TRUE`.

### Details

`stopifnot(A, B)` is conceptually equivalent to `{ if(!all(A)) stop(...) ; if(!all(B)) stop(...) }`.

### Value

(`NULL` if all statements in `...` are TRUE.)

### See Also

`stop`, `warning`.

### Examples

```
stopifnot(1 == 1, all.equal(pi, 3.14159265), 1 < 2) # all TRUE

m <- matrix(c(1,3,3,1), 2,2)
stopifnot(m == t(m), diag(m) == rep(1,2)) # all(.) |=> TRUE

op <- options(error = expression(NULL))
# "disable stop(.)" << Use with CARE! >>

stopifnot(all.equal(pi, 3.141593), 2 < 2, all(1:10 < 12), "a" < "b")
stopifnot(all.equal(pi, 3.1415927), 2 < 2, all(1:10 < 12), "a" < "b")

options(op)# revert to previous error handler
```

---

strptime *Date-time Conversion Functions to and from Character*

---

### Description

Functions to convert between character representations and objects of classes "POSIXlt" and "POSIXct" representing calendar dates and times.

### Usage

```
## S3 method for class 'POSIXct':
format(x, format = "", tz = "", usetz = FALSE, ...)
## S3 method for class 'POSIXlt':
format(x, format = "", usetz = FALSE, ...)

## S3 method for class 'POSIXt':
as.character(x, ...)

strftime(x, format="", usetz = FALSE, ...)
strptime(x, format)

ISODatetime(year, month, day, hour, min, sec, tz = "")
ISOdate(year, month, day, hour = 12, min = 0, sec = 0, tz = "GMT")
```

### Arguments

x	An object to be converted.
tz	A timezone specification to be used for the conversion. System-specific, but "" is the current time zone, and "GMT" is UTC.
format	A character string. The default is "%Y-%m-%d %H:%M:%S" if any component has a time component which is not midnight, and "%Y-%m-%d" otherwise.
...	Further arguments to be passed from or to other methods.
usetz	logical. Should the timezone be appended to the output? This is used in printing time, and as a workaround for problems with using "%Z" on most Linux systems.
year, month, day	numerical values to specify a day.
hour, min, sec	numerical values for a time within a day.

### Details

strftime is an alias for format.POSIXlt, and format.POSIXct first converts to class "POSIXct" by calling as.POSIXct. Note that only that conversion depends on the time zone.

The usual vector re-cycling rules are applied to x and format so the answer will be of length that of the longer of the vectors.

Locale-specific conversions to and from character strings are used where appropriate and available. This affects the names of the days and months, the AM/PM indicator (if used) and the separators in formats such as %x and %X.

The details of the formats are system-specific, but the following are defined by the ISO C / POSIX standard for `strptime` and are likely to be widely available. Any character in the format string other than the % escape sequences is interpreted literally (and %% gives %).

- %a** Abbreviated weekday name.
- %A** Full weekday name.
- %b** Abbreviated month name.
- %B** Full month name.
- %c** Date and time, locale-specific.
- %d** Day of the month as decimal number (01–31).
- %H** Hours as decimal number (00–23).
- %I** Hours as decimal number (01–12).
- %j** Day of year as decimal number (001–366).
- %m** Month as decimal number (01–12).
- %M** Minute as decimal number (00–59).
- %p** AM/PM indicator in the locale. Used in conjunction with %I and **not** with %H.
- %S** Second as decimal number (00–61), allowing for up to two leap-seconds.
- %U** Week of the year as decimal number (00–53) using the first Sunday as day 1 of week 1.
- %w** Weekday as decimal number (0–6, Sunday is 0).
- %W** Week of the year as decimal number (00–53) using the first Monday as day 1 of week 1.
- %x** Date, locale-specific.
- %X** Time, locale-specific.
- %y** Year without century (00–99). If you use this on input, which century you get is system-specific. So don't! Often values up to 69 (or 68) are prefixed by 20 and 70–99 by 19.
- %Y** Year with century.
- %z** (output only.) Offset from Greenwich, so -0800 is 8 hours west of Greenwich.
- %Z** (output only.) Time zone as a character string (empty if not available).

Where leading zeros are shown they will be used on output but are optional on input.

Also defined in the current standards but less widely implemented (e.g. not for output on Windows) are

- %F** Equivalent to %Y-%m-%d (the ISO 8601 date format).
- %g** The last two digits of the week-based year (see %V).
- %G** The week-based year (see %V) as a decimal number.
- %u** Weekday as a decimal number (1–7, Monday is 1).
- %v** Week of the year as decimal number (00–53). If the week (starting on Monday) containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1.

Other format specifiers in common use include

- %D** Locale-specific date format such as %m/%d/%y.
- %k** The 24-hour clock time with single digits preceded by a blank.
- %l** The 12-hour clock time with single digits preceded by a blank.
- %n** Newline on output, arbitrary whitespace on input.
- %r** The 12-hour clock time (using the locale's AM or PM).
- %R** Equivalent to %H:%M.
- %t** Newline on output, arbitrary whitespace on input.
- %T** Equivalent to %H:%M:%S.

There are also %O[dHImMSUVwWy] which may emit numbers in an alternative local-dependent format (e.g. roman numerals), and %E[cCyYxX] which can use an alternative 'era' (e.g. a different religious calendar). Which of these are supported is OS-dependent.

ISODatetime and ISOdate are convenience wrappers for strptime, that differ only in their defaults.

### Value

The format methods and strftime return character vectors representing the time.

strptime turns character representations into an object of class "POSIXlt".

ISODatetime and ISOdate return an object of class "POSIXct".

### Note

The default formats follow the rules of the ISO 8601 international standard which expresses a day as "2001-02-03" and a time as "14:01:02" using leading zeroes as here. The ISO form uses no space to separate dates and times.

If the date string does not specify the date completely, the returned answer may be system-specific. The most common behaviour is to assume that unspecified seconds, minutes or hours are zero, and a missing year, month or day is the current one. If it specifies a date incorrectly, reliable implementations will give an error and the date is reported as NA. Unfortunately some common implementations (such as 'libc') are unreliable and guess at the intended meaning.

If the timezone specified is invalid on your system, what happens is system-specific but it will probably be ignored.

OS facilities will probably not print years before 1CE (aka 1AD) correctly.

### References

International Organization for Standardization (1988, 1997, ...) *ISO 8601. Data elements and interchange formats – Information interchange – Representation of dates and times*. The 1997 version is available on-line at <ftp://ftp.qsl.net/pub/g1smd/8601v03.pdf>

### See Also

[DateTimeClasses](#) for details of the date-time classes; [locales](#) to query or set a locale.

Your system's help pages on strftime and strptime to see how to specify their formats.

**Examples**

```
## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y %Z")

## read in date info in format 'ddmmyyyy'
## This will give NA(s) in some locales; setting the C locale
## as in the commented lines will overcome this on most systems.
## lct <- Sys.getlocale("LC_TIME"); Sys.setlocale("LC_TIME", "C")
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- strptime(x, "%d%b%Y")
## Sys.setlocale("LC_TIME", lct)
z

## read in date/time info in format 'm/d/y h:m:s'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03", "16:56:26")
x <- paste(dates, times)
z <- strptime(x, "%m/%d/%y %H:%M:%S")
z
```

---

strsplit

---

*Split the Elements of a Character Vector*


---

**Description**

Split the elements of a character vector `x` into substrings according to the presence of substring `split` within them.

**Usage**

```
strsplit(x, split, extended = TRUE, fixed = FALSE, perl = FALSE)
```

**Arguments**

<code>x</code>	character vector, each element of which is to be split.
<code>split</code>	character vector containing <a href="#">regular expression</a> (s) (unless <code>fixed = TRUE</code> ) to use as “split”. If empty matches occur, in particular if <code>split</code> has length 0, <code>x</code> is split into single characters. If <code>split</code> has length greater than 1, it is re-cycled along <code>x</code> .
<code>extended</code>	logical. if <code>TRUE</code> , extended regular expression matching is used, and if <code>FALSE</code> basic regular expressions are used.
<code>fixed</code>	logical. If <code>TRUE</code> match string exactly, otherwise use regular expressions.
<code>perl</code>	logical. Should perl-compatible regexps be used? Has priority over <code>extended</code> .

**Details**

Arguments `x` and `split` will be coerced to character, so you will see uses with `split = NULL` to mean `split = character(0)`, including in the examples below.

Note that splitting into single characters can be done via `split=character(0)` or `split=""`; the two are equivalent as from R 1.9.0.

A missing value of `split` does not split the the corresponding element(s) of `x` at all.

**Value**

A list of length `length(x)` the *i*-th element of which contains the vector of splits of `x[i]`.

**Warning**

The standard regular expression code has been reported to be very slow or give errors when applied to extremely long character strings (tens of thousands of characters or more): the code used when `perl=TRUE` seems faster and more reliable for such usages.

**See Also**

[paste](#) for the reverse, [grep](#) and [sub](#) for string search and manipulation; further [nchar](#), [substr](#).

[regular expression](#) for the details of the pattern specification.

**Examples**

```
noquote(strsplit("A text I want to display with spaces", NULL)[[1]])

x <- c(as = "asfef", qu = "qwerty", "yuiop[", "b", "stuff.blah.yech")
# split x on the letter e
strsplit(x,"e")

unlist(strsplit("a.b.c", "."))
## [1] "" "" "" "" ""
## Note that 'split' is a regexp!
## If you really want to split on '.', use
unlist(strsplit("a.b.c", "\\."))
## [1] "a" "b" "c"
## or
unlist(strsplit("a.b.c", ".", fixed = TRUE))

## a useful function: rev() for strings
strReverse <- function(x)
  sapply(lapply(strsplit(x, NULL), rev), paste, collapse="")
strReverse(c("abc", "Statistics"))

## get the first names of the members of R-core
a <- readLines(file.path(R.home(), "AUTHORS"))[-(1:8)]
a <- a[(0:2)-length(a)]
(a <- sub(" .*", "", a))
# and reverse them
```

```
strReverse(a)
```

---

```
structure Attribute Specification
```

---

### Description

structure returns the given object with its attributes set.

### Usage

```
structure(.Data, ...)
```

### Arguments

.Data            an object which will have various attributes attached to it.  
 ...             attributes, specified in tag=value form, which will be attached to data.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### Examples

```
structure(1:6, dim = 2:3)
```

---

```
strwrap Wrap Character Strings to Format Paragraphs
```

---

### Description

Each character string in the input is first split into paragraphs (on lines containing whitespace only). The paragraphs are then formatted by breaking lines at word boundaries. The target columns for wrapping lines and the indentation of the first and all subsequent lines of a paragraph can be controlled independently.

### Usage

```
strwrap(x, width = 0.9 * getOption("width"), indent = 0, exdent = 0,  

       prefix = "", simplify = TRUE)
```

**Arguments**

<code>x</code>	a character vector
<code>width</code>	a positive integer giving the target column for wrapping lines in the output.
<code>indent</code>	a non-negative integer giving the indentation of the first line in a paragraph.
<code>exdent</code>	a non-negative integer specifying the indentation of subsequent lines in paragraphs.
<code>prefix</code>	a character string to be used as prefix for each line.
<code>simplify</code>	a logical. If TRUE, the result is a single character vector of line text; otherwise, it is a list of the same length as <code>x</code> the elements of which are character vectors of line text obtained from the corresponding element of <code>x</code> . (Hence, the result in the former case is obtained by unlisting that of the latter.)

**Details**

Whitespace in the input is destroyed. Double spaces after periods (thought as representing sentence ends) are preserved. Currently, possible sentence ends at line breaks are not considered specially.

Indentation is relative to the number of characters in the prefix string.

**Examples**

```
## Read in file 'THANKS'.
x <- paste(readLines(file.path(R.home(), "THANKS")), collapse = "\n")
## Split into paragraphs and remove the first three ones
x <- unlist(strsplit(x, "\n[ \t\n]*\n"))[-(1:3)]
## Join the rest
x <- paste(x, collapse = "\n\n")
## Now for some fun:
writeLines(strwrap(x, width = 60))
writeLines(strwrap(x, width = 60, indent = 5))
writeLines(strwrap(x, width = 60, exdent = 5))
writeLines(strwrap(x, prefix = "THANKS> "))
```

---

subset

*Subsetting Vectors and Data Frames*


---

**Description**

Return subsets of vectors or data frames which meet conditions.

**Usage**

```
subset(x, ...)

## Default S3 method:
subset(x, subset, ...)

## S3 method for class 'data.frame':
subset(x, subset, select, drop = FALSE, ...)
```

**Arguments**

<code>x</code>	object to be subsetted.
<code>subset</code>	logical expression.
<code>select</code>	expression, indicating columns to select from a data frame.
<code>drop</code>	passed on to <code>[]</code> indexing operator.
<code>...</code>	further arguments to be passed to or from other methods.

**Details**

For ordinary vectors, the result is simply `x[subset & !is.na(subset)]`.

For data frames, the `subset` argument works similarly on the rows. Note that `subset` will be evaluated in the data frame, so columns can be referred to (by name) as variables.

The `select` argument exists only for the method for data frames. It works by first replacing names in the selection expression with the corresponding column numbers in the data frame and then using the resulting integer vector to index the columns. This allows the use of the standard indexing conventions so that for example ranges of columns can be specified easily.

The `drop` argument is passed on to the indexing method for data frames.

**Value**

An object similar to `x` contain just the selected elements (for a vector), rows and columns (for a data frame), and so on.

**Author(s)**

Peter Dalgaard

**See Also**

[\[\], transform](#)

**Examples**

```
subset(airquality, Temp > 80, select = c(Ozone, Temp))
subset(airquality, Day == 1, select = -Temp)
subset(airquality, select = Ozone:Wind)

with(airquality, subset(Ozone, Temp > 80))
```

---

`substitute`*Substituting and Quoting Expressions*

---

### Description

`substitute` returns the parse tree for the (unevaluated) expression `expr`, substituting any variables bound in `env`.

`quote` simply returns its argument. The argument is not evaluated and can be any R expression.

### Usage

```
substitute(expr, env)
quote(expr)
```

### Arguments

<code>expr</code>	Any syntactically valid R expression
<code>env</code>	An environment or a list object. Defaults to the current evaluation environment.

### Details

The typical use of `substitute` is to create informative labels for data sets and plots. The `myplot` example below shows a simple use of this facility. It uses the functions `deparse` and `substitute` to create labels for a plot which are character string versions of the actual arguments to the function `myplot`.

Substitution takes place by examining each component of the parse tree as follows: If it is not a bound symbol in `env`, it is unchanged. If it is a promise object, i.e., a formal argument to a function or explicitly created using `delay()`, the expression slot of the promise replaces the symbol. If it is an ordinary variable, its value is substituted, unless `env` is `.GlobalEnv` in which case the symbol is left unchanged.

### Value

The `mode` of the result is generally `"call"` but may in principle be any type. In particular, single-variable expressions have `mode "name"` and constants have the appropriate base mode.

### Note

`Substitute` works on a purely lexical basis. There is no guarantee that the resulting expression makes any sense.

Substituting and quoting often causes confusion when the argument is `expression(...)`. The result is a call to the `expression` constructor function and needs to be evaluated with `eval` to give the actual expression object.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[missing](#) for argument “missingness”, [bquote](#) for partial substitution, [sQuote](#) and [dQuote](#) for adding quotation marks to strings.

**Examples**

```
(s.e <- substitute(expression(a + b), list(a = 1))) #> expression(1 + b)
(s.s <- substitute( a + b, list(a = 1))) #> 1 + b
c(mode(s.e), typeof(s.e)) # "call", "language"
c(mode(s.s), typeof(s.s)) # (the same)
# but:
(e.s.e <- eval(s.e)) #> expression(1 + b)
c(mode(e.s.e), typeof(e.s.e)) # "expression", "expression"

substitute(x <- x + 1, list(x=1)) # nonsense

myplot <- function(x, y)
  plot(x, y, xlab=deparse(substitute(x)),
       ylab=deparse(substitute(y)))

## Simple examples about lazy evaluation, etc:

f1 <- function(x, y = x) { x <- x + 1; y }
s1 <- function(x, y = substitute(x)) { x <- x + 1; y }
s2 <- function(x, y) { if(missing(y)) y <- substitute(x); x <- x + 1; y }
a <- 10
f1(a)# 11
s1(a)# 11
s2(a)# a
typeof(s2(a))# "symbol"
```

---

 substr

*Substrings of a Character Vector*


---

**Description**

Extract or replace substrings in a character vector.

**Usage**

```
substr(x, start, stop)
substring(text, first, last = 1000000)
substr(x, start, stop) <- value
substring(text, first, last = 1000000) <- value
```

**Arguments**

`x, text` a character vector  
`start, first` integer. The first element to be replaced.  
`stop, last` integer. The last element to be replaced.  
`value` a character vector, recycled if necessary.

**Details**

`substring` is compatible with `S`, with `first` and `last` instead of `start` and `stop`. For vector arguments, it expands the arguments cyclically to the length of the longest *provided* none are of zero length.

When extracting, if `start` is larger than the string length then "" is returned.

For the replacement functions, if `start` is larger than the string length then no replacement is done. If the portion to be replaced is longer than the replacement string, then only the portion the length of the string is replaced.

**Value**

For `substr`, a character vector of the same length as `x`.

For `substring`, a character vector of length the longest of the arguments.

**Note**

The S4 version of `substring<-` ignores `last`; this version does not.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`substring`.)

**See Also**

[strsplit](#), [paste](#), [nchar](#).

**Examples**

```
substr("abcdef",2,4)
substring("abcdef",1:6,1:6)
## strsplit is more efficient ...

substr(rep("abcdef",4),1:4,4:5)
x <- c("asfef", "qwerty", "yuiop[", "b", "stuff.blah.yech")
substr(x, 2, 5)
substring(x, 2, 4:6)

substring(x, 2) <- c("..", "+++")
x
```

sum

*Sum of Vector Elements*

---

**Description**

`sum` returns the sum of all the values present in its arguments. If `na.rm` is `FALSE` an NA value in any of the arguments will cause a value of NA to be returned, otherwise NA values are ignored.

**Usage**

```
sum(..., na.rm = FALSE)
```

**Arguments**

`...` numeric or complex vectors.  
`na.rm` logical. Should missing values be removed?

**Details**

This is a generic function: methods can be defined for it directly or via the [Summary](#) group generic.

**Value**

The sum. If all of `...` are of type integer, then so is the sum, and in that case the result will be NA (with a warning) if integer overflow occurs.

NB: the sum of an empty set is zero, by definition.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

summary*Object Summaries*

---

**Description**

`summary` is a generic function used to produce result summaries of the results of various model fitting functions. The function invokes particular [methods](#) which depend on the [class](#) of the first argument.

**Usage**

```
summary(object, ...)

## Default S3 method:
summary(object, ..., digits = max(3, getOption("digits")-3))
## S3 method for class 'data.frame':
summary(object, maxsum = 7,
         digits = max(3, getOption("digits")-3), ...)

## S3 method for class 'factor':
summary(object, maxsum = 100, ...)

## S3 method for class 'matrix':
summary(object, ...)
```

**Arguments**

object	an object for which a summary is desired.
maxsum	integer, indicating how many levels should be shown for <code>factors</code> .
digits	integer, used for number formatting with <code>signif()</code> (for <code>summary.default</code> ) or <code>format()</code> (for <code>summary.data.frame</code> ).
...	additional arguments affecting the summary produced.

**Details**

For `factors`, the frequency of the first `maxsum - 1` most frequent levels is shown, where the less frequent levels are summarized in "(Others)" (resulting in `maxsum` frequencies).

The functions `summary.lm` and `summary.glm` are examples of particular methods which summarise the results produced by `lm` and `glm`.

**Value**

The form of the value returned by `summary` depends on the class of its argument. See the documentation of the particular methods for details of what is produced by that method.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

`anova`, `summary.glm`, `summary.lm`.

**Examples**

```
summary(attenu, digits = 4) #-> summary.data.frame(...), default precision
summary(attenu $ station, maxsum = 20) #-> summary.factor(...)
```

```
lst <- unclass(attenu$station) > 20 # logical with NAs
## summary.default() for logicals -- different from *.factor:
summary(lst)
summary(as.factor(lst))
```

svd

*Singular Value Decomposition of a Matrix***Description**

Compute the singular-value decomposition of a rectangular matrix.

**Usage**

```
svd(x, nu = min(n, p), nv = min(n, p), LINPACK = FALSE)

La.svd(x, nu = min(n, p), nv = min(n, p),
       method = c("dgesdd", "dgesvd"))
```

**Arguments**

<code>x</code>	a matrix whose SVD decomposition is to be computed.
<code>nu</code>	the number of left singular vectors to be computed. This must be one of 0, <code>nrow(x)</code> and <code>ncol(x)</code> , except for <code>method = "dgesdd"</code> .
<code>nv</code>	the number of right singular vectors to be computed. This must be one of 0 and <code>ncol(x)</code> .
<code>LINPACK</code>	logical. Should LINPACK be used (for compatibility with R < 1.7.0)?
<code>method</code>	The LAPACK routine to use in the real case.

**Details**

The singular value decomposition plays an important role in many statistical techniques. `svd` and `La.svd` provide two slightly different interfaces. The main functions used are the LAPACK routines DGESDD and ZGESVD; `svd(LINPACK=TRUE)` provides an interface to the LINPACK routine DSVDC, purely for backwards compatibility.

`La.svd` provides an interface to both the LAPACK routines DGESVD and DGESDD. The latter is usually substantially faster if singular vectors are required: see <http://www.cs.berkeley.edu/~demmel/DOE2000/Report0100.html>. Most benefit is seen with an optimized BLAS system. Using `method="dgesdd"` requires IEEE 754 arithmetic. Should this not be supported on your platform, `method="dgesvd"` is used, with a warning.

Computing the singular vectors is the slow part for large matrices.

Unsuccessful results from the underlying LAPACK code will result in an error giving a positive error code: these can only be interpreted by detailed study of the FORTRAN code.

**Value**

The SVD decomposition of the matrix as computed by LINPACK,

$$X = UDV',$$

where  $U$  and  $V$  are orthogonal,  $V'$  means  $V$  *transposed*, and  $D$  is a diagonal matrix with the singular values  $D_{ii}$ . Equivalently,  $D = U'XV$ , which is verified in the examples, below.

The returned value is a list with components

d	a vector containing the singular values of $x$ .
u	a matrix whose columns contain the left singular vectors of $x$ , present if <code>nu &gt; 0</code>
v	a matrix whose columns contain the right singular vectors of $x$ , present if <code>nv &gt; 0</code> .

For `La.svd` the return value replaces `v` by `vt`, the (conjugated if complex) transpose of `v`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dongarra, J. J., Bunch, J. R., Moler, C. B. and Stewart, G. W. (1978) *LINPACK Users Guide*. Philadelphia: SIAM Publications.

Anderson. E. and ten others (1999) *LAPACK Users' Guide*. Third Edition. SIAM.

Available on-line at [http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html).

**See Also**

[eigen](#), [qr](#).

[capabilities](#) to test for IEEE 754 arithmetic.

**Examples**

```
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
X <- hilbert(9)[,1:6]
(s <- svd(X))
D <- diag(s$d)
s$u %**% D %**% t(s$v) # X = U D V'
t(s$u) %**% X %**% s$v # D = U' X V
```

---

`sweep`*Sweep out Array Summaries*

---

### Description

Return an array obtained from an input array by sweeping out a summary statistic.

### Usage

```
sweep(x, MARGIN, STATS, FUN=" - ", ...)
```

### Arguments

<code>x</code>	an array.
<code>MARGIN</code>	a vector of indices giving the extents of <code>x</code> which correspond to <code>STATS</code> .
<code>STATS</code>	the summary statistic which is to be swept out.
<code>FUN</code>	the function to be used to carry out the sweep. In the case of binary operators such as <code>" / "</code> etc., the function name must be quoted.
<code>...</code>	optional arguments to <code>FUN</code> .

### Value

An array with the same shape as `x`, but with the summary statistics swept out.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[apply](#) on which `sweep` used to be based; [scale](#) for centering and scaling.

### Examples

```
require(stats) # for median
med.att <- apply(attitude, 2, median)
sweep(data.matrix(attitude), 2, med.att)# subtract the column medians
```

---

switch	<i>Select One of a List of Alternatives</i>
--------	---

---

**Description**

switch evaluates `EXPR` and accordingly chooses one of the further arguments (in `...`).

**Usage**

```
switch(EXPR, ...)
```

**Arguments**

<code>EXPR</code>	an expression evaluating to a number or a character string.
<code>...</code>	the list of alternatives, given explicitly.

**Details**

If the value of `EXPR` is an integer between 1 and `nargs() - 1` then the corresponding element of `...` is evaluated and the result returned.

If `EXPR` returns a character string then that string is used to match the names of the elements in `...`. If there is an exact match then that element is evaluated and returned if there is one, otherwise the next element is chosen, e.g., `switch("cc", a=1, cc=, d=2)` evaluates to 2.

In the case of no match, if there's a further argument in `switch` that one is returned, otherwise `NULL`.

**Warning**

Beware of partial matching: an alternative `E = foo` will match the first argument `EXPR` unless that is named. See the examples for good practice in naming the first argument.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```
require(stats)
centre <- function(x, type) {
  switch(type,
    mean = mean(x),
    median = median(x),
    trimmed = mean(x, trim = .1))
}
x <- rcauchy(10)
centre(x, "mean")
centre(x, "median")
```

```

centre(x, "trimmed")

ccc <- c("b","QQ","a","A","bb")
for(ch in ccc)
  cat(ch,":",switch(EXPR = ch, a=1,      b=2:3), "\n")
for(ch in ccc)
  cat(ch,":",switch(EXPR = ch, a=, A=1, b=2:3, "Otherwise: last"),"\n")

## Numeric EXPR don't allow an 'otherwise':
for(i in c(-1:3,9)) print(switch(i, 1,2,3,4))

```

---

Syntax

---

*Operator Syntax*


---

## Description

Outlines R syntax and gives the precedence of operators

## Details

The following unary and binary operators are defined. They are listed in precedence groups, from highest to lowest.

[ [ [	indexing
::	name space/variable name separator
\$ @	component / slot extraction
^	exponentiation (right to left)
- +	unary minus and plus
:	sequence operator
%any%	special operators
* /	multiply, divide
+ -	(binary) add, subtract
< > <= >= == !=	ordering and comparison
!	negation
& &&	and
	or
~	as in formulae
-> ->>	rightwards assignment
=	assignment (right to left)
<- <<-	assignment (right to left)
?	help (unary and binary)

Within an expression operators of equal precedence are evaluated from left to right except where indicated.

The links in the **See Also** section covers most other aspects of the basic syntax.

**Note**

There are substantial precedence differences between R and S. In particular, in S `?` has the same precedence as `+` `-` and `&` `&&` `|` `||` have equal precedence.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[Arithmetic](#), [Comparison](#), [Control](#), [Extract](#), [Logic](#), [Paren](#)

The *R Language Definition* manual.

---

Sys.getenv                      *Get Environment Variables*

---

**Description**

`Sys.getenv` obtains the values of the environment variables named by `x`.

**Usage**

```
Sys.getenv(x)
```

**Arguments**

`x`                      a character vector, or missing

**Value**

A vector of the same length as `x`, with the variable names as its names attribute. Each element holds the value of the environment variable named by the corresponding component of `x` (or `" "` if no environment variable with that name was found).

On most platforms `Sys.getenv()` will return a named vector giving the values of all the environment variables.

**See Also**

[Sys.putenv](#), [getwd](#) for the working directory.

**Examples**

```
Sys.getenv(c("R_HOME", "R_PAPERSIZE", "R_PRINTCMD", "HOST"))
```

---

`Sys.info`*Extract System and User Information*

---

**Description**

Reports system and user information.

**Usage**

```
Sys.info()
```

**Details**

This function is not implemented on all R platforms, and returns `NULL` when not available. Where possible it is based on POSIX system calls.

`Sys.info()` returns details of the platform R is running on, whereas `R.version` gives details of the platform R was built on: they may well be different.

**Value**

A character vector with fields

<code>sysname</code>	The operating system.
<code>release</code>	The OS release.
<code>version</code>	The OS version.
<code>nodename</code>	A name by which the machine is known on the network (if any).
<code>machine</code>	A concise description of the hardware.
<code>login</code>	The user's login name, or "unknown" if it cannot be ascertained.
<code>user</code>	The name of the real user ID, or "unknown" if it cannot be ascertained.

The first five fields come from the `uname(2)` system call. The login name comes from `getlogin(2)`, and the user name from `getpwuid(getuid())`

**Note**

The meaning of OS “release” and “version” is highly system-dependent and there is no guarantee that the node or login or user names will be what you might reasonably expect. (In particular on some Linux distributions the login name is unknown from sessions with re-directed inputs.)

**See Also**

[.Platform](#), and [R.version](#).

**Examples**

```
Sys.info()  
## An alternative (and probably better) way to get the login name on Unix  
Sys.getenv("LOGNAME")
```

---

`sys.parent`*Functions to Access the Function Call Stack*

---

### Description

These functions provide access to [environments](#) (“frames” in S terminology) associated with functions further up the calling stack.

### Usage

```
sys.call(which = 0)
sys.frame(which = 0)
sys.nframe()
sys.function(n = 0)
sys.parent(n = 1)

sys.calls()
sys.frames()
sys.parents()
sys.on.exit()
sys.status()
parent.frame(n = 1)
```

### Arguments

<code>which</code>	the frame number if non-negative, the number of generations to go back if negative. (See the Details section.)
<code>n</code>	the number of frame generations to go back.

### Details

[.GlobalEnv](#) is given number 0 in the list of frames. Each subsequent function evaluation increases the frame stack by 1 and the environment for evaluation of that function is returned by `sys.frame` with the appropriate index.

The parent of a function evaluation is the environment in which the function was called. It is not necessarily numbered one less than the frame number of the current evaluation, nor is it the environment within which the function was defined. `sys.parent` returns the number of the parent frame if `n` is 1 (the default), the grandparent if `n` is 2, and so on. `sys.frame` returns the environment associated with a given frame number.

`sys.call` and `sys.frame` both accept integer values for the argument `which`. Non-negative values of `which` are normal frame numbers whereas negative values are counted back from the frame number of the current evaluation.

`sys.nframe` returns the number of the current frame in that list. `sys.function` gives the definition of the function currently being evaluated in frame `n`.

`sys.frames` gives a list of all the active frames and `sys.parents` gives the indices of the parent frames of each of the frames.

Notice that even though the `sys.xxx` functions (except `sys.status`) are interpreted, their contexts are not counted nor are they reported. There is no access to them.

`sys.status()` returns a list with components `sys.calls`, `sys.parents` and `sys.frames`.

`sys.on.exit()` retrieves the expression stored for use by `on.exit` in the function currently being evaluated. (Note that this differs from S, which returns a list of expressions for the current frame and its parents.)

`parent.frame(n)` is a convenient shorthand for `sys.frame(sys.parent(n))` (implemented slightly more efficiently).

### Note

The argument to `sys.function` would more logically be `which` rather than `n`. This may be changed in future versions.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (not `parent.frame`.)

### See Also

[eval](#) for the usage of `sys.frame` and `parent.frame`.

### Examples

```
ff <- function(x) gg(x)
gg <- function(y) sys.status()
str(ff(1))

gg <- function(y) {
  ggg <- function() {
    cat("current frame is", sys.nframe(), "\n")
    cat("parents are", sys.parents(), "\n")
    print(sys.function(0)) # ggg
    print(sys.function(2)) # gg
  }
  if(y > 0) gg(y-1) else ggg()
}
gg(3)

t1 <- function() {
  aa <- "here"
  t2 <- function() {
    ## in frame 2 here
    cat("current frame is", sys.nframe(), "\n")
    str(sys.calls()) ## list with two components t1() and t2()
    cat("parents are frame nos", sys.parents(), "\n") ## 0 1
    print(ls(envir=sys.frame(-1))) ## [1] "aa" "t2"
    invisible()
  }
}
```

```
    }
    t2()
  }
  t1()

test.sys.on.exit <- function() {
  on.exit(print(1))
  ex <- sys.on.exit()
  str(ex)
  cat("exiting...\n")
}
test.sys.on.exit()
## gives 'language print(1)', prints 1 on exit
```

---

Sys.putenv

*Set Environment Variables*

---

### Description

putenv sets environment variables (for other processes called from within R or future calls to [Sys.getenv](#) from this R process).

### Usage

```
Sys.putenv(...)
```

### Arguments

... arguments in name=value form, with value coercible to a character string.

### Details

Non-standard R names must be quoted: see the Examples section.

### Value

A logical vector of the same length as `x`, with elements being true if setting the corresponding variable succeeded.

### Note

Not all systems need support `Sys.putenv`.

### See Also

[Sys.getenv](#), [setwd](#) for the working directory.

### Examples

```
print(Sys.putenv("R_TEST"="testit", ABC=123))
Sys.getenv("R_TEST")
```

---

`Sys.sleep`*Suspend Execution for a Time Interval*

---

**Description**

Suspend execution of R expressions for a given number of seconds

**Usage**

```
Sys.sleep(time)
```

**Arguments**

`time`            The time interval to suspend execution for, in seconds.

**Details**

Using this function allows R to be given very low priority and hence not to interfere with more important foreground tasks. A typical use is to allow a process launched from R to set itself up and read its input files before R execution is resumed.

The intention is that this function suspends execution of R expressions but wakes the process up often enough to respond to GUI events, typically every 0.5 seconds.

There is no guarantee that the process will sleep for the whole of the specified interval, and it may well take slightly longer in real time to resume execution. The resolution of the time interval is system-dependent, but will normally be down to 0.02 secs or better.

**Value**

Invisible NULL.

**Note**

This function may not be implemented on all systems.

**Examples**

```
testit <- function(x)
{
  p1 <- proc.time()
  Sys.sleep(x)
  proc.time() - p1 # The cpu usage should be negligible
}
testit(3.7)
```

---

`sys.source`*Parse and Evaluate Expressions from a File*

---

**Description**

Parses expressions in the given file, and then successively evaluates them in the specified environment.

**Usage**

```
sys.source(file, envir = NULL, chdir = FALSE,
           keep.source = getOption("keep.source.pkgs"))
```

**Arguments**

<code>file</code>	a character string naming the file to be read from
<code>envir</code>	an R object specifying the environment in which the expressions are to be evaluated. May also be a list or an integer. The default value <code>NULL</code> corresponds to evaluation in the base environment. This is probably not what you want; you should typically supply an explicit <code>envir</code> argument.
<code>chdir</code>	logical; if <code>TRUE</code> , the R working directory is changed to the directory containing <code>file</code> for evaluating.
<code>keep.source</code>	logical. If <code>TRUE</code> , functions “keep their source” including comments, see <a href="#">options</a> ( <code>keep.source = *</code> ) for more details.

**Details**

For large files, `keep.source = FALSE` may save quite a bit of memory. In order for the code being evaluated to use the correct environment (for example, in global assignments), source code in packages should call [topenv\(\)](#), which will return the namespace, if any, the environment set up by `sys.source`, or the global environment if a saved image is being used.

**See Also**

[source](#), and [library](#) which uses `sys.source`.

---

`Sys.time`*Get Current Date, Time and Timezone*

---

**Description**

`Sys.time` and `Sys.Date` returns the system’s idea of the current date with and without time, and `Sys.timezone` returns the current time zone.

**Usage**

```

Sys.time()
Sys.Date()
Sys.timezone()

```

**Details**

`Sys.time` returns an absolute date-time value which can be converted in various time zones and may return different days.

`Sys.Date` returns the day in the current timezone.

**Value**

`Sys.time` returns an object of class "POSIXct" (see [DateTimeClasses](#)).

`Sys.Date` returns an object of class "Date" (see [Date](#)).

`Sys.timezone` returns an OS-specific character string, possibly an empty string.

**See Also**

[date](#) for the system time in a fixed-format character string.

**Examples**

```

Sys.time()
## locale-specific version of date()
format(Sys.time(), "%a %b %d %X %Y")

Sys.Date()

Sys.timezone()

```

---

system

*Invoke a System Command*

---

**Description**

`system` invokes the OS command specified by `command`.

**Usage**

```

system(command, intern = FALSE, ignore.stderr = FALSE)

```

**Arguments**

<code>command</code>	the system command to be invoked, as a string.
<code>intern</code>	a logical, indicates whether to make the output of the command an R object.
<code>ignore.stderr</code>	a logical indicating whether error messages (written to 'stderr') should be ignored.

**Details**

If `intern` is `TRUE` then `popen` is used to invoke the command and the output collected, line by line, into an R `character` vector which is returned as the value of `system`. Output lines of more than 8095 characters will be split.

If `intern` is `FALSE` then the C function `system` is used to invoke the command and the value returned by `system` is the exit status of this function.

`unix` is a *deprecated* alternative, available for backwards compatibility.

**Value**

If `intern=TRUE`, a character vector giving the output of the command, one line per character string. If the command could not be run or gives an error a R error is generated.

If `intern=FALSE`, the return value is an error code.

**See Also**

[.Platform](#) for platform specific variables.

**Examples**

```
# list all files in the current directory using the -F flag
## Not run: system("ls -F")

# t1 is a character vector, each one
# representing a separate line of output from who
t1 <- system("who", TRUE)

system("ls fizzlipuzzli", TRUE, TRUE)# empty since file doesn't exist
```

---

system.file

*Find Names of R System Files*


---

**Description**

Finds the full file names of files in packages etc.

**Usage**

```
system.file(..., package = "base", lib.loc = NULL)
```

**Arguments**

`...` character strings, specifying subdirectory and file(s) within some package. The default, none, returns the root of the package. Wildcards are not supported.

`package` a character string with the name of a single package. An error occurs if more than one package name is given.

`lib.loc` a character vector with path names of R libraries, or `NULL`. The default value of `NULL` corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.

### Value

A character vector of positive length, containing the file names that matched `...`, or the empty string, `"`, if none matched. If matching the root of a package, there is no trailing separator.

As a special case, `system.file()` gives the root of the **base** package only.

### See Also

[list.files](#)

### Examples

```
system.file()           # The root of the 'base' package
system.file(package = "stats") # The root of package 'stats'
system.file("INDEX")
system.file("help", "AnIndex", package = "splines")
```

---

system.time

*CPU Time Used*

---

### Description

Return CPU (and other) times that `expr` used.

### Usage

```
system.time(expr, gcFirst)
unix.time(expr, gcFirst)
```

### Arguments

<code>expr</code>	Valid R expression to be “timed”
<code>gcFirst</code>	Logical - should a garbage collection be performed immediately before the timing? Default is <code>FALSE</code> .

### Details

`system.time` calls the builtin `proc.time`, evaluates `expr`, and then calls `proc.time` once more, returning the difference between the two `proc.time` calls.

The values returned by the `proc.time` are (on Unix) those returned by the C library function `times(3v)`, if available.

`unix.time` is an alias of `system.time`, for compatibility reasons.

Timings of evaluations of the same expression can vary considerably depending on whether the evaluation triggers a garbage collection. When `gcFirst` is `TRUE` a garbage collection (`gc`) will be performed immediately before the evaluation of `expr`. This will usually produce more consistent timings.

### Value

A numeric vector of length 5 containing the user `cpu`, system `cpu`, `elapsed`, `subproc1`, `subproc2` times. The `subproc` times are the user and system `cpu` time used by child processes (and so are usually zero).

The resolution of the times will be system-specific; it is common for them to be recorded to of the order of 1/100 second, and `elapsed` time is rounded to the nearest 1/100.

### Note

It is possible to compile R without support for `system.time`, when all the values will be `NA`.

### See Also

`proc.time`, `time` which is for time series.

### Examples

```
require(stats)
system.time(for(i in 1:100) mad(runif(1000)))
## Not run:
exT <- function(n = 1000) {
  # Purpose: Test if system.time works ok;   n: loop size
  system.time(for(i in 1:n) x <- mean(rt(1000, df=4)))
}
#-- Try to interrupt one of the following (using Ctrl-C / Escape):
exT()           #- about 3 secs on a 1GHz PIII
system.time(exT())  #~ +/- same
## End(Not run)
```

---

t

*Matrix Transpose*


---

### Description

Given a matrix or `data.frame` `x`, `t` returns the transpose of `x`.

### Usage

```
t(x)
```

### Arguments

`x` a matrix or data frame, typically.

**Details**

A data frame is first coerced to a matrix: see [as.matrix](#). When `x` is a vector, it is treated as “column”, i.e., the result is a 1-row matrix.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[aperm](#) for permuting the dimensions of arrays.

**Examples**

```
a <- matrix(1:30, 5,6)
ta <- t(a) ##-- i.e., a[i, j] == ta[j, i] for all i,j :
for(j in seq(ncol(a)))
  if(! all(a[, j] == ta[j, ])) stop("wrong transpose")
```

---



---

*Cross Tabulation and Table Creation*


---

**Description**

`table` uses the cross-classifying factors to build a contingency table of the counts at each combination of factor levels.

**Usage**

```
table(..., exclude = c(NA, NaN), dnn = list.names(...),
      deparse.level = 1)
as.table(x, ...)
is.table(x)

## S3 method for class 'table':
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

**Arguments**

`...` objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted

`exclude` values to use in the `exclude` argument of `factor` when interpreting non-factor objects; if specified, levels to remove from all factors in `...`

`dnn` the names to be given to the dimensions in the result (the *dimnames names*).

`deparse.level` controls how the default `dnn` is constructed. See details.

<code>x</code>	an arbitrary R object, or an object inheriting from class "table" for the <code>as.data.frame</code> method.
<code>row.names</code>	a character vector giving the row names for the data frame.
<code>optional</code>	a logical controlling whether row names are set. Currently not used.

### Details

If the argument `dnn` is not supplied, the internal function `list.names` is called to compute the 'dimname names'. If the arguments in `...` are named, those names are used. For the remaining arguments, `deparse.level = 0` gives an empty name, `deparse.level = 1` uses the supplied argument if it is a symbol, and `deparse.level = 2` will deparse the argument.

Only when `exclude` is specified (i.e., not by default), will `table` drop levels of factor arguments potentially.

### Value

`table()` returns a *contingency table*, an object of class "table", an array of integer values.

There is a `summary` method for objects created by `table` or `xtabs`, which gives basic information and performs a chi-squared test for independence of factors (note that the function `chisq.test` in package `ctest` currently only handles 2-d tables).

`as.table` and `is.table` coerce to and test for contingency table, respectively.

The `as.data.frame` method for objects inheriting from class "table" can be used to convert the array-based representation of a contingency table to a data frame containing the classifying factors and the corresponding counts (the latter as component `Freq`). This is the inverse of `xtabs`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

Use `fTable` for printing (and more) of multidimensional tables.

### Examples

```
require(stats) # for rpois and xtabs
## Simple frequency distribution
table(rpois(100,5))
attach(warpbreaks)
## Check the design:
table(wool, tension)
detach()
table(state.division, state.region)

# simple two-way contingency table
with(airquality, table(cut(Temp, quantile(Temp)), Month))

a <- letters[1:3]
```

```

table(a, sample(a)) # dnn is c("a", "")
table(a, sample(a), deparse.level = 0) # dnn is c("", "")
table(a, sample(a), deparse.level = 2) # dnn is c("a", "sample(a)")

## xtabs() <-> as.data.frame.table() :
UCBAdmissions ## already a contingency table
DF <- as.data.frame(UCBAdmissions)
class(tab <- xtabs(Freq ~ ., DF)) # xtabs & table
## tab *is* "the same" as the original table:
all(tab == UCBAdmissions)
all.equal(dimnames(tab), dimnames(UCBAdmissions))

a <- rep(c(NA, 1/0:3), 10)
table(a)
table(a, exclude=NULL)
b <- factor(rep(c("A", "B", "C"), 10))
table(b)
table(b, exclude="B")
d <- factor(rep(c("A", "B", "C"), 10), levels=c("A", "B", "C", "D", "E"))
table(d, exclude="B")

## NA counting:
is.na(d) <- 3:4
d <- factor(d, exclude=NULL)
d[1:7]
table(d, exclude = NULL)

```

---

tabulate

*Tabulation for Vectors*


---

### Description

tabulate takes the integer-valued vector `bin` and counts the number of times each integer occurs in it.

### Usage

```
tabulate(bin, nbins = max(1, bin))
```

### Arguments

`bin` a numeric vector (of positive integers), or a factor.  
`nbins` the number of bins to be used.

### Details

tabulate is used as the basis of the `table` function.

If `bin` is a factor, its internal integer representation is tabulated.

If the elements of `bin` are numeric but not integers, they are truncated to the nearest integer.

**Value**

An integer vector (without names). There is a bin for each of the values 1, . . . , nbins; values outside that range are (silently) ignored.

**See Also**

[table](#), [factor](#).

**Examples**

```
tabulate(c(2,3,5))
tabulate(c(2,3,3,5), nbins = 10)
tabulate(c(-2,0,2,3,3,5)) # -2 and 0 are ignored
tabulate(c(-2,0,2,3,3,5), nbins = 3)
tabulate(factor(letters[1:10]))
```

---

tapply

*Apply a Function Over a “Ragged” Array*


---

**Description**

Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

**Usage**

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

**Arguments**

X	an atomic object, typically a vector.
INDEX	list of factors, each of same length as X.
FUN	the function to be applied. In the case of functions like +, %*%, etc., the function name must be quoted. If FUN is NULL, tapply returns a vector which can be used to subscript the multi-way array tapply normally produces.
...	optional arguments to FUN.
simplify	If FALSE, tapply always returns an array of mode "list". If TRUE (the default), then if FUN always returns a scalar, tapply returns an array with the mode of the scalar.

**Value**

When FUN is present, `tapply` calls FUN for each cell that has any data in it. If FUN returns a single atomic value for each cell (e.g., functions `mean` or `var`) and when `simplify` is TRUE, `tapply` returns a multi-way [array](#) containing the values. The array has the same number of dimensions as INDEX has components; the number of levels in a dimension is the number of levels (`nlevels()`) in the corresponding component of INDEX.

Note that contrary to S, `simplify = TRUE` always returns an array, possibly 1-dimensional.

If FUN does not return a single atomic value, `tapply` returns an array of mode [list](#) whose components are the values of the individual calls to FUN, i.e., the result is a list with a `dim` attribute.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

the convenience functions [by](#) and [aggregate](#) (using `tapply`); [apply](#), [lapply](#) with its versions [sapply](#) and [mapply](#).

**Examples**

```
require(stats)
groups <- as.factor(rbinom(32, n = 5, p = .4))
tapply(groups, groups, length) #- is almost the same as
table(groups)

## contingency table from data.frame : array with named dimnames
tapply(warbreaks$breaks, warbreaks[,-1], sum)
tapply(warbreaks$breaks, warbreaks[, 3, drop = FALSE], sum)

n <- 17; fac <- factor(rep(1:3, len = n), levels = 1:5)
table(fac)
tapply(1:n, fac, sum)
tapply(1:n, fac, sum, simplify = FALSE)
tapply(1:n, fac, range)
tapply(1:n, fac, quantile)

## example of ... argument: find quarterly means
tapply(presidents, cycle(presidents), mean, na.rm = TRUE)

ind <- list(c(1, 2, 2), c("A", "A", "B"))
table(ind)
tapply(1:3, ind) #-> the split vector
tapply(1:3, ind, sum)
```

---

taskCallback	<i>Add or remove a top-level task callback</i>
--------------	--

---

### Description

`addTaskCallback` registers an R function that is to be called each time a top-level task is completed.

`removeTaskCallback` un-registers a function that was registered earlier via `addTaskCallback`.

These provide low-level access to the internal/native mechanism for managing task-completion actions. One can use [taskCallbackManager](#) at the S-language level to manage S functions that are called at the completion of each task. This is easier and more direct.

### Usage

```
addTaskCallback(f, data = NULL, name = character(0))
removeTaskCallback(id)
```

### Arguments

<code>f</code>	the function that is to be invoked each time a top-level task is successfully completed. This is called with 5 or 4 arguments depending on whether <code>data</code> is specified or not, respectively. The return value should be a logical value indicating whether to keep the callback in the list of active callbacks or discard it.
<code>data</code>	if specified, this is the 5-th argument in the call to the callback function <code>f</code> .
<code>id</code>	a string or an integer identifying the element in the internal callback list to be removed. Integer indices are 1-based, i.e the first element is 1. The names of currently registered handlers is available using <a href="#">getTaskCallbackNames</a> and is also returned in a call to <a href="#">addTaskCallback</a> .
<code>name</code>	character: names to be used.

### Details

Top-level tasks are individual expressions rather than entire lines of input. Thus an input line of the form `expression1 ; expression2` will give rise to 2 top-level tasks.

A top-level task callback is called with the expression for the top-level task, the result of the top-level task, a logical value indicating whether it was successfully completed or not (always TRUE at present), and a logical value indicating whether the result was printed or not. If the `data` argument was specified in the call to `addTaskCallback`, that value is given as the fifth argument.

The callback function should return a logical value. If the value is FALSE, the callback is removed from the task list and will not be called again by this mechanism. If the function returns TRUE, it is kept in the list and will be called on the completion of the next top-level task.

**Value**

`addTaskCallback` returns an integer value giving the position in the list of task callbacks that this new callback occupies. This is only the current position of the callback. It can be used to remove the entry as long as no other values are removed from earlier positions in the list first.

`removeTaskCallback` returns a logical value indicating whether the specified element was removed. This can fail (i.e., return `FALSE`) if an incorrect name or index is given that does not correspond to the name or position of an element in the list.

**Note**

This is an experimental feature and the interface may be changed in the future.

There is also C-level access to top-level task callbacks to allow C routines rather than R functions be used.

**See Also**

`getTaskCallbackNames`      `taskCallbackManager`      <http://developer.r-project.org/TaskHandlers.pdf>

**Examples**

```
times <- function(total = 3, str="Task a") {
  ctr <- 0

  function(expr, value, ok, visible) {
    ctr <-< ctr + 1
    cat(str, ctr, "\n")
    if(ctr == total) {
      cat("handler removing itself\n")
    }
    return(ctr < total)
  }
}

# add the callback that will work for
# 4 top-level tasks and then remove itself.
n <- addTaskCallback(times(4))

# now remove it, assuming it is still first in the list.
removeTaskCallback(n)

## Not run:
# There is no point in running this
# as
addTaskCallback(times(4))

sum(1:10)
sum(1:10)
sum(1:10)
sum(1:10)
```

```
sum(1:10)
## End(Not run)
```

---

```
taskCallbackManager
```

*Create an R-level task callback manager*

---

## Description

This provides an entirely S-language mechanism for managing callbacks or actions that are invoked at the conclusion of each top-level task. Essentially, we register a single R function from this manager with the underlying, native task-callback mechanism and this function handles invoking the other R callbacks under the control of the manager. The manager consists of a collection of functions that access shared variables to manage the list of user-level callbacks.

## Usage

```
taskCallbackManager(handlers = list(), registered = FALSE,
                    verbose = FALSE)
```

## Arguments

handlers	this can be a list of callbacks in which each element is a list with an element named "f" which is a callback function, and an optional element named "data" which is the 5-th argument to be supplied to the callback when it is invoked. Typically this argument is not specified, and one uses <code>add</code> to register callbacks after the manager is created.
registered	a logical value indicating whether the <code>evaluate</code> function has already been registered with the internal task callback mechanism. This is usually <code>FALSE</code> and the first time a callback is added via the <code>add</code> function, the <code>evaluate</code> function is automatically registered. One can control when the function is registered by specifying <code>TRUE</code> for this argument and calling <code>addTaskCallback</code> manually.
verbose	a logical value, which if <code>TRUE</code> , causes information to be printed to the console about certain activities this dispatch manager performs. This is useful for debugging callbacks and the handler itself.

## Value

A list containing 6 functions:

add	register a callback with this manager, giving the function, an optional 5-th argument, an optional name by which the callback is stored in the list, and a <code>register</code> argument which controls whether the <code>evaluate</code> function is registered with the internal C-level dispatch mechanism if necessary.
remove	remove an element from the manager's collection of callbacks, either by name or position/index.

evaluate	the ‘real’ callback function that is registered with the C-level dispatch mechanism and which invokes each of the R-level callbacks within this manager’s control.
suspend	a function to set the suspend state of the manager. If it is suspended, none of the callbacks will be invoked when a task is completed. One sets the state by specifying a logical value for the <code>status</code> argument.
register	a function to register the <code>evaluate</code> function with the internal C-level dispatch mechanism. This is done automatically by the <code>add</code> function, but can be called manually.
callbacks	returns the list of callbacks being maintained by this manager.

**Note**

This is an experimental feature and the interface may be changed in the future.

**See Also**

[addTaskCallback](http://developer.r-project.org/TaskHandlers.pdf), [removeTaskCallback](http://developer.r-project.org/TaskHandlers.pdf), [getTaskCallbackNames](http://developer.r-project.org/TaskHandlers.pdf) \ <http://developer.r-project.org/TaskHandlers.pdf>

**Examples**

```
# create the manager
h <- taskCallbackManager()

# add a callback
h$add(function(expr, value, ok, visible) {
  cat("In handler\n")
  return(TRUE)
}, name = "simpleHandler")

# look at the internal callbacks.
getTaskCallbackNames()

# look at the R-level callbacks
names(h$callback())

getTaskCallbackNames()
removeTaskCallback("R-taskCallbackManager")
```

---

taskCallbackNames *Query the names of the current internal top-level task callbacks*

---

**Description**

This provides a way to get the names (or identifiers) for the currently registered task callbacks that are invoked at the conclusion of each top-level task. These identifiers can be used to remove a callback.

**Usage**

```
getTaskCallbackNames()
```

**Value**

A character vector giving the name for each of the registered callbacks which are invoked when a top-level task is completed successfully. Each name is the one used when registering the callbacks and returned as the in the call to [addTaskCallback](#).

**Note**

One can use [taskCallbackManager](#) to manage user-level task callbacks, i.e., S-language functions, entirely within the S language and access the names more directly.

**See Also**

[addTaskCallback](#), [removeTaskCallback](#), [taskCallbackManager](#) \ <http://developer.r-project.org/TaskHandlers.pdf>

**Examples**

```
n <- addTaskCallback(function(expr, value, ok, visible) {
  cat("In handler\n")
  return(TRUE)
}, name = "simpleHandler")

getTaskCallbackNames()

# now remove it by name
removeTaskCallback("simpleHandler")

h <- taskCallbackManager()
h$add(function(expr, value, ok, visible) {
  cat("In handler\n")
  return(TRUE)
}, name = "simpleHandler")
getTaskCallbackNames()
removeTaskCallback("R-taskCallbackManager")
```

---

tempfile

*Create Names for Temporary Files*


---

**Description**

tempfile returns a vector of character strings which can be used as names for temporary files.

**Usage**

```
tempfile(pattern = "file", tmpdir = tempdir())
tempdir()
```

**Arguments**

`pattern` a non-empty character vector giving the initial part of the name.  
`tmpdir` a non-empty character vector giving the directory name

**Details**

If `pattern` has length greater than one then the result is of the same length giving a temporary file name for each component of `pattern`.

The names are very likely to be unique among calls to `tempfile` in an R session and across simultaneous R sessions. The filenames are guaranteed not to be currently in use.

The file name is made of the `pattern`, the process number in hex and a random suffix in hex. By default, the filenames will be in the directory given by `tempdir()`. This will be a subdirectory of the directory given by the environment variable `TMPDIR` if set, otherwise `"/tmp"`.

**Value**

For `tempfile` a character vector giving the names of possible (temporary) files. Note that no files are generated by `tempfile`.

For `tempdir`, the path of the per-session temporary directory.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[unlink](#) for deleting files.

**Examples**

```
tempfile(c("ab", "a b c")) # give file name with spaces in!
```

---

`textConnection`      *Text Connections*

---

**Description**

Input and output text connections.

**Usage**

```
textConnection(object, open = "r", local = FALSE)
```

## Arguments

<code>object</code>	character. A description of the connection. For an input this is an R character vector object, and for an output connection the name for the R character vector to receive the output.
<code>open</code>	character. Either "r" (or equivalently " ") for an input connection or "w" or "a" for an output connection.
<code>local</code>	logical. Used only for output connections. If TRUE, output is assigned to a variable in the calling environment. Otherwise the global environment is used.

## Details

An input text connection is opened and the character vector is copied at time the connection object is created, and `close` destroys the copy.

An output text connection is opened and creates an R character vector of the given name in the user's workspace or in the calling environment, depending on the value of the `local` argument. This object will at all times hold the completed lines of output to the connection, and `isIncomplete` will indicate if there is an incomplete final line. Closing the connection will output the final line, complete or not. (A line is complete once it has been terminated by end-of-line, represented by "\n" in R.)

Opening a text connection with `mode = "a"` will attempt to append to an existing character vector with the given name in the user's workspace or the calling environment. If none is found (even if an object exists of the right name but the wrong type) a new character vector will be created, with a warning.

You cannot `seek` on a text connection, and `seek` will always return zero as the position.

## Value

A connection object of class "textConnection" which inherits from class "connection".

## Note

As output text connections keep the character vector up to date line-by-line, they are relatively expensive to use, and it is often better to use an anonymous `file()` connection to collect output.

On platforms where `vsnprintf` does not return the needed length of output (e.g., Windows) there is a 100,000 character limit on the length of line for output connections: longer lines will be truncated with a warning.

## See Also

[connections](#), [showConnections](#), [pushBack](#), [capture.output](#).

## Examples

```
zz <- textConnection(LETTERS)
readLines(zz, 2)
scan(zz, "", 4)
pushBack(c("aa", "bb"), zz)
```

```

scan(zz, "", 4)
close(zz)

zz <- textConnection("foo", "w")
writeLines(c("testit1", "testit2"), zz)
cat("testit3 ", file=zz)
isIncomplete(zz)
cat("testit4\n", file=zz)
isIncomplete(zz)
close(zz)
foo

## Not run:
# capture R output: use part of example from help(lm)
zz <- textConnection("foo", "w")
ctl <- c(4.17, 5.58, 5.18, 6.11, 4.5, 4.61, 5.17, 4.53, 5.33, 5.14)
trt <- c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
sink(zz)
anova(lm.D9 <- lm(weight ~ group))
cat("\nSummary of Residuals:\n\n")
summary(resid(lm.D9))
sink()
close(zz)
cat(foo, sep = "\n")
## End(Not run)

```

---

tilde

*Tilde Operator*


---

## Description

Tilde is used to separate the left- and right-hand sides in model formula.

## Usage

```
y ~ model
```

## Arguments

`y`, `model`      symbolic expressions.

## References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models*. Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

[formula](#)

---

`toString`*toString Converts its Argument to a Character String*

---

### Description

This is a helper function for `format`. It converts its argument to a string. If the argument is a vector then its elements are concatenated with a `,` as a separator. Most methods should honor the width argument. The minimum value for `width` is six.

### Usage

```
toString(x, ...)  
  
## Default S3 method:  
toString(x, width, ...)
```

### Arguments

<code>x</code>	The object to be converted.
<code>width</code>	The returned value is at most the first <code>width</code> characters.
<code>...</code>	Optional arguments for methods.

### Value

A character vector of length 1 is returned.

### Author(s)

Robert Gentleman

### See Also

[format](#)

### Examples

```
x <- c("a", "b", "aaaaaaaaaaaa")  
toString(x)  
toString(x, width=8)
```

---

trace *Interactive Tracing and Debugging of Calls to a Function or Method*

---

### Description

A call to `trace` allows you to insert debugging code (e.g., a call to `browser` or `recover`) at chosen places in any function. A call to `untrace` cancels the tracing. Specified methods can be traced the same way, without tracing all calls to the function. Trace code can be any R expression. Tracing can be temporarily turned on or off globally by calling `tracingState`.

### Usage

```
trace(what, tracer, exit, at, print, signature,
      where = topenv(parent.frame()), edit = FALSE)
untrace(what, signature = NULL, where = topenv(parent.frame()))

tracingState(on = NULL)
```

### Arguments

<code>what</code>	The name (quoted or not) of a function to be traced or untraced. More than one name can be given in the quoted form, and the same action will be applied to each one.
<code>tracer</code>	Either a function or an unevaluated expression. The function will be called or the expression will be evaluated either at the beginning of the call, or before those steps in the call specified by the argument <code>at</code> . See the details section.
<code>exit</code>	Either a function or an unevaluated expression. The function will be called or the expression will be evaluated on exiting the function. See the details section.
<code>at</code>	optional numeric vector. If supplied, <code>tracer</code> will be called just before the corresponding step in the body of the function. See the details section.
<code>print</code>	If <code>TRUE</code> (as per default), a descriptive line is printed before any trace expression is evaluated.
<code>signature</code>	If this argument is supplied, it should be a signature for a method for function <code>what</code> . In this case, the method, and <i>not</i> the function itself, is traced.
<code>edit</code>	For complicated tracing, such as tracing within a loop inside the function, you will need to insert the desired calls by editing the body of the function. If so, supply the <code>edit</code> argument either as <code>TRUE</code> , or as the name of the editor you want to use. Then <code>trace()</code> will call <code>edit</code> and use the version of the function after you edit it. See the details section for additional information.
<code>where</code>	where to look for the function to be traced; by default, the top-level environment of the call to <code>trace</code> .  An important use of this argument is to trace a function when it is called from a package with a namespace. The current namespace mechanism imports the functions to be called (with the exception of functions in the base package). The functions being called are <i>not</i> the same objects seen from the top-level (in

general, the imported packages may not even be attached). Therefore, you must ensure that the correct versions are being traced. The way to do this is to set argument `where` to a function in the namespace. The tracing computations will then start looking in the environment of that function (which will be the namespace of the corresponding package). (Yes, it's subtle, but the semantics here are central to how namespaces work in R.)

on logical; a call to `tracingState` returns `TRUE` if tracing is globally turned on, `FALSE` otherwise. An argument of one or the other of those values sets the state. If the tracing state is `FALSE`, none of the trace actions will actually occur (used, for example, by debugging functions to shut off tracing during debugging).

## Details

The `trace` function operates by constructing a revised version of the function (or of the method, if signature is supplied), and assigning the new object back where the original was found. If only the `what` argument is given, a line of trace printing is produced for each call to the function (back compatible with the earlier version of `trace`).

The object constructed by `trace` is from a class that extends "function" and which contains the original, untraced version. A call to `untrace` re-assigns this version.

If the argument `tracer` or `exit` is the name of a function, the tracing expression will be a call to that function, with no arguments. This is the easiest and most common case, with the functions `browser` and `recover` the likeliest candidates; the former browses in the frame of the function being traced, and the latter allows browsing in any of the currently active calls.

The `tracer` or `exit` argument can also be an unevaluated expression (such as returned by a call to `quote` or `substitute`). This expression itself is inserted in the traced function, so it will typically involve arguments or local objects in the traced function. An expression of this form is useful if you only want to interact when certain conditions apply (and in this case you probably want to supply `print=FALSE` in the call to `trace` also).

When the `at` argument is supplied, it should be a vector of integers referring to the substeps of the body of the function (this only works if the body of the function is enclosed in `{ . . . }`). In this case `tracer` is *not* called on entry, but instead just before evaluating each of the steps listed in `at`. (Hint: you don't want to try to count the steps in the printed version of a function; instead, look at `as.list(body(f))` to get the numbers associated with the steps in function `f`.)

An intrinsic limitation in the `exit` argument is that it won't work if the function itself uses `on.exit`, since the existing calls will override the one supplied by `trace`.

Tracing does not nest. Any call to `trace` replaces previously traced versions of that function or method (except for edited versions as discussed below), and `untrace` always restores an untraced version. (Allowing nested tracing has too many potentials for confusion and for accidentally leaving traced versions behind.)

When the `edit` argument is used repeatedly with no call to `untrace` on the same function or method in between, the previously edited version is retained. If you want to throw away all the previous tracing and then edit, call `untrace` before the next call to `trace`. Editing may be combined with automatic tracing; just supply the other arguments such as `tracer`, and the `edit` argument as well. The `edit=TRUE` argument uses the default editor (see `edit`).

Tracing primitive functions (builtins and specials) from the base package works, but only by a special mechanism and not very informatively. Tracing a primitive causes the primitive to be replaced

by a function with argument `...` (only). You can get a bit of information out, but not much. A warning message is issued when `trace` is used on a primitive.

The practice of saving the traced version of the function back where the function came from means that tracing carries over from one session to another, *if* the traced function is saved in the session image. (In the next session, `untrace` will remove the tracing.) On the other hand, functions that were in a package, not in the global environment, are not saved in the image, so tracing expires with the session for such functions.

Tracing a method is basically just like tracing a function, with the exception that the traced version is stored by a call to `setMethod` rather than by direct assignment, and so is the untraced version after a call to `untrace`.

The version of `trace` described here is largely compatible with the version in S-Plus, although the two work by entirely different mechanisms. The S-Plus `trace` uses the session frame, with the result that tracing never carries over from one session to another (R does not have a session frame). Another relevant distinction has nothing directly to do with `trace`: The browser in S-Plus allows changes to be made to the frame being browsed, and the changes will persist after exiting the browser. The R browser allows changes, but they disappear when the browser exits. This may be relevant in that the S-Plus version allows you to experiment with code changes interactively, but the R version does not. (A future revision may include a “destructive” browser for R.)

### Value

The traced function(s) name(s). The relevant consequence is the assignment that takes place.

### Note

The version of function tracing that includes any of the arguments except for the function name requires the `methods` package (because it uses special classes of objects to store and restore versions of the traced functions).

If `methods` dispatch is not currently on, `trace` will load the `methods` namespace, but will not put the `methods` package on the search list.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`browser` and `recover`, the likeliest tracing functions; also, `quote` and `substitute` for constructing general expressions.

### Examples

```
if(.isMethodsDispatchOn()) { # trace needs method package attached.

f <- function(x, y) {
  y <- pmax(y, .001)
  x ^ y
}
```

```

## arrange to call the browser on entering and exiting
## function f
trace("f", browser, exit = browser)

## instead, conditionally assign some data, and then browse
## on exit, but only then. Don't bother me otherwise

trace("f", quote(if(any(y < 0)) yOrig <- y),
      exit = quote(if(exists("yOrig")) browser()),
      print = FALSE)

## trace a utility function, with recover so we
## can browse in the calling functions as well.

trace("as.matrix", recover)

## turn off the tracing

untrace(c("f", "as.matrix"))

## Not run:
## trace calls to the function lm() that come from the nlme package
## (The function nlme is in that package, and the package has a namespace,
## so the where= argument must be used to get the right version of lm)

trace(lm, exit = recover, where = nlme)
## End(Not run)
}

```

---

traceback

---

*Print Call Stack of Last Error*


---

## Description

`traceback()` prints the call stack of the last uncaught error, i.e., the sequence of calls that lead to the error. This is useful when an error occurs with an unidentifiable error message.

## Usage

```
traceback()
```

## Details

The stack is stored as a list in `.Traceback`, which `traceback` prints in a user-friendly format.

Errors which are caught *via* `try` or `tryCatch` do not generate a traceback, so what is printed is the call sequence for the last uncaught error, and not necessarily the last error.

**Value**

`traceback()` returns nothing, but prints the departed call stack deepest call first. The calls may print on more than one line, and the first line is labelled by the frame number.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```
foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a.variable.which.does.not.exist }
## Not run:
foo(2) # gives a strange error
traceback()
## End(Not run)
## 2: bar(2)
## 1: foo(2)
bar
## Ah, this is the culprit ...
```

---

transform

*Transform an Object, for Example a Data Frame*


---

**Description**

`transform` is a generic function, which—at least currently—only does anything useful with data frames. `transform.default` converts its first argument to a data frame if possible and calls `transform.data.frame`.

**Usage**

```
transform(x, ...)
```

**Arguments**

<code>x</code>	The object to be transformed
<code>...</code>	Further arguments of the form <code>tag=value</code>

**Details**

The `...` arguments to `transform.data.frame` are tagged vector expressions, which are evaluated in the data frame `x`. The tags are matched against `names(x)`, and for those that match, the value replace the corresponding variable in `x`, and the others are appended to `x`.

**Value**

The modified value of `x`.

**Note**

If some of the values are not vectors of the appropriate length, you deserve whatever you get!

**Author(s)**

Peter Dalgaard

**See Also**

[subset](#), [list](#), [data.frame](#)

**Examples**

```
transform(airquality, Ozone = -Ozone)
transform(airquality, new = -Ozone, Temp = (Temp-32)/1.8)

attach(airquality)
transform(Ozone, logOzone = log(Ozone)) # marginally interesting ...
detach(airquality)
```

---

Trig

*Trigonometric Functions*

---

**Description**

These functions give the obvious trigonometric functions. They respectively compute the cosine, sine, tangent, arc-cosine, arc-sine, arc-tangent, and the two-argument arc-tangent.

**Usage**

```
cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)
atan2(y, x)
```

**Arguments**

`x`, `y`            numeric vector

**Details**

The arc-tangent of two arguments `atan2(y, x)` returns the angle between the x-axis and the vector from the origin to  $(x, y)$ , i.e., for positive arguments `atan2(y, x) == atan(y/x)`.

Angles are in radians, not degrees (i.e., a right angle is  $\pi/2$ ).

All except `atan2` are generic functions: methods can be defined for them individually or via the [Math](#) group generic.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

try

*Try an Expression Allowing Error Recovery*

---

## Description

`try` is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery.

## Usage

```
try(expr, silent = FALSE)
```

## Arguments

<code>expr</code>	an R expression to try.
<code>silent</code>	logical: should the report of error messages be suppressed?

## Details

`try` evaluates an expression and traps any errors that occur during the evaluation. `try` establishes a handler for errors that uses the default error handling protocol. It also establishes a `tryRestart` restart that can be used by `invokeRestart`.

## Value

The value of the expression if `expr` is evaluated without error, but an invisible object of class "try-error" containing the error message if it fails. The normal error handling will print the same message unless `options("show.error.messages")` is false or the call includes `silent = TRUE`.

## See Also

[options](#) for setting error handlers and suppressing the printing of error messages; [geterrmessage](#) for retrieving the last error message. `tryCatch` provides another means of catching and handling errors.

**Examples**

```

## this example will not work correctly in example(try), but
## it does work correctly if pasted in
options(show.error.messages = FALSE)
try(log("a"))
print(.Last.value)
options(show.error.messages = TRUE)

## alternatively,
print(try(log("a"), TRUE))

## run a simulation, keep only the results that worked.
set.seed(123)
x <- rnorm(50)
doit <- function(x)
{
  x <- sample(x, replace=TRUE)
  if(length(unique(x)) > 30) mean(x)
  else stop("too few unique points")
}
## alternative 1
res <- lapply(1:100, function(i) try(doit(x), TRUE))
## alternative 2
## Not run:
res <- vector("list", 100)
for(i in 1:100) res[[i]] <- try(doit(x), TRUE)
## End(Not run)
unlist(res[sapply(res, function(x) !inherits(x, "try-error"))])

```

---

type.convert

*Type Conversion on Character Variables*


---

**Description**

Convert a character vector to logical, integer, numeric, complex or factor as appropriate.

**Usage**

```
type.convert(x, na.strings = "NA", as.is = FALSE, dec = ".")
```

**Arguments**

x	a character vector.
na.strings	a vector of strings which are to be interpreted as <a href="#">NA</a> values. Blank fields are also considered to be missing values in logical, integer, numeric or complex vectors.
as.is	logical. See Details.
dec	the character to be assumed for decimal points.

**Details**

This is principally a helper function for `read.table`. Given a character vector, it attempts to convert it to logical, integer, numeric or complex, and failing that converts it to factor unless `as.is = TRUE`. The first type that can accept all the non-missing values is chosen.

Vectors which are entirely missing values are converted to logical, since NA is primarily logical.

**Value**

A vector of the selected class, or a factor.

**See Also**

[read.table](#)

---

typeof

*The Type of an Object*

---

**Description**

`typeof` determines the (R internal) type or storage mode of any object

**Usage**

```
typeof(x)
```

**Arguments**

`x` any R object.

**Value**

A character string.

**See Also**

[mode](#), [storage.mode](#).

**Examples**

```
typeof(2)
mode(2)
```

---

`unique`*Extract Unique Elements*

---

### Description

`unique` returns a vector, data frame or array like `x` but with duplicate elements removed.

### Usage

```
unique(x, incomparables = FALSE, ...)  
  
## S3 method for class 'array':  
unique(x, incomparables = FALSE, MARGIN = 1, ...)
```

### Arguments

<code>x</code>	an atomic vector or a data frame or an array.
<code>incomparables</code>	a vector of values that cannot be compared. Currently, <code>FALSE</code> is the only possible value, meaning that all values can be compared.
<code>...</code>	arguments for particular methods.
<code>MARGIN</code>	the array margin to be held fixed: a single integer.

### Details

This is a generic function with methods for vectors, data frames and arrays (including matrices).

The array method calculates for each element of the dimension specified by `MARGIN` if the remaining dimensions are identical to those for an earlier element (in row-major order). This would most commonly be used to find unique rows (the default) or columns (with `MARGIN = 2`).

### Value

An object of the same type of `x`. but if an element is equal to one with a smaller index, it is removed. Dimensions of arrays are not dropped.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[duplicated](#) which gives the indices of duplicated elements.

### Examples

```
unique(c(3:5, 11:8, 8 + 0:5))
length(unique(sample(100, 100, replace=TRUE)))
## approximately 100(1 - 1/e) = 63.21

unique(iris)
```

---

unlink

*Delete Files and Directories*

---

### Description

unlink deletes the file(s) or directories specified by `x`.

### Usage

```
unlink(x, recursive = FALSE)
```

### Arguments

<code>x</code>	a character vector with the names of the file(s) or directories to be deleted. Wildcards (normally <code>**</code> and <code>?</code> ) are allowed.
<code>recursive</code>	logical. Should directories be deleted recursively?

### Details

If `recursive = FALSE` directories are not deleted, not even empty ones.

[file.remove](#) can only remove files, but gives more detailed error information.

### Value

The return value of the corresponding system command, `rm -f`, normally 0 for success, 1 for failure. Not deleting a non-existent file is not a failure.

### Note

Prior to R version 1.2.0 the default on Unix was `recursive = TRUE`, and on Windows empty directories could be deleted.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[file.remove](#).

---

unlist	<i>Flatten Lists</i>
--------	----------------------

---

### Description

Given a list structure `x`, `unlist` simplifies it to produce a vector which contains all the atomic components which occur in `x`.

### Usage

```
unlist(x, recursive = TRUE, use.names = TRUE)
```

### Arguments

<code>x</code>	A list or vector.
<code>recursive</code>	logical. Should unlisting be applied to list components of <code>x</code> ?
<code>use.names</code>	logical. Should names be preserved?

### Details

`unlist` is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

If `recursive = FALSE`, the function will not recurse beyond the first level items in `x`.

`x` can be a vector, but then `unlist` does nothing useful, not even drop names.

By default, `unlist` tries to retain the naming information present in `x`. If `use.names = FALSE` all naming information is dropped.

Where possible the list elements are coerced to a common mode during the unlisting, and so the result often ends up as a character vector.

A list is a (generic) vector, and the simplified vector might still be a list (and might be unchanged). Non-vector elements of the list (for example language elements such as names, formulas and calls) are not coerced, and so a list containing one or more of these remains a list. (The effect of unlisting an `lm` fit is a list which has individual residuals as components,)

### Value

A vector of an appropriate mode to hold the list components.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[c](#), [as.list](#).

**Examples**

```

unlist(options())
unlist(options(), use.names=FALSE)

l.ex <- list(a = list(1:5, LETTERS[1:5]), b = "Z", c = NA)
unlist(l.ex, recursive = FALSE)
unlist(l.ex, recursive = TRUE)

l1 <- list(a="a", b=2, c=pi+2i)
unlist(l1) # a character vector
l2 <- list(a="a", b=as.name("b"), c=pi+2i)
unlist(l2) # remains a list

```

---

unname

*Remove 'names' or 'dimnames'*


---

**Description**

Remove the `names` or `dimnames` attribute of an R object.

**Usage**

```
unname(obj, force = FALSE)
```

**Arguments**

`obj` the R object which is wanted “nameless”.

`force` logical; if true, the `dimnames` are even removed from `data.frames`. *This argument is currently **experimental** and hence might change!*

**Value**

Object as `obj` but without `names` or `dimnames`.

**Examples**

```

## Answering a question on R-help (14 Oct 1999):
col3 <- 750+ 100* rt(1500, df = 3)
breaks <- factor(cut(col3,breaks=360+5*(0:155)))
z <- table(breaks)
z[1:5] # The names are larger than the data ...
barplot(unname(z), axes= FALSE)

```

UseMethod

*Class Methods***Description**

R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method despatch takes place based on the class of the first argument to the generic function or on the object supplied as an argument to `UseMethod` or `NextMethod`.

**Usage**

```
UseMethod(generic, object)
NextMethod(generic = NULL, object = NULL, ...)
```

**Arguments**

<code>generic</code>	a character string naming a function.
<code>object</code>	an object whose class will determine the method to be dispatched. Defaults to the first argument of the enclosing function.
<code>...</code>	further arguments to be passed to the method.

**Details**

An R “object” is a data object which has a `class` attribute. A class attribute is a character vector giving the names of the classes which the object “inherits” from. If the object does not have a class attribute, it has an implicit class, “matrix”, “array” or the result of `mode(x)`.

When a generic function `fun` is applied to an object with class attribute `c("first", "second")`, the system searches for a function called `fun.first` and, if it finds it, applied it to the object. If no such function is found a function called `fun.second` is tried. If no class name produces a suitable function, the function `fun.default` is used.

Function `methods` can be used to find out about the methods for a particular generic function or class.

Now for some obscure details that need to appear somewhere. These comments will be slightly different than those in Appendix A of the White S Book. `UseMethod` creates a “new” function call with arguments matched as they came in to the generic. Any local variables defined before the call to `UseMethod` are retained (unlike S). Any statements after the call to `UseMethod` will not be evaluated as `UseMethod` does not return. `UseMethod` can be called with more than two arguments: a warning will be given and additional arguments ignored. (They are not completely ignored in S.) If it is called with just one argument, the class of the first argument of the enclosing function is used as `object`: unlike S this is the actual argument passed and not the current value of the object of that name.

`NextMethod` invokes the next method (determined by the class). It does this by creating a special call frame for that method. The arguments will be the same in number, order and name as those to the current method but their values will be promises to evaluate their name in the current method and environment. Any arguments matched to `...` are handled specially. They are passed on as the

promise that was supplied as an argument to the current environment. (S does this differently!) If they have been evaluated in the current (or a previous environment) they remain evaluated.

`NextMethod` should not be called except in methods called by `UseMethod`. In particular it will not work inside anonymous calling functions (eg `get("print.ts")(AirPassengers)`).

Name spaces can register methods for generic functions. To support this, `UseMethod` and `NextMethod` search for methods in two places: first in the environment in which the generic function is called, and then in the registration data base for the environment in which the generic is defined (typically a name space). So methods for a generic function need to either be available in the environment of the call to the generic, or they must be registered. It does not matter whether they are visible in the environment in which the generic is defined.

### Note

This scheme is called *S3* (S version 3). For new projects, it is recommended to use the more flexible and robust *S4* scheme provided in the **methods** package.

The function `.isMethodsDispatchOn()` returns TRUE if the S4 method dispatch has been turned on in the evaluator. It is meant for R internal use only.

### References

Chambers, J. M. (1992) *Classes and methods: object-oriented programming in S*. Appendix A of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

### See Also

[methods](#), [class](#), [getS3method](#)

---

UserHooks

*Functions to Get and Set Hooks for Load, Attach, Detach and Unload*

---

### Description

These functions allow users to set actions to be taken before packages are attached/detached and namespaces are (un)loaded.

### Usage

```
getHook(hookName)
setHook(hookName, value, action = c("append", "prepend", "replace"))

packageEvent(pkgname,
             event = c("onLoad", "attach", "detach", "onUnload"))
```

**Arguments**

<code>hookName</code>	character string: the hook name
<code>pkgname</code>	character string: the package/namespace name. If versioned install has been used, <code>pkgname</code> should be the unversioned name of the package and any version information will be stripped.
<code>event</code>	character string: an event for the package
<code>value</code>	A function, or for <code>action="replace"</code> , <code>NULL</code> .
<code>action</code>	The action to be taken. The names can be abbreviated.

**Details**

`setHook` provides a general mechanism for users to register hooks, a list of functions to be called from system (or user) functions. The initial set of hooks is associated with events on packages/namespaces: these hooks are named via calls to `packageEvent`.

To remove a hook completely, call `setHook(hookName, NULL, "replace")`.

When an R package is attached by `library`, it can call initialization code via a function `.First.lib`, and when it is `detach`-ed it can tidy up via a function `.Last.lib`. Users can add their own initialization code via the hooks provided by these functions, functions which will be called as `funname(pkgname, pkgpath)` inside a `try` call. (The attach hook is called after `.First.lib` and the detach hook before `.Last.lib`.)

If a package has a namespace, there are two further actions, when the namespace is loaded (before being attached and after `.onLoad` is called) and when it is unloaded (after being detached and before `.onUnload`). Note that code in these hooks is run without the package being on the search path, so objects in the package need to be referred to using the double colon operator as in the example. (Unlike `.onLoad`, the user hook is run after the name space has been sealed.)

Hooks are normally run in the order shown by `getHook`, but the "detach" and "onUnload" hooks are run in reverse order so the default for package events is to add hooks 'inside' existing ones.

Note that when an R is finished, packages are not detached and namespaces are not unloaded, so the corresponding hooks will not be run.

The hooks are stored in the environment `.userHooksEnv` in the base package, with 'mangled' names.

**Value**

For `getHook` function, a list of functions (possible empty). For `setHook` function, no return value. For `packageEvent`, the derived hook name (a character string).

**See Also**

[library](#), [detach](#), [loadNamespace](#).

Other hooks may be added later: [plot.new](#) and [persp](#) already have them.

**Examples**

```
setHook(packageEvent("grDevices", "onLoad"),
        function(...) grDevices::ps.options(horizontal=FALSE))
```

vector

*Vectors***Description**

`vector` produces a vector of the given length and mode.

`as.vector`, a generic, attempts to coerce its argument into a vector of mode `mode` (the default is to coerce to whichever mode is most convenient). The attributes of `x` are removed.

`is.vector` returns TRUE if `x` is a vector (of mode logical, integer, real, complex, character, raw or list if not specified) and FALSE otherwise.

**Usage**

```
vector(mode = "logical", length = 0)
as.vector(x, mode = "any")
is.vector(x, mode = "any")
```

**Arguments**

<code>mode</code>	A character string giving an atomic mode, or "any".
<code>length</code>	A non-negative integer specifying the desired length.
<code>x</code>	An object.

**Details**

`is.vector` returns FALSE if `x` has any attributes except names. (This is incompatible with S.) On the other hand, `as.vector` removes *all* attributes including names.

Note that factors are *not* vectors; `is.vector` returns FALSE and `as.vector` converts to character mode.

**Value**

For `vector`, a vector of the given length and mode. Logical vector elements are initialized to FALSE, numeric vector elements to 0, character vector elements to "", raw vector elements to nul bytes and list elements to NULL.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`c`, `is.numeric`, `is.list`, etc.

**Examples**

```
df <- data.frame(x=1:3, y=5:7)
## Not run:
## Error:
  as.vector(data.frame(x=1:3, y=5:7), mode="numeric")
## End(Not run)

x <- c(a = 1, b = 2)
is.vector(x)
as.vector(x)
all.equal(x, as.vector(x)) ## FALSE

###-- All the following are TRUE:
is.list(df)
! is.vector(df)
! is.vector(df, mode="list")

is.vector(list(), mode="list")
is.vector(NULL, mode="NULL")
```

---

warning

*Warning Messages*

---

**Description**

Generates a warning message that corresponds to its argument(s) and (optionally) the expression or function from which it was called.

**Usage**

```
warning(..., call. = TRUE)
suppressWarnings(expr)
```

**Arguments**

<code>...</code>	character vectors (which are pasted together with no separator), a condition object, or NULL.
<code>call.</code>	logical, indicating if the call should become part of the warning message.
<code>expr</code>	expression to evaluate.

**Details**

The result *depends* on the value of `options("warn")` and on handlers established in the executing code.

`warning` signals a warning condition by (effectively) calling `signalCondition`. If there are no handlers or if all handlers return, then the value of `warn` is used to determine the appropriate action. If `warn` is negative warnings are ignored; if it is zero they are stored and printed after the top-level function has completed; if it is one they are printed as they occur and if it is 2 (or larger) warnings are turned into errors.

If `warn` is zero (the default), a top-level variable `last.warning` is created. It contains the warnings which can be printed via a call to `warnings`.

Warnings will be truncated to `getOption("warning.length")` characters, default 1000.

While the warning is being processed, a `muffleWarning` restart is available. If this restart is invoked with `invokeRestart`, then `warning` returns immediately.

`suppressWarnings` evaluates its expression in a context that ignores all warnings.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`stop` for fatal errors, `warnings`, and `options` with argument `warn=`.

**Examples**

```
testit <- function() warning("testit")
testit() ## shows call
testit <- function() warning("problem in testit", call. = FALSE)
testit() ## no call
suppressWarnings(warning("testit"))
```

---

warnings

*Print Warning Messages*

---

**Description**

`warnings` prints the top-level variable `last.warning` in a pleasing form.

**Usage**

```
warnings(...)
```

**Arguments**

... arguments to be passed to `cat`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[warning](#).

**Examples**

```
ow <- options("warn")
for(w in -1:1) {
  options(warn = w); cat("\n warn =",w,"\n")
  for(i in 1:3) { cat(i,"..\n"); m <- matrix(1:7, 3,4) }
}
warnings()
options(ow) # reset
```

---

weekdays

---

*Extract Parts of a POSIXt or Date Object*


---

**Description**

Extract the weekday, month or quarter, or the Julian time (days since some origin). These are generic functions: the methods for the internal date-time classes are documented here.

**Usage**

```
weekdays(x, abbreviate)
## S3 method for class 'POSIXt':
weekdays(x, abbreviate = FALSE)
## S3 method for class 'Date':
weekdays(x, abbreviate = FALSE)

months(x, abbreviate)
## S3 method for class 'POSIXt':
months(x, abbreviate = FALSE)
## S3 method for class 'Date':
months(x, abbreviate = FALSE)

quarters(x, abbreviate)
## S3 method for class 'POSIXt':
quarters(x, ...)
## S3 method for class 'Date':
```

```
quarters(x, ...)

julian(x, ...)
## S3 method for class 'POSIXt':
julian(x, origin = as.POSIXct("1970-01-01", tz="GMT"), ...)
## S3 method for class 'Date':
julian(x, origin = as.Date("1970-01-01"), ...)
```

### Arguments

`x` an object inheriting from class "POSIXt" or "Date".

`abbreviate` logical. Should the names be abbreviated?

`origin` an length-one object inheriting from class "POSIXt" or "Date".

`...` arguments for other methods.

### Value

`weekdays` and `months` return a character vector of names in the locale in use.

`quarters` returns a character vector of "Q1" to "Q4".

`julian` returns the number of days (possibly fractional) since the origin, with the origin as a "origin" attribute.

### Note

Other components such as the day of the month or the year are very easy to compute: just use `as.POSIXlt` and extract the relevant component.

### See Also

[DateTimeClasses](#), [Date](#)

### Examples

```
weekdays(.leap.seconds)
months(.leap.seconds)
quarters(.leap.seconds)
```

---

which

*Which indices are TRUE?*

---

### Description

Give the TRUE indices of a logical object, allowing for array indices.

### Usage

```
which(x, arr.ind = FALSE)
```

**Arguments**

`x` a **logical** vector or array. **NA**s are allowed and omitted (treated as if **FALSE**).

`arr.ind` logical; should **array indices** be returned when `x` is an array?

**Value**

If `arr.ind == FALSE` (the default), an integer vector with length equal to `sum(x)`, i.e., to the number of **TRUE**s in `x`; Basically, the result is `(1:length(x))[x]`.

If `arr.ind == TRUE` and `x` is an **array** (has a `dim` attribute), the result is a matrix whose rows each are the indices of one element of `x`; see Examples below.

**Author(s)**

Werner Stahel and Peter Holzer ([holzer@stat.math.ethz.ch](mailto:holzer@stat.math.ethz.ch)), for the array case.

**See Also**

[Logic](#), [which.min](#) for the index of the minimum or maximum, and [match](#) for the first index of an element in a vector, i.e., for a scalar `a`, `match(a, x)` is equivalent to `min(which(x == a))` but much more efficient.

**Examples**

```
which(LETTERS == "R")
which(l1 <- c(TRUE,FALSE,TRUE,NA,FALSE,FALSE,TRUE))#> 1 3 7
names(l1) <- letters[seq(l1)]
which(l1)
which((1:12)%%2 == 0) # which are even?
which(1:10 > 3, arr.ind=TRUE)

( m <- matrix(1:12,3,4) )
which(m %% 3 == 0)
which(m %% 3 == 0, arr.ind=TRUE)
rownames(m) <- paste("Case",1:3, sep="_")
which(m %% 5 == 0, arr.ind=TRUE)

dim(m) <- c(2,2,3); m
which(m %% 3 == 0, arr.ind=FALSE)
which(m %% 3 == 0, arr.ind=TRUE)

vm <- c(m)
dim(vm) <- length(vm) #-- funny thing with length(dim(...)) == 1
which(vm %% 3 == 0, arr.ind=TRUE)
```

---

`which.min`*Where is the Min() or Max() ?*

---

**Description**

Determines the location, i.e., index of the (first) minimum or maximum of a numeric vector.

**Usage**

```
which.min(x)
which.max(x)
```

**Arguments**

`x` numeric vector, whose `min` or `max` is searched (NAs are allowed).

**Value**

an `integer` of length 1 or 0 (iff `x` has no non-NAs) , giving the index of the *first* minimum or maximum respectively of `x`.

If this extremum is unique (or empty), the result is the same (but more efficient) as `which(x == min(x))` or `which(x == max(x))` respectively.

**Author(s)**

Martin Maechler

**See Also**

`which`, `max.col`, `max`, etc.

`which.is.max` in package **nnet** differs in breaking ties at random (and having a “fuzz” in the definition of ties).

**Examples**

```
x <- c(1:4,0:5,11)
which.min(x)
which.max(x)

## it *does* work with NA's present:
presidents[1:30]
range(presidents, na.rm = TRUE)
which.min(presidents)# 28
which.max(presidents)# 2
```

---

with

*Evaluate an Expression in a Data Environment*


---

### Description

Evaluate an R expression in an environment constructed from data.

### Usage

```
with(data, expr, ...)
```

### Arguments

<code>data</code>	data to use for constructing an environment. For the default method this may be an environment, a list, a data frame, or an integer as in <code>sys.call</code> .
<code>expr</code>	expression to evaluate.
<code>...</code>	arguments to be passed to future methods.

### Details

`with` is a generic function that evaluates `expr` in a local environment constructed from `data`. The environment has the caller's environment as its parent. This is useful for simplifying calls to modeling functions.

Note that assignments within `expr` take place in the constructed environment and not in the user's workspace.

### See Also

[evalq](#), [attach](#).

### Examples

```
require(stats); require(graphics)
#examples from glm:
## Not run:
library(MASS)
with(anorexia, {
  anorex.l <- glm(Postwt ~ Prewt + Treat + offset(Prewt),
                 family = gaussian)
  summary(anorex.l)
})
## End(Not run)

with(data.frame(u = c(5,10,15,20,30,40,60,80,100),
               lot1 = c(118,58,42,35,27,25,21,19,18),
               lot2 = c(69,35,26,21,18,16,13,12,12)),
     list(summary(glm(lot1 ~ log(u), family = Gamma)),
          summary(glm(lot2 ~ log(u), family = Gamma))))
```

```

# example from boxplot:
with(ToothGrowth, {
  boxplot(len ~ dose, boxwex = 0.25, at = 1:3 - 0.2,
          subset = (supp == "VC"), col = "yellow",
          main = "Guinea Pigs' Tooth Growth",
          xlab = "Vitamin C dose mg",
          ylab = "tooth length", ylim = c(0,35))
  boxplot(len ~ dose, add = TRUE, boxwex = 0.25, at = 1:3 + 0.2,
          subset = supp == "OJ", col = "orange")
  legend(2, 9, c("Ascorbic acid", "Orange juice"),
        fill = c("yellow", "orange"))
})

# alternate form that avoids subset argument:
with(subset(ToothGrowth, supp == "VC"),
     boxplot(len ~ dose, boxwex = 0.25, at = 1:3 - 0.2,
             col = "yellow", main = "Guinea Pigs' Tooth Growth",
             xlab = "Vitamin C dose mg",
             ylab = "tooth length", ylim = c(0,35)))
with(subset(ToothGrowth, supp == "OJ"),
     boxplot(len ~ dose, add = TRUE, boxwex = 0.25, at = 1:3 + 0.2,
             col = "orange"))
legend(2, 9, c("Ascorbic acid", "Orange juice"),
      fill = c("yellow", "orange"))

```

---

write

*Write Data to a File*


---

## Description

The data (usually a matrix) *x* are written to file *file*. If *x* is a two-dimensional matrix you need to transpose it to get the columns in *file* the same as those in the internal representation.

## Usage

```

write(x, file = "data",
      ncolumns = if(is.character(x)) 1 else 5,
      append = FALSE)

```

## Arguments

<i>x</i>	the data to be written out.
<i>file</i>	A connection, or a character string naming the file to write to. If "", print to the standard output connection. If it is "  cmd", the output is piped to the command given by 'cmd'.
<i>ncolumns</i>	the number of columns to write the data in.
<i>append</i>	if TRUE the data <i>x</i> is appended to file <i>file</i> .

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[save](#) for writing any R objects, [write.table](#) for data frames, and [scan](#) for reading data.

## Examples

```
# create a 2 by 5 matrix
x <- matrix(1:10,ncol=5)

# the file data contains x, two rows, five cols
# 1 3 5 6 9 will form the first row
write(t(x))

# the file data now contains the data in x,
# two rows, five cols but the first row is 1 2 3 4 5
write(x)
unlink("data") # tidy up
```

---

write.table	<i>Data Output</i>
-------------	--------------------

---

## Description

`write.table` prints its required argument `x` (after converting it to a data frame if it is not one already) to file. The entries in each line (row) are separated by the value of `sep`.

## Usage

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
           eol = "\n", na = "NA", dec = ".", row.names = TRUE,
           col.names = TRUE, qmethod = c("escape", "double"))
```

## Arguments

<code>x</code>	the object to be written, typically a data frame. If not, it is attempted to coerce <code>x</code> to a data frame.
<code>file</code>	either a character string naming a file or a connection. "" indicates output to the console.
<code>append</code>	logical. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.
<code>quote</code>	a logical value or a numeric vector. If TRUE, any character or factor columns will be surrounded by double quotes. If a numeric vector, its elements are taken as the indices of the columns to quote. In both cases, row and column names are quoted if they are written. If FALSE, nothing is quoted.

<code>sep</code>	the field separator string. Values within each row of <code>x</code> are separated by this string.
<code>eol</code>	the character(s) to print at the end of each line (row).
<code>na</code>	the string to use for missing values in the data.
<code>dec</code>	the string to use for decimal points.
<code>row.names</code>	either a logical value indicating whether the row names of <code>x</code> are to be written along with <code>x</code> , or a character vector of row names to be written.
<code>col.names</code>	either a logical value indicating whether the column names of <code>x</code> are to be written along with <code>x</code> , or a character vector of column names to be written.
<code>qmethod</code>	a character string specifying how to deal with embedded double quote characters when quoting strings. Must be one of "escape" (default), in which case the quote character is escaped in C style by a backslash, or "double", in which case it is doubled. You can specify just the initial letter.

### Details

Normally there is no column name for a column of row names. If `col.names=NA` a blank column name is added. This can be used to write CSV files for input to spreadsheets.

`write.table` can be slow for data frames with large numbers (hundreds or more) of columns: this is inevitable as each column could be of a different class and so must be handled separately. Function `write.matrix` in package **MASS** may be much more efficient if `x` is a matrix or can be represented in a numeric matrix.

If the table has no columns the rownames will be written if `row.names=TRUE`, and vice versa.

### See Also

The “R Data Import/Export” manual.

`read.table`, `write`.

`write.matrix` in package **MASS**.

### Examples

```
## Not run:
## To write a CSV file for input to Excel one might use
write.table(x, file = "foo.csv", sep = ",", col.names = NA)
## and to read this file back into R one needs
read.table("file.csv", header = TRUE, sep = ",", row.names=1)
## End(Not run)
```

---

writeLines	<i>Write Lines to a Connection</i>
------------	------------------------------------

---

**Description**

Write text lines to a connection.

**Usage**

```
writeLines(text, con = stdout(), sep = "\n")
```

**Arguments**

text	A character vector
con	A connection object or a character string.
sep	character. A string to be written to the connection after each line of text.

**Details**

If the `con` is a character string, the functions call `file` to obtain an file connection which is opened for the duration of the function call.

If the connection is open it is written from its current position. If it is not open, it is opened for the duration of the call and then closed again.

Normally `writeLines` is used with a text connection, and the default separator is converted to the normal separator for that platform (LF on Unix/Linux, CRLF on Windows, CR on Classic MacOS). For more control, open a binary connection and specify the precise value you want written to the file in `sep`. For even more control, use `writeChar` on a binary connection.

**See Also**

[connections](#), [writeChar](#), [writeBin](#), [readLines](#), [cat](#)

---

zip.file.extract	<i>Extract File from a Zip Archive</i>
------------------	--

---

**Description**

This will extract the file named `file` from the zip archive, if possible, and write it in a temporary location.

**Usage**

```
zip.file.extract(file, zipname = "R.zip")
```

**Arguments**

`file`            A file name.  
`zipname`        The file name of a zip archive, including the ".zip" extension if required.

**Details**

The method used is selected by `options(unzip=)`. All platforms support an "internal" unzip: this is the default under Windows and the fall-back under Unix if no unzip program was found during configuration and `R_UNZIPCMD` is not set.

The file will be extracted if it is in the archive and any required unzip utility is available. It will probably be extracted to the directory given by `tempdir`, overwriting an existing file of that name.

**Value**

The name of the original or extracted file. Success is indicated by returning a different name.

**Note**

The "internal" method is very simple, and will not set file dates.

---

zpackages

*Listing of Packages*


---

**Description**

`.packages` returns information about package availability.

**Usage**

```
.packages(all.available = FALSE, lib.loc = NULL)
```

**Arguments**

`all.available`    logical; if TRUE return a character vector of all available packages in `lib.loc`.  
`lib.loc`            a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.

**Details**

`.packages()` returns the "base names" of the currently attached packages *invisibly* whereas `.packages(all.available = TRUE)` gives (visibly) *all* packages available in the library location path `lib.loc`.

**Value**

A character vector of package "base names", invisible unless `all.available = TRUE`.

**Author(s)**

R core; Guido Masarotto for the `all.available=TRUE` part of `.packages`.

**See Also**

[library](#), [.libPaths](#).

**Examples**

```
(.packages())           # maybe just "base"  
.packages(all = TRUE)  # return all available as character vector  
require(splines)       # the same  
(.packages())         # "splines", too  
detach("package:splines")
```



## Chapter 2

# The datasets package

---

`ability.cov`

*Ability and Intelligence Tests*

---

### Description

Six tests were given to 112 individuals. The covariance matrix is given in this object.

### Usage

```
ability.cov
```

### Details

The tests are described as

**general:** a non-verbal measure of general intelligence using Cattell's culture-fair test.

**picture:** a picture-completion test

**blocks:** block design

**maze:** mazes

**reading:** reading comprehension

**vocab:** vocabulary

Bartholomew gives both covariance and correlation matrices, but these are inconsistent. Neither are in the original paper.

### Source

Bartholomew, D. J. (1987) *Latent Variable Analysis and Factor Analysis*. Griffin.

Bartholomew, D. J. and Knott, M. (1990) *Latent Variable Analysis and Factor Analysis*. Second Edition, Arnold.

**References**

Smith, G. A. and Stanley G. (1983) Clocking *g*: relating intelligence and measures of timed performance. *Intelligence*, **7**, 353–368.

**Examples**

```
require(stats)
(ability.FA <- factanal(factors = 1, covmat=ability.cov))
update(ability.FA, factors=2)
update(ability.FA, factors=2, rotation="promax")
```

---

airmiles

*Passenger Miles on Commercial US Airlines, 1937–1960*

---

**Description**

The revenue passenger miles flown by commercial airlines in the United States for each year from 1937 to 1960.

**Usage**

airmiles

**Format**

A time series of 24 observations; yearly, 1937–1960.

**Source**

F.A.A. Statistical Handbook of Aviation.

**References**

Brown, R. G. (1963) *Smoothing, Forecasting and Prediction of Discrete Time Series*. Prentice-Hall.

**Examples**

```
plot(airmiles, main = "airmiles data",
     xlab = "Passenger-miles flown by U.S. commercial airlines", col = 4)
```

---

AirPassengers      *Monthly Airline Passenger Numbers 1949-1960*

---

### Description

The classic Box & Jenkins airline data. Monthly totals of international airline passengers, 1949 to 1960.

### Usage

```
AirPassengers
```

### Format

A monthly time series, in thousands.

### Source

Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1976) *Time Series Analysis, Forecasting and Control*. Third Edition. Holden-Day. Series G.

### Examples

```
## Not run:
## These are quite slow and so not run by example(AirPassengers)

## The classic 'airline model', by full ML
(fit <- arima(log10(AirPassengers), c(0, 1, 1),
              seasonal = list(order=c(0, 1, 1), period=12)))
update(fit, method = "CSS")
update(fit, x=window(log10(AirPassengers), start = 1954))
pred <- predict(fit, n.ahead = 24)
tl <- pred$pred - 1.96 * pred$se
tu <- pred$pred + 1.96 * pred$se
ts.plot(AirPassengers, 10^tl, 10^tu, log = "y", lty = c(1,2,2))

## full ML fit is the same if the series is reversed, CSS fit is not
ap0 <- rev(log10(AirPassengers))
attributes(ap0) <- attributes(AirPassengers)
arima(ap0, c(0, 1, 1), seasonal = list(order=c(0, 1, 1), period=12))
arima(ap0, c(0, 1, 1), seasonal = list(order=c(0, 1, 1), period=12),
      method = "CSS")

## Structural Time Series
ap <- log10(AirPassengers) - 2
(fit <- StructTS(ap, type= "BSM"))
par(mfrow=c(1,2))
plot(cbind(ap, fitted(fit)), plot.type = "single")
plot(cbind(ap, tsSmooth(fit)), plot.type = "single")
## End(Not run)
```

---

airquality                      *New York Air Quality Measurements*

---

### Description

Daily air quality measurements in New York, May to September 1973.

### Usage

```
airquality
```

### Format

A data frame with 154 observations on 6 variables.

[ ,1]	Ozone	numeric	Ozone (ppb)
[ ,2]	Solar.R	numeric	Solar R (lang)
[ ,3]	Wind	numeric	Wind (mph)
[ ,4]	Temp	numeric	Temperature (degrees F)
[ ,5]	Month	numeric	Month (1–12)
[ ,6]	Day	numeric	Day of month (1–31)

### Details

Daily readings of the following air quality values for May 1, 1973 (a Tuesday) to September 30, 1973.

- Ozone: Mean ozone in parts per billion from 1300 to 1500 hours at Roosevelt Island
- Solar.R: Solar radiation in Langleys in the frequency band 4000–7700 Angstroms from 0800 to 1200 hours at Central Park
- Wind: Average wind speed in miles per hour at 0700 and 1000 hours at LaGuardia Airport
- Temp: Maximum daily temperature in degrees Fahrenheit at La Guardia Airport.

### Source

The data were obtained from the New York State Department of Conservation (ozone data) and the National Weather Service (meteorological data).

### References

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

### Examples

```
pairs(airquality, panel = panel.smooth, main = "airquality data")
```

---

 anscombe

*Anscombe's Quartet of "Identical" Simple Linear Regressions*


---

### Description

Four  $x$ - $y$  datasets which have the same traditional statistical properties (mean, variance, correlation, regression line, etc.), yet are quite different.

### Usage

```
anscombe
```

### Format

A data frame with 11 observations on 8 variables.

```
x1 == x2 == x3  the integers 4:14, specially arranged
                x4  values 8 and 19
y1, y2, y3, y4  numbers in (3, 12.5) with mean 7.5 and sdev 2.03
```

### Source

Tufte, Edward R. (1989) *The Visual Display of Quantitative Information*, 13–14. Graphics Press.

### References

Anscombe, Francis J. (1973) Graphs in statistical analysis. *American Statistician*, **27**, 17–21.

### Examples

```
require(stats)
summary(anscombe)

##-- now some "magic" to do the 4 regressions in a loop:
ff <- y ~ x
for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y","x"), i, sep=""), as.name)
  ## or  ff[[2]] <- as.name(paste("y", i, sep=""))
  ##      ff[[3]] <- as.name(paste("x", i, sep=""))
  assign(paste("lm.",i,sep=""), lmi <- lm(ff, data= anscombe))
  print(anova(lmi))
}

## See how close they are (numerically!)
sapply(objects(pat="lm\\.[1-4]$", function(n) coef(get(n)))
lapply(objects(pat="lm\\.[1-4]$", function(n) summary(get(n))$coef)

## Now, do what you should have done in the first place: PLOTS
op <- par(mfrow=c(2,2), mar=.1+c(4,4,1,1), oma= c(0,0,2,0))
```

```

for(i in 1:4) {
  ff[2:3] <- lapply(paste(c("y", "x"), i, sep=""), as.name)
  plot(ff, data =anscombe, col="red", pch=21, bg = "orange", cex = 1.2,
        xlim=c(3,19), ylim=c(3,13))
  abline(get(paste("lm.", i, sep="")), col="blue")
}
mtext("Anscombe's 4 Regression data sets", outer = TRUE, cex=1.5)
par(op)

```

---

attenu

*The Joyner–Boore Attenuation Data*


---

### Description

This data gives peak accelerations measured at various observation stations for 23 earthquakes in California. The data have been used by various workers to estimate the attenuating affect of distance on ground acceleration.

### Usage

```
attenu
```

### Format

A data frame with 182 observations on 5 variables.

[,1]	event	numeric	Event Number
[,2]	mag	numeric	Moment Magnitude
[,3]	station	factor	Station Number
[,4]	dist	numeric	Station-hypocenter distance (km)
[,5]	accel	numeric	Peak acceleration (g)

### Source

Joyner, W.B., D.M. Boore and R.D. Porcella (1981). Peak horizontal acceleration and velocity from strong-motion records including records from the 1979 Imperial Valley, California earthquake. USGS Open File report 81-365. Menlo Park, Ca.

### References

- Boore, D. M. and Joyner, W.B.(1982) The empirical prediction of ground motion, *Bull. Seism. Soc. Am.*, **72**, S269–S268.
- Bolt, B. A. and Abrahamson, N. A. (1982) New attenuation relations for peak and expected accelerations of strong ground motion, *Bull. Seism. Soc. Am.*, **72**, 2307–2321.
- Bolt B. A. and Abrahamson, N. A. (1983) Reply to W. B. Joyner & D. M. Boore’s “Comments on: New attenuation relations for peak and expected accelerations for peak and expected accelerations of strong ground motion”, *Bull. Seism. Soc. Am.*, **73**, 1481–1483.

Brillinger, D. R. and Preisler, H. K. (1984) An exploratory analysis of the Joyner-Boore attenuation data, *Bull. Seism. Soc. Am.*, **74**, 1441–1449.

Brillinger, D. R. and Preisler, H. K. (1984) *Further analysis of the Joyner-Boore attenuation data*. Manuscript.

### Examples

```
## check the data class of the variables
sapply(attenu, data.class)
summary(attenu)
pairs(attenu, main = "attenu data")
coplot(accel ~ dist | as.factor(event), data = attenu, show = FALSE)
coplot(log(accel) ~ log(dist) | as.factor(event),
       data = attenu, panel = panel.smooth, show.given = FALSE)
```

---

attitude

*The Chatterjee–Price Attitude Data*

---

### Description

From a survey of the clerical employees of a large financial organization, the data are aggregated from the questionnaires of the approximately 35 employees for each of 30 (randomly selected) departments. The numbers give the percent proportion of favourable responses to seven questions in each department.

### Usage

```
attitude
```

### Format

A dataframe with 30 observations on 7 variables. The first column are the short names from the reference, the second one the variable names in the data frame:

Y	rating	numeric	Overall rating
X[1]	complaints	numeric	Handling of employee complaints
X[2]	privileges	numeric	Does not allow special privileges
X[3]	learning	numeric	Opportunity to learn
X[4]	raises	numeric	Raises based on performance
X[5]	critical	numeric	Too critical
X[6]	advancel	numeric	Advancement

### Source

Chatterjee, S. and Price, B. (1977) *Regression Analysis by Example*. New York: Wiley. (Section 3.7, p.68ff of 2nd ed.(1991).)

**Examples**

```
require(stats)
pairs(attitude, main = "attitude data")
summary(attitude)
summary(fm1 <- lm(rating ~ ., data = attitude))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
summary(fm2 <- lm(rating ~ complaints, data = attitude))
plot(fm2)
par(opar)
```

---

austres

*Quarterly Time Series of the Number of Australian Residents*


---

**Description**

Numbers (in thousands) of Australian residents measured quarterly from March 1971 to March 1994. The object is of class "ts".

**Usage**

```
austres
```

**Source**

P. J. Brockwell and R. A. Davis (1996) *Introduction to Time Series and Forecasting*. Springer

---

beavers

*Body Temperature Series of Two Beavers*


---

**Description**

Reynolds (1994) describes a small part of a study of the long-term temperature dynamics of beaver *Castor canadensis* in north-central Wisconsin. Body temperature was measured by telemetry every 10 minutes for four females, but data from a one period of less than a day for each of two animals is used there.

**Usage**

```
beaver1
beaver2
```

**Format**

The `beaver1` data frame has 114 rows and 4 columns on body temperature measurements at 10 minute intervals.

The `beaver2` data frame has 100 rows and 4 columns on body temperature measurements at 10 minute intervals.

The variables are as follows:

**day** Day of observation (in days since the beginning of 1990), December 12–13 (`beaver1`) and November 3–4 (`beaver2`).

**time** Time of observation, in the form 0330 for 3:30am

**temp** Measured body temperature in degrees Celsius.

**activ** Indicator of activity outside the retreat.

**Note**

The observation at 22:20 is missing in `beaver1`.

**Source**

P. S. Reynolds (1994) Time-series analyses of beaver body temperatures. Chapter 11 of Lange, N., Ryan, L., Billard, L., Brillinger, D., Conquest, L. and Greenhouse, J. eds (1994) *Case Studies in Biometry*. New York: John Wiley and Sons.

**Examples**

```
(yl <- range(beaver1$temp, beaver2$temp))

beaver.plot <- function(bdat, ...) {
  nam <- deparse(substitute(bdat))
  attach(bdat)
  # Hours since start of day:
  hours <- time %% 100 + 24*(day - day[1]) + (time %% 100)/60
  plot(hours, temp, type = "l", ...,
        main = paste(nam, "body temperature"))
  abline(h = 37.5, col = "gray", lty = 2)
  is.act <- activ == 1
  points(hours[is.act], temp[is.act], col = 2, cex = .8)
}
op <- par(mfrow = c(2,1), mar = c(3,3,4,2), mgp = .9* 2:0)
beaver.plot(beaver1, ylim = yl)
beaver.plot(beaver2, ylim = yl)
par(op)
```

---

`BJsales`*Sales Data with Leading Indicator*

---

**Description**

The sales time series `BJsales` and leading indicator `BJsales.lead` each contain 150 observations. The objects are of class `"ts"`.

**Usage**

```
BJsales
BJsales.lead
```

**Source**

The data are given in Box & Jenkins (1976). Obtained from the Time Series Data Library at <http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/>

**References**

G. E. P. Box and G. M. Jenkins (1976): *Time Series Analysis, Forecasting and Control*, Holden-Day, San Francisco, p. 537.

P. J. Brockwell and R. A. Davis (1991): *Time Series: Theory and Methods*, Second edition, Springer Verlag, NY, pp. 414.

---

`BOD`*Biochemical Oxygen Demand*

---

**Description**

The BOD data frame has 6 rows and 2 columns giving the biochemical oxygen demand versus time in an evaluation of water quality.

**Usage**

```
BOD
```

**Format**

This data frame contains the following columns:

**Time** A numeric vector giving the time of the measurement (days).

**demand** A numeric vector giving the biochemical oxygen demand (mg/l).

**Source**

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, Appendix A1.4.

Originally from Marske (1967), *Biochemical Oxygen Demand Data Interpretation Using Sum of Squares Surface* M.Sc. Thesis, University of Wisconsin – Madison.

**Examples**

```
require(stats)
# simplest form of fitting a first-order model to these data
fm1 <- nls(demand ~ A*(1-exp(-exp(lrc)*Time)), data = BOD,
  start = c(A = 20, lrc = log(.35)))
coef(fm1)
print(fm1)
# using the plinear algorithm
fm2 <- nls(demand ~ (1-exp(-exp(lrc)*Time)), data = BOD,
  start = c(lrc = log(.35)), algorithm = "plinear", trace = TRUE)
# using a self-starting model
fm3 <- nls(demand ~ SSasymptOrig(Time, A, lrc), data = BOD)
summary( fm3 )
```

---

cars

*Speed and Stopping Distances of Cars*


---

**Description**

The data give the speed of cars and the distances taken to stop. Note that the data were recorded in the 1920s.

**Usage**

```
cars
```

**Format**

A data frame with 50 observations on 2 variables.

[,1]	speed	numeric	Speed (mph)
[,2]	dist	numeric	Stopping distance (ft)

**Source**

Ezekiel, M. (1930) *Methods of Correlation Analysis*. Wiley.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

**Examples**

```

require(stats)
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1)
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
title(main = "cars data")
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1, log = "xy")
title(main = "cars data (logarithmic scales)")
lines(lowess(cars$speed, cars$dist, f = 2/3, iter = 3), col = "red")
summary(fml <- lm(log(dist) ~ log(speed), data = cars))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
           mar = c(4.1, 4.1, 2.1, 1.1))

plot(fml)
par(opar)

## An example of polynomial regression
plot(cars, xlab = "Speed (mph)", ylab = "Stopping distance (ft)",
     las = 1, xlim = c(0, 25))
d <- seq(0, 25, len = 200)
for(degree in 1:4) {
  fm <- lm(dist ~ poly(speed, degree), data = cars)
  assign(paste("cars", degree, sep="."), fm)
  lines(d, predict(fm, data.frame(speed=d)), col = degree)
}
anova(cars.1, cars.2, cars.3, cars.4)

```

ChickWeight

*Weight versus age of chicks on different diets***Description**

The ChickWeight data frame has 578 rows and 4 columns from an experiment on the effect of diet on early growth of chicks.

**Usage**

```
ChickWeight
```

**Format**

This data frame contains the following columns:

**weight** a numeric vector giving the body weight of the chick (gm).

**Time** a numeric vector giving the number of days since birth when the measurement was made.

**Chick** an ordered factor with levels 18 < ... < 48 giving a unique identifier for the chick. The ordering of the levels groups chicks on the same diet together and orders them according to their final weight (lightest to heaviest) within diet.

**Diet** a factor with levels 1, ..., 4 indicating which experimental diet the chick received.

**Details**

The body weights of the chicks were measured at birth and every second day thereafter until day 20. They were also measured on day 21. There were four groups on chicks on different protein diets.

**Source**

Crowder, M. and Hand, D. (1990), *Analysis of Repeated Measures*, Chapman and Hall (example 5.3)

Hand, D. and Crowder, M. (1996), *Practical Longitudinal Data Analysis*, Chapman and Hall (table A.2)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

**Examples**

```
require(stats)
coplot(weight ~ Time | Chick, data = ChickWeight,
       type = "b", show = FALSE)
## fit a representative chick
fml <- nls(weight ~ SSlogis( Time, Asym, xmid, scal ),
          data = ChickWeight, subset = Chick == 1)
summary( fml )
```

---

chickwts

*Chicken Weights by Feed Type*


---

**Description**

An experiment was conducted to measure and compare the effectiveness of various feed supplements on the growth rate of chickens.

**Usage**

```
chickwts
```

**Format**

A data frame with 71 observations on 2 variables.

**weight** a numeric variable giving the chick weight.

**feed** a factor giving the feed type.

**Details**

Newly hatched chicks were randomly allocated into six groups, and each group was given a different feed supplement. Their weights in grams after six weeks are given along with feed types.

**Source**

Anonymous (1948) *Biometrika*, **35**, 214.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
require(stats)
boxplot(weight ~ feed, data = chickwts, col = "lightgray",
        varwidth = TRUE, notch = TRUE, main = "chickwt data",
        ylab = "Weight at six weeks (gm)")
anova(fm1 <- lm(weight ~ feed, data = chickwts))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
           mar = c(4.1, 4.1, 2.1, 1.1))
plot(fm1)
par(opar)
```

---

CO2

*Carbon Dioxide uptake in grass plants*


---

**Description**

The CO2 data frame has 84 rows and 5 columns of data from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*.

**Usage**

CO2

**Format**

This data frame contains the following columns:

**Plant** an ordered factor with levels Qn1 < Qn2 < Qn3 < ... < Mc1 giving a unique identifier for each plant.

**Type** a factor with levels Quebec Mississippi giving the origin of the plant

**Treatment** a factor with levels nonchilled chilled

**conc** a numeric vector of ambient carbon dioxide concentrations (mL/L).

**uptake** a numeric vector of carbon dioxide uptake rates ( $\mu\text{mol}/m^2 \text{ sec}$ ).

**Details**

The  $CO_2$  uptake of six plants from Quebec and six plants from Mississippi was measured at several levels of ambient  $CO_2$  concentration. Half the plants of each type were chilled overnight before the experiment was conducted.

**Source**

Potvin, C., Lechowicz, M. J. and Tardif, S. (1990) “The statistical analysis of ecophysiological response curves obtained from experiments involving repeated measures”, *Ecology*, **71**, 1389–1400.

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

**Examples**

```
require(stats)
coplot(uptake ~ conc | Plant, data = CO2, show = FALSE, type = "b")
## fit the data for the first plant
fml <- nls(uptake ~ SSasymp(conc, Asym, lrc, c0),
  data = CO2, subset = Plant == 'Qn1')
summary(fml)
## fit each plant separately
fmlist <- list()
for (pp in levels(CO2$Plant)) {
  fmlist[[pp]] <- nls(uptake ~ SSasymp(conc, Asym, lrc, c0),
    data = CO2, subset = Plant == pp)
}
## check the coefficients by plant
sapply(fmlist, coef)
```

---

co2

---

*Mauna Loa Atmospheric CO2 Concentration*


---

**Description**

Atmospheric concentrations of CO<sub>2</sub> are expressed in parts per million (ppm) and reported in the preliminary 1997 SIO manometric mole fraction scale.

**Usage**

```
co2
```

**Format**

A time series of 468 observations; monthly from 1959 to 1997.

**Details**

The values for February, March and April of 1964 were missing and have been obtained by interpolating linearly between the values for January and May of 1964.

**Source**

Keeling, C. D. and Whorf, T. P., Scripps Institution of Oceanography (SIO), University of California, La Jolla, California USA 92093-0220.

<ftp://cdiac.esd.ornl.gov/pub/maunaloa-co2/maunaloa.co2>.

**References**

Cleveland, W. S. (1993) *Visualizing Data*. New Jersey: Summit Press.

**Examples**

```
plot(co2, ylab = expression("Atmospheric concentration of CO"[2]),  
     las = 1)  
title(main = "co2 data set")
```

---

discoveries

*Yearly Numbers of Important Discoveries*

---

**Description**

The numbers of “great” inventions and scientific discoveries in each year from 1860 to 1959.

**Usage**

```
discoveries
```

**Format**

A time series of 100 values.

**Source**

The World Almanac and Book of Facts, 1975 Edition, pages 315–318.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

**Examples**

```
plot(discoveries, ylab = "Number of important discoveries",  
     las = 1)  
title(main = "discoveries data set")
```

---

DNase

*Elisa assay of DNase*

---

### Description

The DNase data frame has 176 rows and 3 columns of data obtained during development of an ELISA assay for the recombinant protein DNase in rat serum.

### Usage

```
DNase
```

### Format

This data frame contains the following columns:

**Run** an ordered factor with levels 10 < ... < 3 indicating the assay run.

**conc** a numeric vector giving the known concentration of the protein.

**density** a numeric vector giving the measured optical density (dimensionless) in the assay. Duplicate optical density measurements were obtained.

### Source

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.2.4, p. 134)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

### Examples

```
require(stats)
coplot(density ~ conc | Run, data = DNase,
       show = FALSE, type = "b")
coplot(density ~ log(conc) | Run, data = DNase,
       show = FALSE, type = "b")
## fit a representative run
fm1 <- nls(density ~ SSlogis( log(conc), Asym, xmid, scal ),
          data = DNase, subset = Run == 1)
## compare with a four-parameter logistic
fm2 <- nls(density ~ SSfpl( log(conc), A, B, xmid, scal ),
          data = DNase, subset = Run == 1)
summary(fm2)
anova(fm1, fm2)
```

---

 esoph

*Smoking, Alcohol and (O)esophageal Cancer*


---

**Description**

Data from a case-control study of (o)esophageal cancer in Ile-et-Vilaine, France.

**Usage**

esoph

**Format**

A data frame with records for 88 age/alcohol/tobacco combinations.

[,1]	"agegp"	Age group	1 25–34 years 2 35–44 3 45–54 4 55–64 5 65–74 6 75+
[,2]	"alcgp"	Alcohol consumption	1 0–39 gm/day 2 40–79 3 80–119 4 120+
[,3]	"tobgp"	Tobacco consumption	1 0– 9 gm/day 2 10–19 3 20–29 4 30+
[,4]	"ncases"	Number of cases	
[,5]	"ncontrols"	Number of controls	

**Author(s)**

Thomas Lumley

**Source**

Breslow, N. E. and Day, N. E. (1980) *Statistical Methods in Cancer Research. 1: The Analysis of Case-Control Studies*. IARC Lyon / Oxford University Press.

**Examples**

```
require(stats)
require(graphics) # for mosaicplot
summary(esoph)
## effects of alcohol, tobacco and interaction, age-adjusted
modell <- glm(cbind(ncases, ncontrols) ~ agegp + tobgp * alcgp,
```

```

                data = esoph, family = binomial())
anova(model1)
## Try a linear effect of alcohol and tobacco
model2 <- glm(cbind(ncases, ncontrols) ~ agegp + unclass(tobgp)
              + unclass(alcgp),
              data = esoph, family = binomial())
summary(model2)
## Re-arrange data for a mosaic plot
ttt <- table(esoph$agegp, esoph$alcgp, esoph$tobgp)
ttt[ttt == 1] <- esoph$ncases
ttl <- table(esoph$agegp, esoph$alcgp, esoph$tobgp)
ttl[ttl == 1] <- esoph$ncontrols
tt <- array(c(ttt, ttl), c(dim(ttt),2),
            c(dimnames(ttt), list(c("Cancer", "control"))))
mosaicplot(tt, main = "esoph data set", color = TRUE)

```

---

euro

*Conversion Rates of Euro Currencies*


---

### Description

Conversion rates between the various Euro currencies.

### Usage

```
euro
euro.cross
```

### Format

euro is a named vector of length 11, euro.cross a matrix of size 11 by 11, with dimnames.

### Details

The data set euro contains the value of 1 Euro in all currencies participating in the European monetary union (Austrian Schilling ATS, Belgian Franc BEF, German Mark DEM, Spanish Peseta ESP, Finnish Markka FIM, French Franc FRF, Irish Punt IEP, Italian Lira ITL, Luxembourg Franc LUF, Dutch Guilder NLG and Portugese Escudo PTE). These conversion rates were fixed by the European Union on December 31, 1998. To convert old prices to Euro prices, divide by the respective rate and round to 2 digits.

The data set euro.cross contains conversion rates between the various Euro currencies, i.e., the result of `outer(1 / euro, euro)`.

### Examples

```

cbind(euro)

## These relations hold:
euro == signif(euro,6) # [6 digit precision in Euro's definition]

```

```

all(euro.cross == outer(1/euro, euro))

## Convert 20 Euro to Belgian Franc
20 * euro["BEF"]
## Convert 20 Austrian Schilling to Euro
20 / euro["ATS"]
## Convert 20 Spanish Pesetas to Italian Lira
20 * euro.cross["ESP", "ITL"]

require(graphics)
dotchart(euro,
         main = "euro data: 1 Euro in currency unit")
dotchart(1/euro,
         main = "euro data: 1 currency unit in Euros")
dotchart(log(euro, 10),
         main = "euro data: log10(1 Euro in currency unit)")

```

---

eurodist

*Distances Between European Cities*


---

### Description

The data give the road distances (in km) between 21 cities in Europe. The data are taken from a table in “The Cambridge Encyclopaedia”.

### Usage

```
eurodist
```

### Format

A `dist` object based on 21 objects. (You must have the **stats** package loaded to have the methods for this kind of object available).

### Source

Crystal, D. Ed. (1990) *The Cambridge Encyclopaedia*. Cambridge: Cambridge University Press,

---

EuStockMarkets

*Daily Closing Prices of Major European Stock Indices, 1991–1998*


---

### Description

Contains the daily closing prices of major European stock indices: Germany DAX (Ibis), Switzerland SMI, France CAC, and UK FTSE. The data are sampled in business time, i.e., weekends and holidays are omitted.

**Usage**

```
EuStockMarkets
```

**Format**

A multivariate time series with 1860 observations on 4 variables. The object is of class "mts".

**Source**

The data were kindly provided by Erste Bank AG, Vienna, Austria.

---

faithful	<i>Old Faithful Geyser Data</i>
----------	---------------------------------

---

**Description**

Waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park, Wyoming, USA.

**Usage**

```
faithful
```

**Format**

A data frame with 272 observations on 2 variables.

[,1]	eruptions	numeric	Eruption time in mins
[,2]	waiting	numeric	Waiting time to next eruption

**Details**

A closer look at `faithful$eruptions` reveals that these are heavily rounded times originally in seconds, where multiples of 5 are more frequent than expected under non-human measurement. For a “better” version of the eruptions times, see the example below.

There are many versions of this dataset around: Azzalini and Bowman (1990) use a more complete version.

**Source**

W. Härdle.

**References**

Härdle, W. (1991) *Smoothing Techniques with Implementation in S*. New York: Springer.

Azzalini, A. and Bowman, A. W. (1990). A look at some data on the Old Faithful geyser. *Applied Statistics* **39**, 357–365.

**See Also**

geyser in package **MASS** for the Azzalini-Bowman version.

**Examples**

```
f.tit <- "faithful data: Eruptions of Old Faithful"

ne60 <- round(e60 <- 60 * faithful$eruptions)
all.equal(e60, ne60) # relative diff. ~ 1/10000
table(zapsmall(abs(e60 - ne60))) # 0, 0.02 or 0.04
faithful$better.eruptions <- ne60 / 60
te <- table(ne60)
te[te >= 4] # (too) many multiples of 5 !
plot(names(te), te, type="h", main = f.tit, xlab = "Eruption time (sec)")

plot(faithful[, -3], main = f.tit,
     xlab = "Eruption time (min)",
     ylab = "Waiting time to next eruption (min)")
lines(lowess(faithful$eruptions, faithful$waiting, f = 2/3, iter = 3),
      col = "red")
```

---

 Formaldehyde

*Determination of Formaldehyde*


---

**Description**

These data are from a chemical experiment to prepare a standard curve for the determination of formaldehyde by the addition of chromotropic acid and concentrated sulphuric acid and the reading of the resulting purple color on a spectrophotometer.

**Usage**

Formaldehyde

**Format**

A data frame with 6 observations on 2 variables.

```
[,1] carb    numeric Carbohydrate (ml)
[,2] optden  numeric Optical Density
```

**Source**

Bennett, N. A. and N. L. Franklin (1954) *Statistical Analysis in Chemistry and the Chemical Industry*. New York: Wiley.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
require(stats)
plot(optden ~ carb, data = Formaldehyde,
     xlab = "Carbohydrate (ml)", ylab = "Optical Density",
     main = "Formaldehyde data", col = 4, las = 1)
abline(fm1 <- lm(optden ~ carb, data = Formaldehyde))
summary(fm1)
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(fm1)
par(opar)
```

---

freeny

*Freeny's Revenue Data*

---

**Description**

Freeny's data on quarterly revenue and explanatory variables.

**Usage**

```
freeny
freeny.x
freeny.y
```

**Format**

There are three 'freeny' data sets.

`freeny.y` is a time series with 39 observations on quarterly revenue from (1962,2Q) to (1971,4Q).

`freeny.x` is a matrix of explanatory variables. The columns are `freeny.y` lagged 1 quarter, price index, income level, and market potential.

Finally, `freeny` is a data frame with variables `y`, `lag.quarterly.revenue`, `price.index`, `income.level`, and `market.potential` obtained from the above two data objects.

**Source**

A. E. Freeny (1977) *A Portable Linear Regression Package with Test Programs*. Bell Laboratories memorandum.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```
summary(freeny)
pairs(freeny, main = "freeny data")
summary(fml <- lm(y ~ ., data = freeny))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))
plot(fml)
par(opar)
```

---

HairEyeColor

*Hair and Eye Color of Statistics Students*


---

**Description**

Distribution of hair and eye color and sex in 592 statistics students.

**Usage**

```
HairEyeColor
```

**Format**

A 3-dimensional array resulting from cross-tabulating 592 observations on 3 variables. The variables and their levels are as follows:

No	Name	Levels
1	Hair	Black, Brown, Red, Blond
2	Eye	Brown, Blue, Hazel, Green
3	Sex	Male, Female

**Details**

This data set is useful for illustrating various techniques for the analysis of contingency tables, such as the standard chi-squared test or, more generally, log-linear modelling, and graphical methods such as mosaic plots, sieve diagrams or association plots.

**References**

Snee, R. D. (1974), Graphical display of two-way contingency tables. *The American Statistician*, **28**, 9–12.

Friendly, M. (1992), Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, **17**, 190–200. <http://www.math.yorku.ca/SCS/sugi/sugi17-paper.html>

Friendly, M. (1992), Mosaic displays for loglinear models. *Proceedings of the Statistical Graphics Section*, American Statistical Association, pp. 61–68. <http://www.math.yorku.ca/SCS/Papers/asa92.html>

**See Also**

[chisq.test](#), [loglin](#), [mosaicplot](#)

**Examples**

```
require(graphics)
## Full mosaic
mosaicplot(HairEyeColor)
## Aggregate over sex:
x <- apply(HairEyeColor, c(1, 2), sum)
x
mosaicplot(x, main = "Relation between hair and eye color")
```

---

Harman23.cor

*Harman Example 2.3*

---

**Description**

A correlation matrix of eight physical measurements on 305 girls between ages seven and seventeen.

**Usage**

Harman23.cor

**Source**

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 2.3.

**Examples**

```
require(stats)
(Harman23.FA <- factanal(factors = 1, covmat = Harman23.cor))
for(factors in 2:4) print(update(Harman23.FA, factors = factors))
```

---

Harman74.cor

*Harman Example 7.4*

---

**Description**

A correlation matrix of 24 psychological tests given to 145 seventh and eight-grade children in a Chicago suburb by Holzinger and Swineford.

**Usage**

Harman74.cor

**Source**

Harman, H. H. (1976) *Modern Factor Analysis*, Third Edition Revised, University of Chicago Press, Table 7.4.

**Examples**

```
require(stats)
(Harman74.FA <- factanal(factors = 1, covmat = Harman74.cor))
for(factors in 2:5) print(update(Harman74.FA, factors = factors))
Harman74.FA <- factanal(factors = 5, covmat = Harman74.cor,
                       rotation="promax")
print(Harman74.FA$loadings, sort = TRUE)
```

---

Indometh

*Pharmacokinetics of Indomethicin*


---

**Description**

The `Indometh` data frame has 66 rows and 3 columns of data on the pharmacokinetics of indomethicin.

**Usage**

```
Indometh
```

**Format**

This data frame contains the following columns:

**Subject** an ordered factor with containing the subject codes. The ordering is according to increasing maximum response.

**time** a numeric vector of times at which blood samples were drawn (hr).

**conc** a numeric vector of plasma concentrations of indomethicin (mcg/ml).

**Details**

Each of the six subjects were given an intravenous injection of indomethicin.

**Source**

Kwan, Breault, Umbenhauer, McMahon and Duggan (1976), "Kinetics of Indomethicin absorption, elimination, and enterohepatic circulation in man", *Journal of Pharmacokinetics and Biopharmaceutics*, **4**, 255–280.

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.2.4, p. 134)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

**Examples**

```
require(stats)
fml <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2),
          data = Indometh, subset = Subject == 1)
summary(fml)
```

---

 infert

---

*Infertility after Spontaneous and Induced Abortion*


---

**Description**

This is a matched case-control study dating from before the availability of conditional logistic regression.

**Usage**

```
infert
```

**Format**

1. Education	0 = 0-5 years 1 = 6-11 years 2 = 12+ years
2. age	age in years of case
3. parity	count
4. number of prior induced abortions	0 = 0 1 = 1 2 = 2 or more
5. case status	1 = case 0 = control
6. number of prior spontaneous abortions	0 = 0 1 = 1 2 = 2 or more
7. matched set number	1-83
8. stratum number	1-63

**Note**

One case with two prior spontaneous abortions and two prior induced abortions is omitted.

**Source**

Trichopoulos et al. (1976) *Br. J. of Obst. and Gynaec.* **83**, 645–650.

**Examples**

```

require(stats)
modell1 <- glm(case ~ spontaneous+induced, data=infert,family=binomial())
summary(modell1)
## adjusted for other potential confounders:
summary(model2 <- glm(case ~ age+parity+education+spontaneous+induced,
                      data=infert,family=binomial()))
## Really should be analysed by conditional logistic regression
## which is in the survival package
if(require(survival)){
  model3 <- clogit(case~spontaneous+induced+strata(stratum),data=infert)
  summary(model3)
  detach()# survival (conflicts)
}

```

---

InsectSprays

*Effectiveness of Insect Sprays*


---

**Description**

The counts of insects in agricultural experimental units treated with different insecticides.

**Usage**

```
InsectSprays
```

**Format**

A data frame with 72 observations on 2 variables.

[,1]	count	numeric	Insect count
[,2]	spray	factor	The type of spray

**Source**

Beall, G., (1942) The Transformation of data from entomological field experiments, *Biometrika*, **29**, 243–262.

**References**

McNeil, D. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```

require(stats)
boxplot(count ~ spray, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE, col = "lightgray")

```

```
fm1 <- aov(count ~ spray, data = InsectSprays)
summary(fm1)
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(fm1)
fm2 <- aov(sqrt(count) ~ spray, data = InsectSprays)
summary(fm2)
plot(fm2)
par(opar)
```

---

iris

*Edgar Anderson's Iris Data*

---

### Description

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*.

### Usage

```
iris
iris3
```

### Format

`iris` is a data frame with 150 cases (rows) and 5 variables (columns) named `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species`.

`iris3` gives the same data arranged as a 3-dimensional array of size 50 by 4 by 3, as represented by S-PLUS. The first dimension gives the case number within the species subsample, the second the measurements with names `Sepal L.`, `Sepal W.`, `Petal L.`, and `Petal W.`, and the third the species.

### Source

Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7, Part II, 179–188.

The data were collected by Anderson, Edgar (1935). The irises of the Gaspé Peninsula, *Bulletin of the American Iris Society*, 59, 2–5.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (has `iris3` as `iris`.)

### See Also

[matplot](#) some examples of which use `iris`.

**Examples**

```
dni3 <- dimnames(iris3)
ii <- data.frame(matrix(aperm(iris3, c(1,3,2)), ncol=4,
                        dimnames = list(NULL, sub(" L.", ".Length",
                                                sub(" W.", ".Width", dni3[[2]]))),
                  Species = gl(3, 50,
                              lab=sub("S", "s", sub("V", "v", dni3[[3]]))))
all.equal(ii, iris) # TRUE
```

---

islands

*Areas of the World's Major Landmasses*


---

**Description**

The areas in thousands of square miles of the landmasses which exceed 10,000 square miles.

**Usage**

```
islands
```

**Format**

A named vector of length 48.

**Source**

The World Almanac and Book of Facts, 1975, page 406.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

**Examples**

```
require(graphics)
dotchart(log(islands, 10),
         main = "islands data: log10(area) (log10(sq. miles))")
dotchart(log(islands[order(islands)], 10),
         main = "islands data: log10(area) (log10(sq. miles))")
```

---

JohnsonJohnson	<i>Quarterly Earnings per Johnson &amp; Johnson Share</i>
----------------	---

---

**Description**

Quarterly earnings (dollars) per Johnson & Johnson share 1960–80.

**Usage**

JohnsonJohnson

**Format**

A quarterly time series

**Source**

Shumway, R. H. and Stoffer, D. S. (2000) *Time Series Analysis and its Applications*. Second Edition. Springer. Example 1.1.

**Examples**

```
require(stats)
JJ <- log10(JohnsonJohnson)
plot(JJ)
(fit <- StructTS(JJ, type="BSM"))
tsdiag(fit)
sm <- tsSmooth(fit)
plot(cbind(JJ, sm[, 1], sm[, 3]-0.5), plot.type = "single",
     col = c("black", "green", "blue"))
abline(h = -0.5, col = "grey60")

monthplot(fit)
```

---

LakeHuron	<i>Level of Lake Huron 1875–1972</i>
-----------	--------------------------------------

---

**Description**

Annual measurements of the level, in feet, of Lake Huron 1875–1972.

**Usage**

LakeHuron

**Format**

A time series of length 98.

**Source**

Brockwell, P. J. & Davis, R. A. (1991). *Time Series and Forecasting Methods*. Second edition. Springer, New York. Series A, page 555.

Brockwell, P. J. & Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 5.1 and 7.6.

---

1h *Luteinizing Hormone in Blood Samples*

---

**Description**

A regular time series giving the luteinizing hormone in blood samples at 10 mins intervals from a human female, 48 samples.

**Usage**

1h

**Source**

P.J. Diggle (1990) *Time Series: A Biostatistical Introduction*. Oxford, table A.1, series 3

---

LifeCycleSavings *Intercountry Life-Cycle Savings Data*

---

**Description**

Data on the savings ratio 1960–1970.

**Usage**

LifeCycleSavings

**Format**

A data frame with 50 observations on 5 variables.

[,1]	sr	numeric	aggregate personal savings
[,2]	pop15	numeric	% of population under 15
[,3]	pop75	numeric	% of population over 75
[,4]	dpi	numeric	real per-capita disposable income
[,5]	ddpi	numeric	% growth rate of dpi

## Details

Under the life-cycle savings hypothesis as developed by Franco Modigliani, the savings ratio (aggregate personal saving divided by disposable income) is explained by per-capita disposable income, the percentage rate of change in per-capita disposable income, and two demographic variables: the percentage of population less than 15 years old and the percentage of the population over 75 years old. The data are averaged over the decade 1960–1970 to remove the business cycle or other short-term fluctuations.

## Source

The data were obtained from Belsley, Kuh and Welsch (1980). They in turn obtained the data from Sterling (1977).

## References

Sterling, Arnie (1977) Unpublished BS Thesis. Massachusetts Institute of Technology.  
Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.

## Examples

```
require(stats)
pairs(LifeCycleSavings, panel = panel.smooth,
      main = "LifeCycleSavings data")
fml <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)
summary(fml)
```

---

Loblolly

*Growth of Loblolly pine trees*

---

## Description

The Loblolly data frame has 84 rows and 3 columns of records of the growth of Loblolly pine trees.

## Usage

```
Loblolly
```

## Format

This data frame contains the following columns:

**height** a numeric vector of tree heights (ft).

**age** a numeric vector of tree ages (yr).

**Seed** an ordered factor indicating the seed source for the tree. The ordering is according to increasing maximum height.

**Source**

Kung, F. H. (1986), "Fitting logistic growth curve with predetermined carrying capacity", *Proceedings of the Statistical Computing Section, American Statistical Association*, 340–343.

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

**Examples**

```
require(stats)
plot(height ~ age, data = Loblolly, subset = Seed == 329,
      xlab = "Tree age (yr)", las = 1,
      ylab = "Tree height (ft)",
      main = "Loblolly data and fitted curve (Seed 329 only)")
fml <- nls(height ~ SSasymp(age, Asym, R0, lrc),
          data = Loblolly, subset = Seed == 329)
summary(fml)
age <- seq(0, 30, len = 101)
lines(age, predict(fml, list(age = age)))
```

---

longley

*Longley's Economic Regression Data*


---

**Description**

A macroeconomic data set which provides a well-known example for a highly collinear regression.

**Usage**

```
longley
```

**Format**

A data frame with 7 economical variables, observed yearly from 1947 to 1962 ( $n = 16$ ).

**GNP.deflator:** GNP implicit price deflator (1954 = 100)

**GNP:** Gross National Product.

**Unemployed:** number of unemployed.

**Armed.Forces:** number of people in the armed forces.

**Population:** 'noninstitutionalized' population  $\geq 14$  years of age.

**Year:** the year (time).

**Employed:** number of people employed.

The regression `lm(Employed ~ .)` is known to be highly collinear.

**Source**

J. W. Longley (1967) An appraisal of least-squares programs from the point of view of the user. *Journal of the American Statistical Association*, **62**, 819–841.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## Examples

```
## give the data set in the form it is used in S-PLUS:
longley.x <- data.matrix(longley[, 1:6])
longley.y <- longley[, "Employed"]
pairs(longley, main = "longley data")
summary(fml <- lm(Employed ~ ., data = longley))
opar <- par(mfrow = c(2, 2), oma = c(0, 0, 1.1, 0),
            mar = c(4.1, 4.1, 2.1, 1.1))

plot(fml)
par(opar)
```

---

lynx

*Annual Canadian Lynx trappings 1821–1934*

---

## Description

Annual numbers of lynx trappings for 1821–1934 in Canada. Taken from Brockwell & Davis (1991), this appears to be the series considered by Campbell & Walker (1977).

## Usage

lynx

## Source

Brockwell, P. J. and Davis, R. A. (1991) *Time Series and Forecasting Methods*. Second edition. Springer. Series G (page 557).

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Campbell, M. J. and A. M. Walker (1977). A Survey of statistical work on the Mackenzie River series of annual Canadian lynx trappings for the years 1821–1934 and a new analysis. *Journal of the Royal Statistical Society series A*, **140**, 411–431.

---

`morley`*Michaelson-Morley Speed of Light Data*

---

**Description**

The classical data of Michaelson and Morley on the speed of light. The data consists of five experiments, each consisting of 20 consecutive ‘runs’. The response is the speed of light measurement, suitably coded.

**Usage**`morley`**Format**

A data frame contains the following components:

**Expt** The experiment number, from 1 to 5.

**Run** The run number within each experiment.

**Speed** Speed-of-light measurement.

**Details**

The data is here viewed as a randomized block experiment with ‘experiment’ and ‘run’ as the factors. ‘run’ may also be considered a quantitative variate to account for linear (or polynomial) changes in the measurement over the course of a single experiment.

**Source**

A. J. Weekes (1986) *A Genstat Primer*. London: Edward Arnold.

**Examples**

```
require(stats)
morley$Expt <- factor(morley$Expt)
morley$Run <- factor(morley$Run)
attach(morley)
plot(Expt, Speed, main = "Speed of Light Data", xlab = "Experiment No.")
fm <- aov(Speed ~ Run + Expt, data = morley)
summary(fm)
fm0 <- update(fm, . ~ . - Run)
anova(fm0, fm)
detach(morley)
```

mtcars

*Motor Trend Car Road Tests***Description**

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

**Usage**

```
mtcars
```

**Format**

A data frame with 32 observations on 11 variables.

[, 1]	mpg	Miles/(US) gallon
[, 2]	cyl	Number of cylinders
[, 3]	disp	Displacement (cu.in.)
[, 4]	hp	Gross horsepower
[, 5]	drat	Rear axle ratio
[, 6]	wt	Weight (lb/1000)
[, 7]	qsec	1/4 mile time
[, 8]	vs	V/S
[, 9]	am	Transmission (0 = automatic, 1 = manual)
[,10]	gear	Number of forward gears
[,11]	carb	Number of carburetors

**Source**

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391–411.

**Examples**

```
pairs(mtcars, main = "mtcars data")
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 1)
```

nhtemp

*Average Yearly Temperatures in New Haven***Description**

The mean annual temperature in degrees Fahrenheit in New Haven, Connecticut, from 1912 to 1971.

**Usage**

nhtemp

**Format**

A time series of 60 observations.

**Source**

Vaux, J. E. and Brinker, N. B. (1972) *Cycles*, **1972**, 117–121.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
plot(nhtemp, main = "nhtemp data",  
     ylab = "Mean annual temperature in New Haven, CT (deg. F)")
```

---

Nile

*Flow of the River Nile*

---

**Description**

Measurements of the annual flow of the river Nile at Ashwan 1871–1970.

**Usage**

Nile

**Format**

A time series of length 100.

**Source**

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press. <http://www.ssfpack.com/dkbook/>

**References**

Balke, N. S. (1993) Detecting level shifts in time series. *Journal of Business and Economic Statistics* **11**, 81–92.

Cobb, G. W. (1978) The problem of the Nile: conditional solution to a change-point problem. *Biometrika* **65**, 243–51.

**Examples**

```

require(stats)
par(mfrow = c(2,2))
plot(Nile)
acf(Nile)
pacf(Nile)
ar(Nile) # selects order 2
cpgram(ar(Nile)$resid)
par(mfrow = c(1,1))
arima(Nile, c(2, 0, 0))

## Now consider missing values, following Durbin & Koopman
NileNA <- Nile
NileNA[c(21:40, 61:80)] <- NA
arima(NileNA, c(2, 0, 0))
plot(NileNA)
pred <- predict(arima(window(NileNA, 1871, 1890), c(2,0,0)), n.ahead = 20)
lines(pred$pred, lty = 3, col = "red")
lines(pred$pred + 2*pred$se, lty=2, col="blue")
lines(pred$pred - 2*pred$se, lty=2, col="blue")
pred <- predict(arima(window(NileNA, 1871, 1930), c(2,0,0)), n.ahead = 20)
lines(pred$pred, lty = 3, col = "red")
lines(pred$pred + 2*pred$se, lty=2, col="blue")
lines(pred$pred - 2*pred$se, lty=2, col="blue")

## Structural time series models
par(mfrow = c(3, 1))
plot(Nile)
## local level model
(fit <- StructTS(Nile, type = "level"))
lines(fitted(fit), lty = 2)
lines(tsSmooth(fit), lty = 2, col = 4) # contempareneous smoothing
plot(residuals(fit)); abline(h = 0, lty = 3)
## local trend model
(fit2 <- StructTS(Nile, type = "trend")) ## constant trend fitted
pred <- predict(fit, n.ahead = 30)
## with 50% confidence interval
ts.plot(Nile, pred$pred, pred$pred + 0.67*pred$se, pred$pred -0.67*pred$se)

## Now consider missing values
plot(NileNA)
(fit3 <- StructTS(NileNA, type = "level"))
lines(fitted(fit3), lty = 2)
lines(tsSmooth(fit3), lty = 3)
plot(residuals(fit3)); abline(h = 0, lty = 3)

```

**Description**

A time series object containing average air temperatures at Nottingham Castle in degrees Fahrenheit for 20 years.

**Usage**

```
nottem
```

**Source**

Anderson, O. D. (1976) *Time Series Analysis and Forecasting: The Box-Jenkins approach*. Butterworths. Series R.

**Examples**

```
## Not run:
nott <- window(nottem, end=c(1936,12))
fit <- arima(nott,order=c(1,0,0), list(order=c(2,1,0), period=12))
nott.fore <- predict(fit, n.ahead=36)
ts.plot(nott, nott.fore$pred, nott.fore$pred+2*nott.fore$se,
        nott.fore$pred-2*nott.fore$se, gpars=list(col=c(1,1,4,4)))
## End(Not run)
```

---

Orange

*Growth of orange trees*

---

**Description**

The Orange data frame has 35 rows and 3 columns of records of the growth of orange trees.

**Usage**

```
Orange
```

**Format**

This data frame contains the following columns:

**Tree** an ordered factor indicating the tree on which the measurement is made. The ordering is according to increasing maximum diameter.

**age** a numeric vector giving the age of the tree (days since 1968/12/31)

**circumference** a numeric vector of trunk circumferences (mm). This is probably “circumference at breast height”, a standard measurement in forestry.

**Source**

Draper, N. R. and Smith, H. (1998), *Applied Regression Analysis (3rd ed)*, Wiley (exercise 24.N).

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer.

**Examples**

```
require(stats)
coplot(circumference ~ age | Tree, data = Orange, show = FALSE)
fml <- nls(circumference ~ SSlogis(age, Asym, xmid, scal),
          data = Orange, subset = Tree == 3)
plot(circumference ~ age, data = Orange, subset = Tree == 3,
     xlab = "Tree age (days since 1968/12/31)",
     ylab = "Tree circumference (mm)", las = 1,
     main = "Orange tree data and fitted model (Tree 3 only)")
age <- seq(0, 1600, len = 101)
lines(age, predict(fml, list(age = age)))
```

OrchardSprays

*Potency of Orchard Sprays***Description**

An experiment was conducted to assess the potency of various constituents of orchard sprays in repelling honeybees, using a Latin square design.

**Usage**

```
OrchardSprays
```

**Format**

A data frame with 64 observations on 4 variables.

[,1]	rowpos	numeric	Row of the design
[,2]	colpos	numeric	Column of the design
[,3]	treatment	factor	Treatment level
[,4]	decrease	numeric	Response

**Details**

Individual cells of dry comb were filled with measured amounts of lime sulphur emulsion in sucrose solution. Seven different concentrations of lime sulphur ranging from a concentration of 1/100 to 1/1,562,500 in successive factors of 1/5 were used as well as a solution containing no lime sulphur.

The responses for the different solutions were obtained by releasing 100 bees into the chamber for two hours, and then measuring the decrease in volume of the solutions in the various cells.

An  $8 \times 8$  Latin square design was used and the treatments were coded as follows:

A	highest level of lime sulphur
B	next highest level of lime sulphur
.	
.	
.	

G lowest level of lime sulphur  
 H no lime sulphur

### Source

Finney, D. J. (1947) *Probit Analysis*. Cambridge.

### References

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

### Examples

```
pairs(OrchardSprays, main = "OrchardSprays data")
```

---

PlantGrowth

*Results from an Experiment on Plant Growth*

---

### Description

Results from an experiment to compare yields (as measured by dried weight of plants) obtained under a control and two different treatment conditions.

### Usage

```
PlantGrowth
```

### Format

A data frame of 30 cases on 2 variables.

```
[, 1] weight numeric
[, 2] group  factor
```

The levels of group are 'ctrl', 'trt1', and 'trt2'.

### Source

Dobson, A. J. (1983) *An Introduction to Statistical Modelling*. London: Chapman and Hall.

### Examples

```
## One factor ANOVA example from Dobson's book, cf. Table 7.4:
require(stats)
boxplot(weight ~ group, data = PlantGrowth, main = "PlantGrowth data",
        ylab = "Dried weight of plants", col = "lightgray",
        notch = TRUE, varwidth = TRUE)
anova(lm(weight ~ group, data = PlantGrowth))
```

---

precip

*Annual Precipitation in US Cities*

---

**Description**

The average amount of precipitation (rainfall) in inches for each of 70 United States (and Puerto Rico) cities.

**Usage**

```
precip
```

**Format**

A named vector of length 70.

**Source**

Statistical Abstracts of the United States, 1975.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
require(graphics)
dotchart(precip[order(precip)], main = "precip data")
title(sub = "Average annual precipitation (in.)")
```

---

presidents

*Quarterly Approval Ratings of US Presidents*

---

**Description**

The (approximately) quarterly approval rating for the President of the United states from the first quarter of 1945 to the last quarter of 1974.

**Usage**

```
presidents
```

**Format**

A time series of 120 values.

**Details**

The data are actually a fudged version of the approval ratings. See McNeil's book for details.

**Source**

The Gallup Organisation.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
plot(presidents, las = 1, ylab = "Approval rating (%)",
     main = "presidents data")
```

---

pressure

*Vapor Pressure of Mercury as a Function of Temperature*

---

**Description**

Data on the relation between temperature in degrees Celsius and vapor pressure of mercury in millimeters (of mercury).

**Usage**

```
pressure
```

**Format**

A data frame with 19 observations on 2 variables.

[, 1]	temperature	numeric	temperature (deg C)
[, 2]	pressure	numeric	pressure (mm)

**Source**

Weast, R. C., ed. (1973) *Handbook of Chemistry and Physics*. CRC Press.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
plot(pressure, xlab = "Temperature (deg C)",
     ylab = "Pressure (mm of Hg)",
     main = "pressure data: Vapor Pressure of Mercury")
```

```
plot(pressure, xlab = "Temperature (deg C)", log = "y",
     ylab = "Pressure (mm of Hg)",
     main = "pressure data: Vapor Pressure of Mercury")
```

---

Puromycin

*Reaction velocity of an enzymatic reaction*

---

## Description

The Puromycin data frame has 23 rows and 3 columns of the reaction velocity versus substrate concentration in an enzymatic reaction involving untreated cells or cells treated with Puromycin.

## Usage

```
Puromycin
```

## Format

This data frame contains the following columns:

**conc** a numeric vector of substrate concentrations (ppm)

**rate** a numeric vector of instantaneous reaction rates (counts/min/min)

**state** a factor with levels treated untreated

## Details

Data on the “velocity” of an enzymatic reaction were obtained by Treloar (1974). The number of counts per minute of radioactive product from the reaction was measured as a function of substrate concentration in parts per million (ppm) and from these counts the initial rate, or “velocity,” of the reaction was calculated (counts/min/min). The experiment was conducted once with the enzyme treated with Puromycin, and once with the enzyme untreated.

## Source

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley, Appendix A1.3.

Treloar, M. A. (1974), *Effects of Puromycin on Galactosyltransferase in Golgi Membranes*, M.Sc. Thesis, U. of Toronto.

## Examples

```
plot(rate ~ conc, data = Puromycin, las = 1,
     xlab = "Substrate concentration (ppm)",
     ylab = "Reaction velocity (counts/min/min)",
     pch = as.integer(Puromycin$state),
     col = as.integer(Puromycin$state),
     main = "Puromycin data and fitted Michaelis-Menten curves")
## simplest form of fitting the Michaelis-Menten model to these data
```

```

fm1 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "treated",
           start = c(Vm = 200, K = 0.05), trace = TRUE)
fm2 <- nls(rate ~ Vm * conc/(K + conc), data = Puromycin,
           subset = state == "untreated",
           start = c(Vm = 160, K = 0.05), trace = TRUE)
summary(fm1)
summary(fm2)
## using partial linearity
fm3 <- nls(rate ~ conc/(K + conc), data = Puromycin,
           subset = state == "treated", start = c(K = 0.05),
           algorithm = "plinear", trace = TRUE)
## using a self-starting model
fm4 <- nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,
           subset = state == "treated")
summary(fm4)
## add fitted lines to the plot
conc <- seq(0, 1.2, len = 101)
lines(conc, predict(fm1, list(conc = conc)), lty = 1, col = 1)
lines(conc, predict(fm2, list(conc = conc)), lty = 2, col = 2)
legend(0.8, 120, levels(Puromycin$state),
       col = 1:2, lty = 1:2, pch = 1:2)

```

---

quakes

*Locations of Earthquakes off Fiji*


---

### Description

The data set give the locations of 1000 seismic events of  $MB > 4.0$ . The events occurred in a cube near Fiji since 1964.

### Usage

```
quakes
```

### Format

A data frame with 1000 observations on 5 variables.

[,1]	lat	numeric	Latitude of event
[,2]	long	numeric	Longitude
[,3]	depth	numeric	Depth (km)
[,4]	mag	numeric	Richter Magnitude
[,5]	stations	numeric	Number of stations reporting

### Details

There are two clear planes of seismic activity. One is a major plate junction; the other is the Tonga trench off New Zealand. These data constitute a subsample from a larger dataset of containing 5000

observations.

### Source

This is one of the Harvard PRIM-H project data sets. They in turn obtained it from Dr. John Woodhouse, Dept. of Geophysics, Harvard University.

### Examples

```
pairs(quakes, main = "Fiji Earthquakes, N = 1000", cex.main=1.2, pch=".")
```

---

randu

*Random Numbers from Congruential Generator RANDU*

---

### Description

400 triples of successive random numbers were taken from the VAX FORTRAN function RANDU running under VMS 1.5.

### Usage

```
randu
```

### Format

A data frame with 400 observations on 3 variables named  $x$ ,  $y$  and  $z$  which give the first, second and third random number in the triple.

### Details

In three dimensional displays it is evident that the triples fall on 15 parallel planes in 3-space. This can be shown theoretically to be true for all triples from the RANDU generator.

These particular 400 triples start 5 apart in the sequence, that is they are  $(U[5i+1], U[5i+2], U[5i+3])$ ,  $i=0, \dots, 399$ , and they are rounded to 6 decimal places.

Under VMS versions 2.0 and higher, this problem has been fixed.

### Source

David Donoho

**Examples**

```
## Not run:
## We could re-generate the dataset by the following R code
seed <- as.double(1)
RANDU <- function() {
  seed <-- ((2^16 + 3) * seed) %% (2^31)
  seed/(2^31)
}
for(i in 1:400) {
  U <- c(RANDU(), RANDU(), RANDU(), RANDU(), RANDU())
  print(round(U[1:3], 6))
}
## End(Not run)
```

---

rivers

*Lengths of Major North American Rivers*

---

**Description**

This data set gives the lengths (in miles) of 141 “major” rivers in North America, as compiled by the US Geological Survey.

**Usage**

```
rivers
```

**Format**

A vector containing 141 observations.

**Source**

World Almanac and Book of Facts, 1975, page 406.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

---

 rock
 

---



---

*Measurements on Petroleum Rock Samples*


---

**Description**

Measurements on 48 rock samples from a petroleum reservoir.

**Usage**

rock

**Format**

A data frame with 48 rows and 4 numeric columns.

[,1]	area	area of pores space, in pixels out of 256 by 256
[,2]	peri	perimeter in pixels
[,3]	shape	perimeter/sqrt(area)
[,4]	perm	permeability in milli-Darcies

**Details**

Twelve core samples from petroleum reservoirs were sampled by 4 cross-sections. Each core sample was measured for permeability, and each cross-section has total area of pores, total perimeter of pores, and shape.

**Source**

Data from BP Research, image analysis by Ronit Katz, U. Oxford.

---

 sleep
 

---



---

*Student's Sleep Data*


---

**Description**

Data which show the effect of two soporific drugs (increase in hours of sleep) on groups consisting of 10 patients each.

**Usage**

sleep

**Format**

A data frame with 20 observations on 2 variables.

[, 1]	extra	numeric	increase in hours of sleep
[, 2]	group	factor	patient group

**Source**

Student (1908) The probable error of the mean. *Biometrika*, **6**, 20.

**References**

Scheffé, Henry (1959) *The Analysis of Variance*. New York, NY: Wiley.

**Examples**

```
require(stats)
## ANOVA
anova(lm(extra ~ group, data = sleep))
```

---

stackloss

*Brownlee's Stack Loss Plant Data*

---

**Description**

Operational data of a plant for the oxidation of ammonia to nitric acid.

**Usage**

```
stackloss

stack.x
stack.loss
```

**Format**

stackloss is a data frame with 21 observations on 4 variables.

[,1]	Air Flow	Flow of cooling air
[,2]	Water Temp	Cooling Water Inlet Temperature
[,3]	Acid Conc.	Concentration of acid [per 1000, minus 500]
[,4]	stack.loss	Stack loss

For compatibility with S-PLUS, the data sets `stack.x`, a matrix with the first three (independent) variables of the data frame, and `stack.loss`, the numeric vector giving the fourth (dependent) variable, are provided as well.

**Details**

“Obtained from 21 days of operation of a plant for the oxidation of ammonia (NH<sub>3</sub>) to nitric acid (HNO<sub>3</sub>). The nitric oxides produced are absorbed in a countercurrent absorption tower”. (Brownlee,

cited by Dodge, slightly reformatted by MM.)

`Air Flow` represents the rate of operation of the plant. `Water Temp` is the temperature of cooling water circulated through coils in the absorption tower. `Acid Conc .` is the concentration of the acid circulating, minus 50, times 10: that is, 89 corresponds to 58.9 per cent acid. `stack.loss` (the dependent variable) is 10 times the percentage of the ingoing ammonia to the plant that escapes from the absorption column unabsorbed; that is, an (inverse) measure of the over-all efficiency of the plant.

### Source

Brownlee, K. A. (1960, 2nd ed. 1965) *Statistical Theory and Methodology in Science and Engineering*. New York: Wiley. pp. 491–500.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Dodge, Y. (1996) The guinea pig of multiple regression. In: *Robust Statistics, Data Analysis, and Computer Intensive Methods; In Honor of Peter Huber's 60th Birthday, 1996, Lecture Notes in Statistics 109*, Springer-Verlag, New York.

### Examples

```
summary(lm.stack <- lm(stack.loss ~ stack.x))
```

---

state

*US State Facts and Figures*

---

### Description

Data sets related to the 50 states of the United States of America.

### Usage

```
state.abb  
state.area  
state.center  
state.division  
state.name  
state.region  
state.x77
```

## Details

R currently contains the following “state” data sets. Note that all data are arranged according to alphabetical order of the state names.

**state.abb:** character vector of 2-letter abbreviations for the state names.

**state.area:** numeric vector of state areas (in square miles).

**state.center:** list with components named *x* and *y* giving the approximate geographic center of each state in negative longitude and latitude. Alaska and Hawaii are placed just off the West Coast.

**state.division:** factor giving state divisions (New England, Middle Atlantic, South Atlantic, East South Central, West South Central, East North Central, West North Central, Mountain, and Pacific).

**state.name:** character vector giving the full state names.

**state.region:** factor giving the region (Northeast, South, North Central, West) that each state belongs to.

**state.x77:** matrix with 50 rows and 8 columns giving the following statistics in the respective columns.

**Population:** population estimate as of July 1, 1975

**Income:** per capita income (1974)

**Illiteracy:** illiteracy (1970, percent of population)

**Life Exp:** life expectancy in years (1969–71)

**Murder:** murder and non-negligent manslaughter rate per 100,000 population (1976)

**HS Grad:** percent high-school graduates (1970)

**Frost:** mean number of days with minimum temperature below freezing (1931–1960) in capital or large city

**Area:** land area in square miles

## Source

U.S. Department of Commerce, Bureau of the Census (1977) *Statistical Abstract of the United States*.

U.S. Department of Commerce, Bureau of the Census (1977) *County and City Data Book*.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

---

sunspot.month      *Monthly Sunspot Data, 1749–1997*

---

### Description

Monthly numbers of sunspots.

### Usage

```
sunspot.month
```

### Format

The univariate time series `sunspot.year` and `sunspot.month` contain 289 and 2988 observations, respectively. The objects are of class "ts".

### Note

Prior to R 2.0.0 `sunspot.month` and `sunspot.year` were copied to the user's workspace (or elsewhere) by `data(sunspot)`. However, as package **lattice** has a dataset `sunspot` which can be retrieved by the same command, that usage has been removed.

### Source

World Data Center-C1 For Sunspot Index Royal Observatory of Belgium, Av. Circulaire, 3, B-1180 BRUSSELS [http://www.oma.be/KSB-ORB/SIDC/sidc\\_txt.html](http://www.oma.be/KSB-ORB/SIDC/sidc_txt.html)

### See Also

`sunspot.month` is a longer version of `sunspots` that runs until 1988 rather than 1983.

### Examples

```
require(stats)
## Compare the monthly series
plot(sunspot.month, main = "sunspot.month [stats]", col = 2)
lines(sunspots) # "very barely" see something

## Now look at the difference :
all(tsp(sunspots)      [c(1,3)] ==
     tsp(sunspot.month)[c(1,3)]) ## Start & Periodicity are the same
n1 <- length(sunspots)
table(eq <- sunspots == sunspot.month[1:n1]) #> 132 are different !
i <- which(!eq)
rug(time(eq)[i])
s1 <- sunspots[i] ; s2 <- sunspot.month[i]
cbind(i = i, sunspots = s1, ss.month = s2,
      perc.diff = round(100*2*abs(s1-s2)/(s1+s2), 1))
```

---

<code>sunspot.year</code>	<i>Yearly Sunspot Data, 1700–1988</i>
---------------------------	---------------------------------------

---

**Description**

Yearly numbers of sunspots.

**Usage**

```
sunspot.year
```

**Format**

The univariate time series `sunspot.year` contains 289 observations, and is of class "ts".

**Note**

Prior to R 2.0.0 `sunspot.year` and `sunspot.year` were copied to the user's workspace (or elsewhere) by `data(sunspot)`. However, as package **lattice** has a dataset `sunspot` which can be retrieved by the same command, that usage has been removed.

**Source**

H. Tong (1996) *Non-Linear Time Series*. Clarendon Press, Oxford, p. 471.

---

<code>sunspots</code>	<i>Monthly Sunspot Numbers, 1749–1983</i>
-----------------------	---

---

**Description**

Monthly mean relative sunspot numbers from 1749 to 1983. Collected at Swiss Federal Observatory, Zurich until 1960, then Tokyo Astronomical Observatory.

**Usage**

```
sunspots
```

**Format**

A time series of monthly data from 1749 to 1983.

**Source**

Andrews, D. F. and Herzberg, A. M. (1985) *Data: A Collection of Problems from Many Fields for the Student and Research Worker*. New York: Springer-Verlag.

**See Also**

`sunspot.month` has a longer (and a bit different) series.

**Examples**

```
plot(sunspots, main = "sunspots data", xlab = "Year",
     ylab = "Monthly sunspot numbers")
```

---

 swiss

*Swiss Fertility and Socioeconomic Indicators (1888) Data*


---

**Description**

Standardized fertility measure and socio-economic indicators for each of 47 French-speaking provinces of Switzerland at about 1888.

**Usage**

```
swiss
```

**Format**

A data frame with 47 observations on 6 variables, *each* of which is in percent, i.e., in [0, 100].

[,1]	Fertility	$I_g$ , “common standardized fertility measure”
[,2]	Agriculture	% of males involved in agriculture as occupation
[,3]	Examination	% “draftees” receiving highest mark on army examination
[,4]	Education	% education beyond primary school for “draftees”.
[,5]	Catholic	% catholic (as opposed to “protestant”).
[,6]	Infant.Mortality	live births who live less than 1 year.

All variables but ‘Fertility’ give proportions of the population.

**Details**

(paraphrasing Mosteller and Tukey):

Switzerland, in 1888, was entering a period known as the “demographic transition”; i.e., its fertility was beginning to fall from the high level typical of underdeveloped countries.

The data collected are for 47 French-speaking “provinces” at about 1888.

Here, all variables are scaled to [0, 100], where in the original, all but “Catholic” were scaled to [0, 1].

**Note**

Files for all 182 districts in 1888 and other years are available at <http://opr.princeton.edu/archive/eufert/switz.html>.

They state that variables Examination and Education are averages for 1887, 1888 and 1889.

### Source

Project “16P5”, pages 549–551 in

Mosteller, F. and Tukey, J. W. (1977) *Data Analysis and Regression: A Second Course in Statistics*. Addison-Wesley, Reading Mass.

indicating their source as “Data used by permission of Franice van de Walle. Office of Population Research, Princeton University, 1976. Unpublished data assembled under NICHD contract number No 1-HD-O-2077.”

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### Examples

```
pairs(swiss, panel = panel.smooth, main = "swiss data",
      col = 3 + (swiss$Catholic > 50))
summary(lm(Fertility ~ . , data = swiss))
```

---

Theoph

*Pharmacokinetics of theophylline*

---

### Description

The Theoph data frame has 132 rows and 5 columns of data from an experiment on the pharmacokinetics of theophylline.

### Usage

Theoph

### Format

This data frame contains the following columns:

**Subject** an ordered factor with levels 1, . . . , 12 identifying the subject on whom the observation was made. The ordering is by increasing maximum concentration of theophylline observed.

**Wt** weight of the subject (kg).

**Dose** dose of theophylline administered orally to the subject (mg/kg).

**Time** time since drug administration when the sample was drawn (hr).

**conc** theophylline concentration in the sample (mg/L).

## Details

Boeckmann, Sheiner and Beal (1994) report data from a study by Dr. Robert Upton of the kinetics of the anti-asthmatic drug theophylline. Twelve subjects were given oral doses of theophylline then serum concentrations were measured at 11 time points over the next 25 hours.

These data are analyzed in Davidian and Giltinan (1995) and Pinheiro and Bates (2000) using a two-compartment open pharmacokinetic model, for which a self-starting model function, `SSfol`, is available.

## Source

Boeckmann, A. J., Sheiner, L. B. and Beal, S. L. (1994), *NONMEM Users Guide: Part V*, NONMEM Project Group, University of California, San Francisco.

Davidian, M. and Giltinan, D. M. (1995) *Nonlinear Models for Repeated Measurement Data*, Chapman & Hall (section 5.5, p. 145 and section 6.6, p. 176)

Pinheiro, J. C. and Bates, D. M. (2000) *Mixed-effects Models in S and S-PLUS*, Springer (Appendix A.29)

## See Also

[SSfol](#)

## Examples

```
require(stats)
coplot(conc ~ Time | Subject, data = Theoph, show = FALSE)
Theoph.4 <- subset(Theoph, Subject == 4)
fml <- nls(conc ~ SSfol(Dose, Time, lKe, lKa, lCl),
          data = Theoph.4)
summary(fml)
plot(conc ~ Time, data = Theoph.4,
     xlab = "Time since drug administration (hr)",
     ylab = "Theophylline concentration (mg/L)",
     main = "Observed concentrations and fitted model",
     sub = "Theophylline data - Subject 4 only",
     las = 1, col = 4)
xvals <- seq(0, par("usr")[2], len = 55)
lines(xvals, predict(fml, newdata = list(Time = xvals)),
      col = 4)
```

---

Titanic

*Survival of passengers on the Titanic*

---

## Description

This data set provides information on the fate of passengers on the fatal maiden voyage of the ocean liner ‘Titanic’, summarized according to economic status (class), sex, age and survival.

**Usage**

```
Titanic
```

**Format**

A 4-dimensional array resulting from cross-tabulating 2201 observations on 4 variables. The variables and their levels are as follows:

No	Name	Levels
1	Class	1st, 2nd, 3rd, Crew
2	Sex	Male, Female
3	Age	Child, Adult
4	Survived	No, Yes

**Details**

The sinking of the Titanic is a famous event, and new books are still being published about it. Many well-known facts—from the proportions of first-class passengers to the “women and children first” policy, and the fact that that policy was not entirely successful in saving the women and children in the third class—are reflected in the survival rates for various classes of passenger.

These data were originally collected by the British Board of Trade in their investigation of the sinking. Note that there is not complete agreement among primary sources as to the exact numbers on board, rescued, or lost.

Due in particular to the very successful film ‘Titanic’, the last years saw a rise in public interest in the Titanic. Very detailed data about the passengers is now available on the Internet, at sites such as *Encyclopedia Titanica* (<http://www.rmp1c.co.uk/eduweb/sites/phind>).

**Source**

Dawson, Robert J. MacG. (1995), The ‘Unusual Episode’ Data Revisited. *Journal of Statistics Education*, **3**. <http://www.amstat.org/publications/jse/v3n3/datasets.dawson.html>

The source provides a data set recording class, sex, age, and survival status for each person on board of the Titanic, and is based on data originally collected by the British Board of Trade and reprinted in:

British Board of Trade (1990), *Report on the Loss of the ‘Titanic’ (S.S.)*. British Board of Trade Inquiry Report (reprint). Gloucester, UK: Allan Sutton Publishing.

**Examples**

```
require(graphics)
mosaicplot(Titanic, main = "Survival on the Titanic")
## Higher survival rates in children?
apply(Titanic, c(3, 4), sum)
## Higher survival rates in females?
apply(Titanic, c(2, 4), sum)
## Use loglm() in package 'MASS' for further analysis ...
```

---

 ToothGrowth

*The Effect of Vitamin C on Tooth Growth in Guinea Pigs*


---

**Description**

The response is the length of odontoblasts (teeth) in each of 10 guinea pigs at each of three dose levels of Vitamin C (0.5, 1, and 2 mg) with each of two delivery methods (orange juice or ascorbic acid).

**Usage**

ToothGrowth

**Format**

A data frame with 60 observations on 3 variables.

[,1]	len	numeric	Tooth length
[,2]	supp	factor	Supplement type (VC or OJ).
[,3]	dose	numeric	Dose in milligrams.

**Source**

C. I. Bliss (1952) *The Statistics of Bioassay*. Academic Press.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
coplot(len ~ dose | supp, data = ToothGrowth, panel = panel.smooth,
       xlab = "ToothGrowth data: length vs dose, given type of supplement")
```

---

 treering

*Yearly Treering Data, -6000–1979*


---

**Description**

Contains normalized tree-ring widths in dimensionless units.

**Usage**

treering

**Format**

A univariate time series with 7981 observations. The object is of class "ts".  
Each tree ring corresponds to one year.

**Details**

The data were recorded by Donald A. Graybill, 1980, from Gt Basin Bristlecone Pine 2805M, 3726-11810 in Methuselah Walk, California.

**Source**

Time Series Data Library: <http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/>, series 'CA535.DAT'

**References**

For background on Bristlecone pines and Methuselah Walk, see <http://www.sonic.net/bristlecone/>; for some photos see <http://www.ltrr.arizona.edu/~hallman/sitephotos/meth.html>

---

trees

*Girth, Height and Volume for Black Cherry Trees*

---

**Description**

This data set provides measurements of the girth, height and volume of timber in 31 felled black cherry trees. Note that girth is the diameter of the tree (in inches) measured at 4 ft 6 in above the ground.

**Usage**

trees

**Format**

A data frame with 31 observations on 3 variables.

[,1]	Girth	numeric	Tree diameter in inches
[,2]	Height	numeric	Height in ft
[,3]	Volume	numeric	Volume of timber in cubic ft

**Source**

Ryan, T. A., Joiner, B. L. and Ryan, B. F. (1976) *The Minitab Student Handbook*. Duxbury Press.

## References

Atkinson, A. C. (1985) *Plots, Transformations and Regression*. Oxford University Press.

## Examples

```
pairs(trees, panel = panel.smooth, main = "trees data")
plot(log(Volume) ~ Girth, data = trees, log = "xy")
coplot(log(Volume) ~ log(Girth) | Height, data = trees,
       panel = panel.smooth)
summary(fm1 <- lm(log(Volume) ~ log(Girth), data = trees))
summary(fm2 <- update(fm1, ~ . + log(Height), data = trees))
step(fm2)
## i.e., Volume ~ c * Height * Girth^2 seems reasonable
```

---

UCBAdmissions

*Student Admissions at UC Berkeley*

---

## Description

Aggregate data on applicants to graduate school at Berkeley for the six largest departments in 1973 classified by admission and sex.

## Usage

UCBAdmissions

## Format

A 3-dimensional array resulting from cross-tabulating 4526 observations on 3 variables. The variables and their levels are as follows:

No	Name	Levels
1	Admit	Admitted, Rejected
2	Gender	Male, Female
3	Dept	A, B, C, D, E, F

## Details

This data set is frequently used for illustrating Simpson's paradox, see Bickel et al. (1975). At issue is whether the data show evidence of sex bias in admission practices. There were 2691 male applicants, of whom 1198 (44.5%) were admitted, compared with 1835 female applicants of whom 557 (30.4%) were admitted. This gives a sample odds ratio of 1.83, indicating that males were almost twice as likely to be admitted. In fact, graphical methods (as in the example below) or log-linear modelling show that the apparent association between admission and sex stems from differences in the tendency of males and females to apply to the individual departments (females used to apply "more" to departments with higher rejection rates).

This data set can also be used for illustrating methods for graphical display of categorical data, such

as the general-purpose mosaic plot or the “fourfold display” for 2-by-2-by- $k$  tables. See the home page of Michael Friendly (<http://www.math.yorku.ca/SCS/friendly.html>) for further information.

## References

Bickel, P. J., Hammel, E. A., and O’Connell, J. W. (1975) Sex bias in graduate admissions: Data from Berkeley. *Science*, **187**, 398–403.

## Examples

```
require(graphics)
## Data aggregated over departments
apply(UCBAdmissions, c(1, 2), sum)
mosaicplot(apply(UCBAdmissions, c(1, 2), sum),
           main = "Student admissions at UC Berkeley")
## Data for individual departments
opar <- par(mfrow = c(2, 3), oma = c(0, 0, 2, 0))
for(i in 1:6)
  mosaicplot(UCBAdmissions[, , i],
            xlab = "Admit", ylab = "Sex",
            main = paste("Department", LETTERS[i]))
mtext(expression(bold("Student admissions at UC Berkeley")),
       outer = TRUE, cex = 1.5)
par(opar)
```

---

UKDriverDeaths

*Road Casualties in Great Britain 1969–84*

---

## Description

UKDriverDeaths is a time series giving the monthly totals of car drivers in Great Britain killed or seriously injured Jan 1969 to Dec 1984. Compulsory wearing of seat belts was introduced on 31 Jan 1983.

Seatbelts is more information on the same problem.

## Usage

```
UKDriverDeaths
Seatbelts
```

## Format

Seatbelts is a multiple time series, with columns

**DriversKilled** car drivers killed.

**drivers** same as UKDriverDeaths.

**front** front-seat passengers killed or seriously injured.

**rear** rear-seat passengers killed or seriously injured.  
**kms** distance driven.  
**PetrolPrice** petrol price.  
**VanKilled** number of van ('light goods vehicle') drivers.  
**law** 0/1: was the law in effect that month?

### Source

Harvey, A.C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, pp. 519–523.

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press. <http://www.ssfpack.com/dkbook/>

### References

Harvey, A. C. and Durbin, J. (1986) The effects of seat belt legislation on British road casualties: A case study in structural time series modelling. *Journal of the Royal Statistical Society series B*, **149**, 187–227.

### Examples

```
require(stats)
## work with pre-seatbelt period to identify a model, use logs
work <- window(log10(UKDriverDeaths), end = 1982+11/12)
par(mfrow = c(3,1))
plot(work); acf(work); pacf(work)
par(mfrow = c(1,1))
(fit <- arima(work, c(1,0,0), seasonal = list(order= c(1,0,0))))
z <- predict(fit, n.ahead = 24)
ts.plot(log10(UKDriverDeaths), z$pred, z$pred+2*z$se, z$pred-2*z$se,
        lty = c(1,3,2,2), col = c("black", "red", "blue", "blue"))

## now see the effect of the explanatory variables
X <- Seatbelts[, c("kms", "PetrolPrice", "law")]
X[, 1] <- log10(X[, 1]) - 4
arima(log10(Seatbelts[, "drivers"]), c(1,0,0),
      seasonal = list(order= c(1,0,0)), xreg = X)
```

---

 UKgas

*UK Quarterly Gas Consumption*


---

### Description

Quarterly UK gas consumption from 1960Q1 to 1986Q4, in millions of therms.

### Usage

UKgas

**Format**

A quarterly time series of length 108.

**Source**

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press. <http://www.ssfpack.com/dkbook/>

**Examples**

```
## maybe str(UKgas) ; plot(UKgas) ...
```

---

UKLungDeaths

*Monthly Deaths from Lung Diseases in the UK*

---

**Description**

Three time series giving the monthly deaths from bronchitis, emphysema and asthma in the UK, 1974–1979, both sexes (ldeaths), males (mdeaths) and females (fdeaths).

**Usage**

```
ldeaths
fdeaths
mdeaths
```

**Source**

P. J. Diggle (1990) *Time Series: A Biostatistical Introduction*. Oxford, table A.3

**Examples**

```
require(stats) # for time
plot(ldeaths)
plot(mdeaths, fdeaths)
## Better labels:
yr <- floor(tt <- time(mdeaths))
plot(mdeaths, fdeaths,
      xy.labels = paste(month.abb[12*(tt - yr)], yr-1900, sep=""))
```

---

USAccDeaths      *Accidental Deaths in the US 1973–1978*

---

**Description**

A time series giving the monthly totals of accidental deaths in the USA. The values for the first six months of 1979 are 7798 7406 8363 8460 9217 9316.

**Usage**

USAccDeaths

**Source**

P. J. Brockwell and R. A. Davis (1991) *Time Series: Theory and Methods*. Springer, New York.

---

USArrests      *Violent Crime Rates by US State*

---

**Description**

This data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.

**Usage**

USArrests

**Format**

A data frame with 50 observations on 4 variables.

[,1]	Murder	numeric	Murder arrests (per 100,000)
[,2]	Assault	numeric	Assault arrests (per 100,000)
[,3]	UrbanPop	numeric	Percent urban population
[,4]	Rape	numeric	Rape arrests (per 100,000)

**Source**

World Almanac and Book of facts 1975. (Crime rates).

Statistical Abstracts of the United States 1975. (Urban rates).

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**See Also**

The [state](#) data sets.

**Examples**

```
pairs(USArrests, panel = panel.smooth, main = "USArrests data")
```

---

USJudgeRatings	<i>Lawyers' Ratings of State Judges in the US Superior Court</i>
----------------	--

---

**Description**

Lawyers' ratings of state judges in the US Superior Court.

**Usage**

```
USJudgeRatings
```

**Format**

A data frame containing 43 observations on 12 numeric variables.

[,1]	CONT	Number of contacts of lawyer with judge.
[,2]	INTG	Judicial integrity.
[,3]	DMNR	Demeanor.
[,4]	DILG	Diligence.
[,5]	CFMG	Case flow managing.
[,6]	DECI	Prompt decisions.
[,7]	PREP	Preparation for trial.
[,8]	FAMI	Familiarity with law.
[,9]	ORAL	Sound oral rulings.
[,10]	WRIT	Sound written rulings.
[,11]	PHYS	Physical ability.
[,12]	RTEN	Worthy of retention.

**Source**

New Haven Register, 14 January, 1977 (from John Hartigan).

**Examples**

```
pairs(USJudgeRatings, main = "USJudgeRatings data")
```

---

USPersonalExpenditure  
*Personal Expenditure Data*

---

**Description**

This data set consists of United States personal expenditures (in billions of dollars) in the categories; food and tobacco, household operation, medical and health, personal care, and private education for the years 1940, 1945, 1950, 1955 and 1960.

**Usage**

USPersonalExpenditure

**Format**

A matrix with 5 rows and 5 columns.

**Source**

The World Almanac and Book of Facts, 1962, page 756.

**References**

Tukey, J. W. (1977) *Exploratory Data Analysis*. Addison-Wesley.

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

**Examples**

```
require(stats) # for medpolish
USPersonalExpenditure
medpolish(log10(USPersonalExpenditure))
```

---

uspop *Populations Recorded by the US Census*

---

**Description**

This data set gives the population of the United States (in millions) as recorded by the decennial census for the period 1790–1970.

**Usage**

uspop

**Format**

A time series of 19 values.

**Source**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
plot(uspop, log = "y", main = "uspop data", xlab = "Year",
     ylab = "U.S. Population (millions)")
```

---

VADeaths

*Death Rates in Virginia (1940)*

---

**Description**

Death rates per 100 in Virginia in 1940.

**Usage**

VADeaths

**Format**

A matrix with 5 rows and 5 columns.

**Details**

The death rates are cross-classified by age group (rows) and population group (columns). The age groups are: 50–54, 55–59, 60–64, 65–69, 70–74 and the population groups are Rural/Male, Rural/Female, Urban/Male and Urban/Female.

This provides a rather nice 3-way analysis of variance example.

**Source**

Moyneau, L., Gilliam, S. K., and Florant, L. C. (1947) Differences in Virginia death rates by color, sex, age, and rural or urban residence. *American Sociological Review*, **12**, 525–535.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

**Examples**

```
require(stats)
n <- length(dr <- c(VADeaths))
nam <- names(VADeaths)
d.VAD <- data.frame(
  Drate = dr,
  age = rep(ordered(rownames(VADeaths)),length=n),
  gender= gl(2,5,n, labels= c("M", "F")),
  site = gl(2,10, labels= c("rural", "urban")))
coplot(Drate ~ as.numeric(age) | gender * site, data = d.VAD,
  panel = panel.smooth, xlab = "VADeaths data - Given: gender")
summary(aov.VAD <- aov(Drate ~ .^2, data = d.VAD))
opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(aov.VAD)
par(opar)
```

---

volcano

*Topographic Information on Auckland's Maunga Whau Volcano*


---

**Description**

Maunga Whau (Mt Eden) is one of about 50 volcanos in the Auckland volcanic field. This data set gives topographic information for Maunga Whau on a 10m by 10m grid.

**Usage**

```
volcano
```

**Format**

A matrix with 87 rows and 61 columns, rows corresponding to grid lines running east to west and columns to grid lines running south to north.

**Source**

Digitized from a topographic map by Ross Ihaka. These data should not be regarded as accurate.

**See Also**

[filled.contour](#) for a nice plot.

**Examples**

```
require(graphics)
filled.contour(volcano, color = terrain.colors, asp = 1)
title(main = "volcano data: filled contour map")
```

---

 warpbreaks

*The Number of Breaks in Yarn during Weaving*


---

### Description

This data set gives the number of warp breaks per loom, where a loom corresponds to a fixed length of yarn.

### Usage

```
warpbreaks
```

### Format

A data frame with 54 observations on 3 variables.

[ ,1]	breaks	numeric	The number of breaks
[ ,2]	wool	factor	The type of wool (A or B)
[ ,3]	tension	factor	The level of tension (L, M, H)

There are measurements on 9 looms for each of the six types of warp (AL, AM, AH, BL, BM, BH).

### Source

Tippett, L. H. C. (1950) *Technological Applications of Statistics*. Wiley. Page 106.

### References

Tukey, J. W. (1977) *Exploratory Data Analysis*. Addison-Wesley.

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

### See Also

[xtabs](#) for ways to display these data as a table.

### Examples

```
summary(warpbreaks)
opar <- par(mfrow = c(1,2), oma = c(0, 0, 1.1, 0))
plot(breaks ~ tension, data = warpbreaks, col = "lightgray",
      varwidth = TRUE, subset = wool == "A", main = "Wool A")
plot(breaks ~ tension, data = warpbreaks, col = "lightgray",
      varwidth = TRUE, subset = wool == "B", main = "Wool B")
mtext("warpbreaks data", side = 3, outer = TRUE)
par(opar)
summary(fml <- lm(breaks ~ wool*tension, data = warpbreaks))
anova(fml)
```

---

women

*Average Heights and Weights for American Women*

---

**Description**

This data set gives the average heights and weights for American women aged 30–39.

**Usage**

women

**Format**

A data frame with 15 observations on 2 variables.

```
[ ,1] height numeric Height (in)
[ ,2] weight numeric Weight (lbs)
```

**Details**

The data set appears to have been taken from the American Society of Actuaries *Build and Blood Pressure Study* for some (unknown to us) earlier year.

The World Almanac notes: “The figures represent weights in ordinary indoor clothing and shoes, and heights with shoes”.

**Source**

The World Almanac and Book of Facts, 1975.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

**Examples**

```
plot(women, xlab = "Height (in)", ylab = "Weight (lb)",
     main = "women data: American women aged 30-39")
```

---

WorldPhones

*The World's Telephones*

---

**Description**

The number of telephones in various regions of the world (in thousands).

**Usage**

phones

**Format**

A matrix with 7 rows and 8 columns. The columns of the matrix give the figures for a given region, and the rows the figures for a year.

The regions are: North America, Europe, Asia, South America, Oceania, Africa, Central America.

The years are: 1951, 1956, 1957, 1958, 1959, 1960, 1961.

**Warning**

Prior to R 2.0.0 this dataset was called `phones` and conflicted with a dataset in package **MASS**.

**Source**

AT&T (1961) *The World's Telephones*.

**References**

McNeil, D. R. (1977) *Interactive Data Analysis*. New York: Wiley.

**Examples**

```
matplot(rownames(WorldPhones), WorldPhones, type = "b", log = "y",
        xlab = "Year", ylab = "Number of telephones (1000's)")
legend(1951.5, 80000, colnames(WorldPhones), col = 1:6, lty = 1:5,
       pch = rep(21, 7))
title(main = "World phones data: log scale for response")
```

---

WWWusage

*Internet Usage per Minute*

---

**Description**

A time series of the numbers of users connected to the Internet through a server every minute.

**Usage**

WWWusage

**Format**

A time series of length 100.

**Source**

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press. <http://www.ssfpack.com/dkbook/>

**References**

Makridakis, S., Wheelwright, S. C. and Hyndman, R. J. (1998) *Forecasting: Methods and Applications*. Wiley.

**Examples**

```
work <- diff(WWWusage)
par(mfrow = c(2,1)); plot(WWWusage); plot(work)
## Not run:
require(stats)
aics <- matrix(, 6, 6, dimnames=list(p=0:5, q=0:5))
for(q in 1:5) aics[1, 1+q] <- arima(WWWusage, c(0,1,q),
  optim.control = list(maxit = 500))$aic
for(p in 1:5)
  for(q in 0:5) aics[1+p, 1+q] <- arima(WWWusage, c(p,1,q),
    optim.control = list(maxit = 500))$aic
round(aics - min(aics, na.rm=TRUE), 2)
## End(Not run)
```



## Chapter 3

# The grDevices package

---

check.options      *Set Options with Consistency Checks*

---

### Description

Utility function for setting options with some consistency checks. The `attributes` of the new settings in `new` are checked for consistency with the *model* (often default) list in `name.opt`.

### Usage

```
check.options(new, name.opt, reset = FALSE, assign.opt = FALSE,
              envir = .GlobalEnv,
              check.attributes = c("mode", "length"),
              override.check = FALSE)
```

### Arguments

<code>new</code>	a <i>named</i> list
<code>name.opt</code>	character with the name of R object containing the “model” (default) list.
<code>reset</code>	logical; if TRUE, reset the options from <code>name.opt</code> . If there is more than one R object with name <code>name.opt</code> , remove the first one in the <code>search()</code> path.
<code>assign.opt</code>	logical; if TRUE, assign the ...
<code>envir</code>	the <code>environment</code> used for <code>get</code> and <code>assign</code> .
<code>check.attributes</code>	character containing the attributes which <code>check.options</code> should check.
<code>override.check</code>	logical vector of length <code>length(new)</code> (or 1 which entails recycling). For those <code>new[i]</code> where <code>override.check[i] == TRUE</code> , the checks are overridden and the changes made anyway.

**Value**

A list of components with the same names as the one called `name.opt`. The values of the components are changed from the new list, as long as these pass the checks (when these are not overridden according to `override.check`).

**Author(s)**

Martin Maechler

**See Also**

`ps.options` which uses `check.options`.

**Examples**

```
(L1 <- list(a=1:3, b=pi, ch="CH"))
check.options(list(a=0:2), name.opt = "L1")
check.options(NULL, reset = TRUE, name.opt = "L1")
```

---

 col2rgb

---

*Color to RGB Conversion*


---

**Description**

“Any R color” to RGB (red/green/blue) conversion.

**Usage**

```
col2rgb(col, alpha = FALSE)
```

**Arguments**

<code>col</code>	vector of any of the three kind of R colors, i.e., either a color name (an element of <code>colors()</code> ), a hexadecimal string of the form <code>"#rrggbb"</code> , or an integer <code>i</code> meaning <code>palette()[i]</code> .
<code>alpha</code>	logical value indicating whether alpha channel values should be returned.

**Details**

For integer colors, 0 is shorthand for the current `par("bg")`, and `NA` means “nothing” which effectively does not draw the corresponding item.

For character colors, `"NA"` is equivalent to `NA` above.

**Value**

an integer matrix with three or four rows and number of columns the length (and names if any) as `col`.

**Author(s)**

Martin Maechler

**See Also**[rgb](#), [colors](#), [palette](#), etc.**Examples**

```

col2rgb("peachpuff")
col2rgb(c(blu = "royalblue", reddish = "tomato")) # names kept

col2rgb(1:8)# the ones from the palette() :

col2rgb(paste("gold", 1:4, sep=""))

col2rgb("#08a0ff")
## all three kind of colors mixed :
col2rgb(c(red="red", palette= 1:3, hex="#abcdef"))

##-- NON-INTRODUCTORY examples --

grC <- col2rgb(paste("gray",0:100,sep=""))
table(print(diff(grC["red",])))# '2' or '3': almost equidistant
## The 'named' grays are in between {"slate gray" is not gray, strictly}
col2rgb(c(g66="gray66", darkg= "dark gray", g67="gray67",
          g74="gray74", gray =      "gray", g75="gray75",
          g82="gray82", light="light gray", g83="gray83"))

crgb <- col2rgb(cc <- colors())
colnames(crgb) <- cc
t(crgb)## The whole table

ccodes <- c(256^(2:0) %% crgb)## = internal codes
## How many names are 'aliases' of each other:
table(tcc <- table(ccodes))
length(uc <- unique(sort(ccodes))) # 502
## All the multiply named colors:
mult <- uc[tcc >= 2]
cl <- lapply(mult, function(m) cc[ccodes == m])
names(cl) <- apply(col2rgb(sapply(cl, function(x)x[1])),
                  2, function(n)paste(n, collapse=","))
utils::str(cl)
## Not run:
if(require(xgobi)) { ## Look at the color cube dynamically :
  tc <- t(crgb[, !duplicated(ccodes)])
  table(is.gray <- tc[,1] == tc[,2] & tc[,2] == tc[,3])# (397, 105)
  xgobi(tc, color = c("gold", "gray")[1 + is.gray])
}
## End(Not run)

```

---

`colors`*Color Names*

---

**Description**

Returns the built-in color names which R knows about.

**Usage**

```
colors()  
colours()
```

**Details**

These color names can be used with a `col=` specification in graphics functions.

An even wider variety of colors can be created with primitives `rgb` and `hsv` or the derived `rainbow`, `heat.colors`, etc.

**Value**

A character vector containing all the built-in color names.

**See Also**

[palette](#) for setting the “palette” of colors for `par(col=<num>)`; [rgb](#), [hsv](#), [gray](#); [rainbow](#) for a nice example; and [heat.colors](#), [topo.colors](#) for images.

[col2rgb](#) for translating to RGB numbers and extended examples.

**Examples**

```
cl <- colors()  
length(cl); cl[1:20]
```

---

`dev.interactive`*Test if an Interactive Graphics Device is in Use*

---

**Description**

Test if an interactive graphics device is in use.

**Usage**

```
dev.interactive()
```

## Details

The X11 (Unix), windows (Windows) and quartz (MacOS X) are regarded as interactive, together with GTK and gnome (used with the GNOME GUI: GTK is available in package **gtkDevice** and gnome is expected to be in a package **gnomeDevice**).

## Value

`dev.interactive()` returns a logical, TRUE iff an interactive (screen) device is in use.

## See Also

[Devices](#) for the available devices on your platform.

---

dev.xxx

*Control Multiple Devices*

---

## Description

These functions provide control over multiple graphics devices.

Only one device is the *active* device. This is the device in which all graphics operations occur.

Devices are associated with a name (e.g., "X11" or "postscript") and a number; the "null device" is always device 1.

`dev.off` shuts down the specified (by default the current) device. `graphics.off()` shuts down all open graphics devices.

`dev.set` makes the specified device the active device.

## Usage

```
dev.cur()
dev.list()
dev.next(which = dev.cur())
dev.prev(which = dev.cur())
dev.off(which = dev.cur())
dev.set(which = dev.next())
graphics.off()
```

## Arguments

`which`            An integer specifying a device number

**Value**

`dev.cur` returns the number and name of the active device, or 1, the null device, if none is active.

`dev.list` returns the numbers of all open devices, except device 1, the null device. This is a numeric vector with a `names` attribute giving the names, or `NULL` if there is no open device.

`dev.next` and `dev.prev` return the number and name of the next / previous device in the list of devices. The list is regarded as a circular list, and "null device" will be included only if there are no open devices.

`dev.off` returns the name and number of the new active device (after the specified device has been shut down).

`dev.set` returns the name and number of the new active device.

**See Also**

[Devices](#), such as [postscript](#), etc.

[layout](#) and its links for setting up plotting regions on the current device.

**Examples**

```
## Not run:
## Unix-specific example
x11()
plot(1:10)
x11()
plot(rnorm(10))
dev.set(dev.prev())
abline(0,1)# through the 1:10 points
dev.set(dev.next())
abline(h=0, col="gray")# for the residual plot
dev.set(dev.prev())
dev.off(); dev.off()#- close the two X devices
## End(Not run)
```

---

dev2

---

*Copy Graphics Between Multiple Devices*


---

**Description**

`dev.copy` copies the graphics contents of the current device to the device specified by `which` or to a new device which has been created by the function specified by `device` (it is an error to specify both `which` and `device`). (If recording is off on the current device, there are no contents to copy: this will result in no plot or an empty plot.) The device copied to becomes the current device.

`dev.print` copies the graphics contents of the current device to a new device which has been created by the function specified by `device` and then shuts the new device.

`dev.copy2eps` is similar to `dev.print` but produces an EPSF output file, in portrait orientation (`horizontal = FALSE`)

`dev.control` allows the user to control the recording of graphics operations in a device. If `displaylist` is "inhibit" ("enable") then recording is turned off (on). It is only safe to change this at the beginning of a plot (just before or just after a new page). Initially recording is on for screen devices, and off for print devices.

### Usage

```
dev.copy(device, ..., which = dev.next())
dev.print(device = postscript, ...)
dev.copy2eps(...)
dev.control(displaylist = c("inhibit", "enable"))
```

### Arguments

<code>device</code>	A device function (e.g., <code>x11</code> , <code>postscript</code> , ...)
<code>...</code>	Arguments to the device function above. For <code>dev.print</code> , this includes <code>which</code> and by default any <code>postscript</code> arguments.
<code>which</code>	A device number specifying the device to copy to
<code>displaylist</code>	A character string: the only valid values are "inhibit" and "enable".

### Details

For `dev.copy2eps`, `width` and `height` are taken from the current device unless otherwise specified. If just one of `width` and `height` is specified, the other is adjusted to preserve the aspect ratio of the device being copied. The default file name is `Rplot.eps`.

The default for `dev.print` is to produce and print a postscript copy, if `options("printcmd")` is set suitably.

`dev.print` is most useful for producing a postscript print (its default) when the following applies. Unless `file` is specified, the plot will be printed. Unless `width`, `height` and `pointsize` are specified the plot dimensions will be taken from the current device, shrunk if necessary to fit on the paper. (`pointsize` is rescaled if the plot is shrunk.) If `horizontal` is not specified and the plot can be printed at full size by switching its value this is done instead of shrinking the plot region.

If `dev.print` is used with a specified device (even `postscript`) it sets the width and height in the same way as `dev.copy2eps`.

### Value

`dev.copy` returns the name and number of the device which has been copied to.

`dev.print` and `dev.copy2eps` return the name and number of the device which has been copied from.

### Note

Most devices (including all screen devices) have a display list which records all of the graphics operations that occur in the device. `dev.copy` copies graphics contents by copying the display list from one device to another device. Also, automatic redrawing of graphics contents following the resizing of a device depends on the contents of the display list.

After the command `dev.control("inhibit")`, graphics operations are not recorded in the display list so that `dev.copy` and `dev.print` will not copy anything and the contents of a device will not be redrawn automatically if the device is resized.

The recording of graphics operations is relatively expensive in terms of memory so the command `dev.control("inhibit")` can be useful if memory usage is an issue.

### See Also

[dev.cur](#) and other `dev.xxx` functions

### Examples

```
## Not run:
x11()
plot(rnorm(10), main="Plot 1")
dev.copy(device=x11)
mtext("Copy 1", 3)
dev.print(width=6, height=6, horizontal=FALSE) # prints it
dev.off(dev.prev())
dev.off()
## End(Not run)
```

---

dev2bitmap

*Graphics Device for Bitmap Files via GhostScript*


---

### Description

`bitmap` generates a graphics file. `dev2bitmap` copies the current graphics device to a file in a graphics format.

### Usage

```
bitmap(file, type = "png256", height = 6, width = 6, res = 72,
        pointsize, ...)

dev2bitmap(file, type = "png256", height = 6, width = 6, res = 72,
           pointsize, ...)
```

### Arguments

<code>file</code>	The output file name, with an appropriate extension.
<code>type</code>	The type of bitmap. the default is "png256".
<code>height</code>	The plot height, in inches.
<code>width</code>	The plot width, in inches.
<code>res</code>	Resolution, in dots per inch.
<code>pointsize</code>	The pointsize to be used for text: defaults to something reasonable given the width and height
<code>...</code>	Other parameters passed to <a href="#">postscript</a> .

## Details

`dev2bitmap` works by copying the current device to a [postscript](#) device, and post-processing the output file using `ghostscript`. `bitmap` works in the same way using a `postscript` device and postprocessing the output as “printing”.

You will need a version of `ghostscript` (5.10 and later have been tested): the full path to the executable can be set by the environment variable `R_GSCMD`.

The types available will depend on the version of `ghostscript`, but are likely to include `"pcxmono"`, `"pcxgray"`, `"pcx16"`, `"pcx256"`, `"pcx24b"`, `"pcxcmyk"`, `"pbm"`, `"pbmraw"`, `"pgm"`, `"pgmraw"`, `"pgnm"`, `"pgnmraw"`, `"pnm"`, `"pnmraw"`, `"ppm"`, `"ppmraw"`, `"pkm"`, `"pkmraw"`, `"tiffcrle"`, `"tiffg3"`, `"tiffg32d"`, `"tiffg4"`, `"tiffllzw"`, `"tiffpack"`, `"tiff12nc"`, `"tiff24nc"`, `"psmono"`, `"psgray"`, `"psrgb"`, `"bit"`, `"bitrgb"`, `"bitcmyk"`, `"pngmono"`, `"pnggray"`, `"png16"`, `"png256"`, `"png16m"`, `"jpeg"`, `"jpeggray"`, `"pdfwrite"`.

Note: despite the name of the functions they can produce PDF *via* `type = "pdfwrite"`, and the PDF produced is not bitmapped.

For formats which contain a single image, a file specification like `Rplots%03d.png` can be used: this is interpreted by GhostScript.

For `dev2bitmap` if just one of `width` and `height` is specified, the other is chosen to preserve aspect ratio of the device being copied.

## Value

None.

## See Also

[postscript](#), [png](#) and [jpeg](#) and on Windows `bmp`.

[pdf](#) generate PDF directly.

To display an array of data, see [image](#).

## Description

The following graphics devices are currently available:

- [postscript](#) Writes PostScript graphics commands to a file
- [pdf](#) Write PDF graphics commands to a file
- [pictex](#) Writes LaTeX/PicTeX graphics commands to a file
- [xfig](#) Device for XFIG graphics file format
- [bitmap](#) `bitmap` pseudo-device via GhostScript (if available).

The following devices will be available if R was compiled to use them and started with the appropriate ‘--gui’ argument:

- [X11](#) The graphics driver for the X11 Window system
- [png](#) PNG bitmap device
- [jpeg](#) JPEG bitmap device

None of these are available under R CMD BATCH.

### Details

If no device is open, using a high-level graphics function will cause a device to be opened. Which device is given by `options("device")` which is initially set as the most appropriate for each platform: a screen device for most interactive use and `postscript` otherwise. The exception is interactive use under Unix if no screen device is known to be available, when `postscript()` is used for most systems; `pdf()` for Mac OS X.

### See Also

The individual help files for further information on any of the devices listed here;

[dev.interactive](#), [dev.cur](#), [dev.print](#), [graphics.off](#), [image](#), [dev2bitmap.capabilities](#) to see if [X11](#), [jpeg](#) and [png](#) are available.

### Examples

```
## Not run:
## open the default screen device on this platform if no device is
## open
if(dev.cur() == 1) get(getOption("device"))()
## End(Not run)
```

---

gray

*Gray Level Specification*

---

### Description

Create a vector of colors from a vector of gray levels.

### Usage

```
gray(level)
grey(level)
```

### Arguments

`level` a vector of desired gray levels between 0 and 1; zero indicates "black" and one indicates "white".

**Details**

The values returned by `gray` can be used with a `col=` specification in graphics functions or in `par`.

`grey` is an alias for `gray`.

**Value**

A vector of “colors” of the same length as `level`.

**See Also**

`rainbow`, `hsv`, `rgb`.

**Examples**

```
gray(0:8 / 8)
```

---

Hershey

*Hershey Vector Fonts in R*

---

**Description**

If the `vfont` argument to one of the text-drawing functions (`text`, `mtext`, `title`, `axis`, and `contour`) is a character vector of length 2, Hershey vector fonts are used to render the text.

These fonts have two advantages:

1. vector fonts describe each character in terms of a set of points; R renders the character by joining up the points with straight lines. This intimate knowledge of the outline of each character means that R can arbitrarily transform the characters, which can mean that the vector fonts look better for rotated and 3d text.
2. this implementation was adapted from the GNU libplot library which provides support for non-ASCII and non-English fonts. This means that it is possible, for example, to produce weird plotting symbols and Japanese characters.

Drawback:

You cannot use mathematical expressions (`plotmath`) with Hershey fonts.

**Usage**

Hershey

## Details

The Hershey characters are organised into a set of fonts, which are specified by a typeface (e.g., `serif` or `sans serif`) and a fontindex or “style” (e.g., `plain` or `italic`). The first element of `vfont` specifies the typeface and the second element specifies the fontindex. The first table produced by `demo(Hershey)` shows the character `a` produced by each of the different fonts.

The available `typeface` and `fontindex` values are available as list components of the variable `Hershey`. The allowed pairs for `(typeface, fontindex)` are:

<code>serif</code>	<code>plain</code>
<code>serif</code>	<code>italic</code>
<code>serif</code>	<code>bold</code>
<code>serif</code>	<code>bold italic</code>
<code>serif</code>	<code>cyrillic</code>
<code>serif</code>	<code>oblique cyrillic</code>
<code>serif</code>	<code>EUC</code>
<code>sans serif</code>	<code>plain</code>
<code>sans serif</code>	<code>italic</code>
<code>sans serif</code>	<code>bold</code>
<code>sans serif</code>	<code>bold italic</code>
<code>script</code>	<code>plain</code>
<code>script</code>	<code>italic</code>
<code>script</code>	<code>bold</code>
<code>gothic english</code>	<code>plain</code>
<code>gothic german</code>	<code>plain</code>
<code>gothic italian</code>	<code>plain</code>
<code>serif symbol</code>	<code>plain</code>
<code>serif symbol</code>	<code>italic</code>
<code>serif symbol</code>	<code>bold</code>
<code>serif symbol</code>	<code>bold italic</code>
<code>sans serif symbol</code>	<code>plain</code>
<code>sans serif symbol</code>	<code>italic</code>

and the indices of these are available as `Hershey$allowed`.

**Escape sequences:** The string to be drawn can include escape sequences, which all begin with a `\`. When `R` encounters a `\`, rather than drawing the `\`, it treats the subsequent character(s) as a coded description of what to draw.

One useful escape sequence (in the current context) is of the form: `\123`. The three digits following the `\` specify an octal code for a character. For example, the octal code for `p` is 160 so the strings `"p"` and `"\160"` are equivalent. This is useful for producing characters when there is not an appropriate key on your keyboard.

The other useful escape sequences all begin with `\\`. These are described below. Remember that backslashes have to be doubled in `R` character strings, so they need to be entered with *four* backslashes.

**Symbols:** an entire string of Greek symbols can be produced by selecting the `Serif Symbol` or `Sans Serif Symbol` typeface. To allow Greek symbols to be embedded in a string which uses a non-symbol typeface, there are a set of symbol escape sequences of the form `\\ab`. For example,

the escape sequence `\\*a` produces a Greek alpha. The second table in `demo(Hershey)` shows all of the symbol escape sequences and the symbols that they produce.

**ISO Latin-1:** further escape sequences of the form `\\ab` are provided for producing ISO Latin-1 characters (for example, if you only have a US keyboard). Another option is to use the appropriate octal code. The (non-ASCII) ISO Latin-1 characters are in the range 241...377. For example, `\366` produces the character `o` with an umlaut. The third table in `demo(Hershey)` shows all of the ISO Latin-1 escape sequences.

**Special Characters:** a set of characters are provided which do not fall into any standard font. These can only be accessed by escape sequence. For example, `\\LI` produces the zodiac sign for Libra, and `\\JU` produces the astronomical sign for Jupiter. The fourth table in `demo(Hershey)` shows all of the special character escape sequences.

**Cyrillic Characters:** cyrillic characters are implemented according to the KOI8-R encoding. On a US keyboard, these can be produced using the Serif typeface and Cyrillic (or Oblique Cyrillic) fontindex and specifying an octal code in the range 300 to 337 for lower case characters or 340 to 377 for upper case characters. The fifth table in `demo(Hershey)` shows the octal codes for the available cyrillic characters.

**Japanese Characters:** 83 Hiragana, 86 Katakana, and 603 Kanji characters are implemented according to the EUC (Extended Unix Code) encoding. Each character is identified by a unique hexadecimal code. The Hiragana characters are in the range 0x2421 to 0x2473, Katakana are in the range 0x2521 to 0x2576, and Kanji are (scattered about) in the range 0x3021 to 0x6d55. When using the Serif typeface and EUC fontindex, these characters can be produced by a *pair* of octal codes. Given the hexadecimal code (e.g., 0x2421), take the first two digits and add 0x80 and do the same to the second two digits (e.g., 0x21 and 0x24 become 0xa4 and 0xa1), then convert both to octal (e.g., 0xa4 and 0xa1 become 244 and 241). For example, the first Hiragana character is produced by `\244\241`.

It is also possible to use the hexadecimal code directly. This works for all non-EUC fonts by specifying an escape sequence of the form `\\#J1234`. For example, the first Hiragana character is produced by `\\#J2421`.

The Kanji characters may be specified in a third way, using the so-called "Nelson Index", by specifying an escape sequence of the form `\\#N1234`. For example, the Kanji for "one" is produced by `\\#N0001`.

`demo(Japanese)` shows the available Japanese characters.

**Raw Hershey Glyphs:** all of the characters in the Hershey fonts are stored in a large array. Some characters are not accessible in any of the Hershey fonts. These characters can only be accessed via an escape sequence of the form `\\#H1234`. For example, the fleur-de-lys is produced by `\\#H0746`. The sixth and seventh tables of `demo(Hershey)` shows all of the available raw glyphs.

## References

<http://www.gnu.org/software/plotutils/plotutils.html>

## See Also

`demo(Hershey)`, [text](#), [contour](#).

[Japanese](#) for the Japanese characters in the Hershey fonts.

**Examples**

```
Hershey
## for tables of examples, see demo(Hershey)
```

---

hsv

*HSV Color Specification*


---

**Description**

Create a vector of colors from vectors specifying hue, saturation and value.

**Usage**

```
hsv(h = 1, s = 1, v = 1, gamma = 1, alpha)
```

**Arguments**

<code>h, s, v</code>	numeric vectors of values in the range $[0, 1]$ for “hue”, “saturation” and “value” to be combined to form a vector of colors. Values in shorter arguments are recycled.
<code>gamma</code>	a “gamma correction” exponent, $\gamma$
<code>alpha</code>	numeric value in the range $[0, 1]$ for alpha transparency channel (0 means transparent and 1 means opaque).

**Value**

This function creates a vector of “colors” corresponding to the given values in HSV space. The values returned by `hsv` can be used with a `col=` specification in graphics functions or in `par`.

**Gamma correction**

For each color,  $(r, g, b)$  in RGB space (with all values in  $[0, 1]$ ), the final color corresponds to  $(r^\gamma, g^\gamma, b^\gamma)$ .

**See Also**

[rainbow](#), [rgb](#), [gray](#).

**Examples**

```
hsv(.5, .5, .5)

## Look at gamma effect:
n <- 20; y <- -sin(3*pi*((1:n)-1/2)/n)
op <- par(mfrow=c(3,2),mar=rep(1.5,4))
for(gamma in c(.4, .6, .8, 1, 1.2, 1.5))
  plot(y, axes = FALSE, frame.plot = TRUE,
```

```

xlab = "", ylab = "", pch = 21, cex = 30,
bg = rainbow(n, start=.85, end=.1, gamma = gamma),
main = paste("Red tones; gamma=", format(gamma))
par(op)

```

## Description

The implementation of Hershey vector fonts provides a large number of Japanese characters (Hiragana, Katakana, and Kanji).

## Details

Without keyboard support for typing Japanese characters, the only way to produce these characters is to use special escape sequences: see [Hershey](#).

For example, the Hiragana character for the sound "ka" is produced by `\\#J242b` and the Katakana character for this sound is produced by `\\#J252b`. The Kanji ideograph for "one" is produced by `\\#J306c` or `\\#N0001`.

The output from `demo(Japanese)` shows tables of the escape sequences for the available Japanese characters.

## References

<http://www.gnu.org/software/plotutils/plotutils.html>

## See Also

`demo(Japanese)`, [Hershey](#), [text](#), [contour](#)

## Examples

```

plot(1:9, type="n", axes=FALSE, frame=TRUE, ylab="",
     main= "example(Japanese)", xlab= "using Hershey fonts")
par(cex=3)
Vf <- c("serif", "plain")
text(4, 2, "\\#J2438\\#J2421\\#J2451\\#J2473", vfont = Vf)
text(4, 4, "\\#J2538\\#J2521\\#J2551\\#J2573", vfont = Vf)
text(4, 6, "\\#J467c\\#J4b5c", vfont = Vf)
text(4, 8, "Japan", vfont = Vf)
par(cex=1)
text(8, 2, "Hiragana")
text(8, 4, "Katakana")
text(8, 6, "Kanji")
text(8, 8, "English")

```

---

`palette`*Set or View the Graphics Palette*

---

### Description

View or manipulate the color palette which is used when a `col=` has a numeric index.

### Usage

```
palette(value)
```

### Arguments

`value`            an optional character vector.

### Details

If `value` has length 1, it is taken to be the name of a built in color palette. If `value` has length greater than 1 it is assumed to contain a description of the colors which are to make up the new palette (either by name or by RGB levels).

If `value` is omitted or has length 0, no change is made the current palette.

Currently, the only built-in palette is "default".

### Value

The palette which *was* in effect. This is `invisible` unless the argument is omitted.

### See Also

`colors` for the vector of built-in "named" colors; `hsv`, `gray`, `rainbow`, `terrain.colors`,... to construct colors;

`col2rgb` for translating colors to RGB 3-vectors.

### Examples

```
palette()                    # obtain the current palette
palette(rainbow(6))         # six color rainbow

(palette(gray(seq(0,.9,len=25)))) # gray scales; print old palette
matplot(outer(1:100,1:30), type='l', lty=1,lwd=2, col=1:30,
        main = "Gray Scales Palette",
        sub = "palette(gray(seq(0,.9,len=25)))")
palette("default")         # reset back to the default
```

**Description**

Create a vector of  $n$  “contiguous” colors.

**Usage**

```
rainbow(n, s = 1, v = 1, start = 0, end = max(1, n - 1)/n, gamma = 1)
heat.colors(n)
terrain.colors(n)
topo.colors(n)
cm.colors(n)
```

**Arguments**

<code>n</code>	the number of colors ( $\geq 1$ ) to be in the palette.
<code>s, v</code>	the “saturation” and “value” to be used to complete the HSV color descriptions.
<code>start</code>	the (corrected) hue in $[0,1]$ at which the rainbow begins.
<code>end</code>	the (corrected) hue in $[0,1]$ at which the rainbow ends.
<code>gamma</code>	the gamma correction, see argument <code>gamma</code> in <a href="#">hsv</a> .

**Details**

Conceptually, all of these functions actually use (parts of) a line cut out of the 3-dimensional color space, parametrized by [hsv](#)(`h, s, v, gamma`), where `gamma=1` for the `foo.colors` function, and hence, equispaced hues in RGB space tend to cluster at the red, green and blue primaries.

Some applications such as contouring require a palette of colors which do not “wrap around” to give a final color close to the starting one.

With `rainbow`, the parameters `start` and `end` can be used to specify particular subranges of hues. The following values can be used when generating such a subrange: red=0, yellow= $\frac{1}{6}$ , green= $\frac{2}{6}$ , cyan= $\frac{3}{6}$ , blue= $\frac{4}{6}$  and magenta= $\frac{5}{6}$ .

**Value**

A character vector, `cv`, of color names. This can be used either to create a user-defined color palette for subsequent graphics by [palette](#)(`cv`), a `col=` specification in graphics functions or in [par](#).

**See Also**

[colors](#), [palette](#), [hsv](#), [rgb](#), [gray](#) and [col2rgb](#) for translating to RGB numbers.

**Examples**

```

require(graphics)
# A Color Wheel
pie(rep(1,12), col=rainbow(12))

##----- Some palettes -----
demo.pal <-
  function(n, border = if (n<32) "light gray" else NA,
           main = paste("color palettes; n=",n),
           ch.col = c("rainbow(n, start=.7, end=.1)", "heat.colors(n)",
                     "terrain.colors(n)", "topo.colors(n)", "cm.colors(n)"))
  {
    nt <- length(ch.col)
    i <- 1:n; j <- n / nt; d <- j/6; dy <- 2*d
    plot(i,i+d, type="n", yaxt="n", ylab="", main=main)
    for (k in 1:nt) {
      rect(i-.5, (k-1)*j+ dy, i+.4, k*j,
           col = eval(parse(text=ch.col[k])), border = border)
      text(2*j, k * j +dy/4, ch.col[k])
    }
  }
n <- if(.Device == "postscript") 64 else 16
# Since for screen, larger n may give color allocation problem
demo.pal(n)

```

pdf

*PDF Graphics Device***Description**

pdf starts the graphics device driver for producing PDF graphics.

**Usage**

```

pdf(file = ifelse(onefile, "Rplots.pdf", "Rplot%03d.pdf"),
    width = 6, height = 6, onefile = TRUE, family = "Helvetica",
    title = "R Graphics Output", fonts = NULL, version = "1.1",
    encoding, bg, fg, pointsize)

```

**Arguments**

file	a character string giving the name of the file.
width, height	the width and height of the graphics region in inches.
onefile	logical: if true (the default) allow multiple figures in one file. If false, generate a file name containing the page number.

family	the font family to be used, one of "AvantGarde", "Bookman", "Courier", "Helvetica", "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" or "Times". <b>Note</b> the other specifications allowed for <code>postscript</code> are <b>not</b> available.
title	title string to embed in the file.
encoding	the name of an encoding file. Defaults to "ISOLatin1.enc" in the 'R_HOME/afm' directory, which is used if the path does not contain a path separator. An extension ".enc" can be omitted.
pointsize	the default point size to be used.
bg	the default background color to be used.
fg	the default foreground color to be used.
fonts	a character vector specifying device-independent R graphics font family names for fonts which will be included in the PDF file.
version	a string describing the PDF version that will be used to produce output.

## Details

`pdf()` opens the file `file` and the PDF commands needed to plot any graphics requested are sent to that file.

The `family` argument can be used to specify either a device-independent R graphics font family (see `postscriptFonts`) or a PDF-specific font family as the initial/default font for the device.

If a device-independent R graphics font family is specified (e.g., via `par(family=)` in the graphics package), the PDF device makes use of the PostScript font mappings to convert the R graphics font family to a PDF-specific font family description. R does *not* embed fonts in the PDF file though, so it is only possible to use mappings to the font families that are assumed to be available in a PDF viewer: "Times" or "Times New Roman", "Helvetica" or "Arial", "Courier", "Symbol", and "ZapfDingbats".

See `postscript` for details of encodings, as the internal code is shared between the drivers. The native PDF encoding is given in file 'PDFDoc.enc'.

`pdf` writes uncompressed PDF. It is primarily intended for producing PDF graphics for inclusion in other documents, and PDF-includers such as `pdftex` are usually able to handle compression.

At present the PDF is fairly simple, with each page being represented as a single stream. The R graphics model does not distinguish graphics objects at the level of the driver interface.

The `version` argument modifies the sort of PDF code that gets produced. At the moment this only concerns the production of transparent output. The version must be greater than 1.4 for transparent output to be produced. Specifying a lower version number may be useful if you want to produce PDF output that can be viewed on older PDF viewers.

## Note

Acrobat Reader does not use the fonts specified but rather emulates them from multiple-master fonts. This can be seen in imprecise centering of characters, for example the multiply and divide signs in Helvetica.

**See Also**

[postscriptFonts](#), [Devices](#), [postscript](#)

**Examples**

```
## Not run:
## Test function for encodings
TestChars <- function(encoding="ISOLatin1", ...)
{
  pdf(encoding=encoding, ...)
  par(pty="s")
  plot(c(-1,16), c(-1,16), type="n", xlab="", ylab="", xaxs="i", yaxs="i")
  title(paste("Centred chars in encoding", encoding))
  grid(17, 17, lty=1)
  for(i in c(32:255)) {
    x <- i
    y <- i
    points(x, y, pch=i)
  }
  dev.off()
}
## there will be many warnings.
TestChars("ISOLatin2")
## doesn't view properly in US-spec Acrobat 5.05, but gs7.04 works.
## Lots of characters are not centred.
## End(Not run)
```

---

pictex

*A PicTeX Graphics Driver*

---

**Description**

This function produces graphics suitable for inclusion in TeX and LaTeX documents.

**Usage**

```
pictex(file = "Rplots.tex", width = 5, height = 4, debug = FALSE,
       bg = "white", fg = "black")
```

**Arguments**

file	the file where output will appear.
width	The width of the plot in inches.
height	the height of the plot in inches.
debug	should debugging information be printed.
bg	the background color for the plot.
fg	the foreground color for the plot.

**Details**

This driver does not have any font metric information, so the use of `plotmath` is not supported.

Multiple plots will be placed as separate environments in the output file.

**Author(s)**

This driver was provided by Valerio Aimale ([valerio@svpop.com.dist.unige.it](mailto:valerio@svpop.com.dist.unige.it)) of the Department of Internal Medicine, University of Genoa, Italy.

**References**

Knuth, D. E. (1984) *The TeXbook*. Reading, MA: Addison-Wesley.

Lamport, L. (1994) *LATEX: A Document Preparation System*. Reading, MA: Addison-Wesley.

Goossens, M., Mittelbach, F. and Samarin, A. (1994) *The LATEX Companion*. Reading, MA: Addison-Wesley.

**See Also**

[postscript](#), [Devices](#).

**Examples**

```
pictex()
plot(1:11, (-5:5)^2, type='b', main="Simple Example Plot")
dev.off()
##-----
## Not run:
## LaTeX Example
\documentclass{article}
\usepackage{pictex}
\begin{document}
%...
\begin{figure}[h]
  \centerline{\input{Rplots.tex}}
  \caption{}
\end{figure}
%...
\end{document}

%-- TeX Example --
\input pictex
$$ \input Rplots.tex $$
## End(Not run)
##-----
unlink("Rplots.tex")
```

## Description

If the `text` argument to one of the text-drawing functions (`text`, `mtext`, `axis`) in R is an expression, the argument is interpreted as a mathematical expression and the output will be formatted according to TeX-like rules. Expressions can also be used for titles, subtitles and x- and y-axis labels (but not for axis labels on `persp` plots).

## Details

A mathematical expression must obey the normal rules of syntax for any R expression, but it is interpreted according to very different rules than for normal R expressions.

It is possible to produce many different mathematical symbols, generate sub- or superscripts, produce fractions, etc.

The output from `demo(plotmath)` includes several tables which show the available features. In these tables, the columns of grey text show sample R expressions, and the columns of black text show the resulting output.

The available features are also described in the tables below:

Syntax	Meaning
<code>x + y</code>	x plus y
<code>x - y</code>	x minus y
<code>x*y</code>	juxtapose x and y
<code>x/y</code>	x forwardslash y
<code>x %+-% y</code>	x plus or minus y
<code>x %/% y</code>	x divided by y
<code>x %*% y</code>	x times y
<code>x[i]</code>	x subscript i
<code>x^2</code>	x superscript 2
<code>paste(x, y, z)</code>	juxtapose x, y, and z
<code>sqrt(x)</code>	square root of x
<code>sqrt(x, y)</code>	yth root of x
<code>x == y</code>	x equals y
<code>x != y</code>	x is not equal to y
<code>x &lt; y</code>	x is less than y
<code>x &lt;= y</code>	x is less than or equal to y
<code>x &gt; y</code>	x is greater than y
<code>x &gt;= y</code>	x is greater than or equal to y
<code>x %~~% y</code>	x is approximately equal to y
<code>x %~% y</code>	x and y are congruent
<code>x %==% y</code>	x is defined as y
<code>x %prop% y</code>	x is proportional to y
<code>plain(x)</code>	draw x in normal font
<code>bold(x)</code>	draw x in bold font

<code>italic(x)</code>	draw $x$ in italic font
<code>bolditalic(x)</code>	draw $x$ in bolditalic font
<code>list(x, y, z)</code>	comma-separated list
<code>...</code>	ellipsis (height varies)
<code>cdots</code>	ellipsis (vertically centred)
<code>ldots</code>	ellipsis (at baseline)
<code>x %subset% y</code>	$x$ is a proper subset of $y$
<code>x %subsepeq% y</code>	$x$ is a subset of $y$
<code>x %notsubset% y</code>	$x$ is not a subset of $y$
<code>x %supset% y</code>	$x$ is a proper superset of $y$
<code>x %supsepeq% y</code>	$x$ is a superset of $y$
<code>x %in% y</code>	$x$ is an element of $y$
<code>x %notin% y</code>	$x$ is not an element of $y$
<code>hat(x)</code>	$x$ with a circumflex
<code>tilde(x)</code>	$x$ with a tilde
<code>dot(x)</code>	$x$ with a dot
<code>ring(x)</code>	$x$ with a ring
<code>bar(xy)</code>	$xy$ with bar
<code>widehat(xy)</code>	$xy$ with a wide circumflex
<code>widetilde(xy)</code>	$xy$ with a wide tilde
<code>x %&lt;-&gt;% y</code>	$x$ double-arrow $y$
<code>x %-&gt;% y</code>	$x$ right-arrow $y$
<code>x %&lt;-% y</code>	$x$ left-arrow $y$
<code>x %up% y</code>	$x$ up-arrow $y$
<code>x %down% y</code>	$x$ down-arrow $y$
<code>x %&lt;=&gt;% y</code>	$x$ is equivalent to $y$
<code>x %=&gt;% y</code>	$x$ implies $y$
<code>x %&lt;=% y</code>	$y$ implies $x$
<code>x %dblup% y</code>	$x$ double-up-arrow $y$
<code>x %dbldown% y</code>	$x$ double-down-arrow $y$
<code>alpha - omega</code>	Greek symbols
<code>Alpha - Omega</code>	uppercase Greek symbols
<code>infinity</code>	infinity symbol
<code>partialdiff</code>	partial differential symbol
<code>32*degree</code>	32 degrees
<code>60*minute</code>	60 minutes of angle
<code>30*second</code>	30 seconds of angle
<code>displaystyle(x)</code>	draw $x$ in normal size (extra spacing)
<code>textstyle(x)</code>	draw $x$ in normal size
<code>scriptstyle(x)</code>	draw $x$ in small size
<code>scriptscriptstyle(x)</code>	draw $x$ in very small size
<code>x ~~ y</code>	put extra space between $x$ and $y$
<code>x + phantom(0) + y</code>	leave gap for "0", but don't draw it
<code>x + over(1, phantom(0))</code>	leave vertical gap for "0" (don't draw)
<code>frac(x, y)</code>	$x$ over $y$
<code>over(x, y)</code>	$x$ over $y$
<code>atop(x, y)</code>	$x$ over $y$ (no horizontal bar)
<code>sum(x[i], i=1, n)</code>	sum $x[i]$ for $i$ equals 1 to $n$

<code>prod(plain(P)(X==x), x)</code>	product of $P(X=x)$ for all values of $x$
<code>integral(f(x)*dx, a, b)</code>	definite integral of $f(x)$ wrt $x$
<code>union(A[i], i==1, n)</code>	union of $A[i]$ for $i$ equals 1 to $n$
<code>intersect(A[i], i==1, n)</code>	intersection of $A[i]$
<code>lim(f(x), x %&gt;% 0)</code>	limit of $f(x)$ as $x$ tends to 0
<code>min(g(x), x &gt; 0)</code>	minimum of $g(x)$ for $x$ greater than 0
<code>inf(S)</code>	infimum of $S$
<code>sup(S)</code>	supremum of $S$
<code>x^y + z</code>	normal operator precedence
<code>x^(y + z)</code>	visible grouping of operands
<code>x^{y + z}</code>	invisible grouping of operands
<code>group("(",list(a, b),")")</code>	specify left and right delimiters
<code>bgroup("(",atop(x,y),")")</code>	use scalable delimiters
<code>group(lceil, x, rceil)</code>	special delimiters

## References

Murrell, P. and Ihaka, R. (2000) An approach to providing mathematical annotation in plots. *Journal of Computational and Graphical Statistics*, **9**, 582–599.

## See Also

`demo(plotmath)`, [axis](#), [mtext](#), [text](#), [title](#), [substitute quote](#), [bquote](#)

## Examples

```
x <- seq(-4, 4, len = 101)
y <- cbind(sin(x), cos(x))
matplot(x, y, type = "l", xaxt = "n",
        main = expression(paste(plain(sin) * phi, " and ",
                                plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
        col.main = "blue")
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     lab = expression(-pi, -pi/2, 0, pi/2, pi))

## How to combine "math" and numeric variables :
plot(1:10, type="n", xlab="", ylab="", main = "plot math & numbers")
theta <- 1.23 ; mtext(bquote(hat(theta) == .(theta)))
for(i in 2:9)
  text(i,i+1, substitute(list(xi,eta) == group("(",list(x,y),")"),
                        list(x=i, y=i+1)))

plot(1:10, 1:10)
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)",
     cex = .8)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i==1, n)))
text(4, 6.4, "expression(bar(x) == sum(frac(x[i], n), i==1, n))",
     cex = .8)
```

```
text(8, 5, expression(paste(frac(1, sigma*sqrt(2*pi)), " ",
                             plain(e)^{frac(-(x-mu)^2, 2*sigma^2)})),
     cex = 1.2)
```

---

 png

*JPEG and PNG graphics devices*


---

### Description

A graphics device for JPEG or PNG format bitmap files.

### Usage

```
jpeg(filename = "Rplot%03d.jpeg", width = 480, height = 480,
      pointsize = 12, quality = 75, bg = "white", res = NA, ...)

png(filename = "Rplot%03d.png", width = 480, height = 480,
     pointsize = 12, bg = "white", res = NA, ...)
```

### Arguments

filename	the name of the output file. The page number is substituted if an integer format is included in the character string. (The result must be less than <code>PATH_MAX</code> characters long, and may be truncated if not.) Tilde expansion is performed where supported by the platform.
width	the width of the device in pixels.
height	the height of the device in pixels.
pointsize	the default pointsize of plotted text.
quality	the ‘quality’ of the JPEG image, as a percentage. Smaller values will give more compression but also more degradation of the image.
bg	default background colour.
res	The nominal resolution in dpi which will be recorded in the bitmap file, if a positive integer.
...	additional arguments to the <a href="#">X11</a> device.

### Details

Plots in PNG and JPEG format can easily be converted to many other bitmap formats, and both can be displayed in most modern web browsers. The PNG format is lossless and is best for line diagrams and blocks of solid colour. The JPEG format is lossy, but may be useful for image plots, for example.

png supports transparent backgrounds: use `bg = "transparent"`. Not all PNG viewers render files with transparency correctly. When transparency is in use a very light grey is used as the background and so will appear as transparent if used in the plot. This allows opaque white to be used, as on the example.

R can be compiled without support for either or both of these devices: this will be reported if you attempt to use them on a system where they are not supported. They will not be available if R has been started with `--gui=none` (and will give a different error message), and they may not be usable unless the X11 display is available to the owner of the R process.

By default no resolution is recorded in the file. Readers will often assume nominal resolution of 72dpi when none is recorded. As resolutions in PNG files are recorded in pixels/metre, the dpi value will be changed slightly.

### Value

A plot device is opened: nothing is returned to the R interpreter.

### Warning

If you plot more than one page on one of these devices and do not include something like `%d` for the sequence number in `file`, the file will contain the last page plotted.

### Note

These are based on the [X11](#) device, so the additional arguments to that device work, but are rarely appropriate. The colour handling will be that of the X11 device in use.

### Author(s)

Guido Masarotto and Brian Ripley

### See Also

[Devices](#), [dev.print](#)

[capabilities](#) to see if these devices are supported by this build of R.

[bitmap](#) provides an alternative way to generate PNG and JPEG plots that does not depend on accessing the X11 display but does depend on having GhostScript installed.

### Examples

```
## these examples will work only if the devices are available
## and the X11 display is available.

## copy current plot to a PNG file
## Not run: dev.print(png, file="myplot.png", width=480, height=480)

png(file="myplot.png", bg="transparent")
plot(1:10)
rect(1, 5, 3, 7, col="white")
dev.off()

jpeg(file="myplot.jpeg")
example(rect)
dev.off()
## End(Not run)
```

---

 postscript

*PostScript Graphics*


---

### Description

postscript starts the graphics device driver for producing PostScript graphics.

The auxiliary function `ps.options` can be used to set and view (if called without arguments) default values for the arguments to `postscript`.

### Usage

```
postscript(file = ifelse(onefile, "Rplots.ps", "Rplot%03d.ps"),
           onefile = TRUE,
           paper, family, encoding, bg, fg,
           width, height, horizontal, pointsize,
           pagecentre, print.it, command,
           title = "R Graphics Output", fonts = NULL)
```

```
ps.options(paper, horizontal, width, height, family, encoding,
           pointsize, bg, fg,
           onefile = TRUE, print.it = FALSE, append = FALSE,
           reset = FALSE, override.check = FALSE)
```

### Arguments

file	a character string giving the name of the file. If it is "", the output is piped to the command given by the argument <code>command</code> . If it is " <code> cmd</code> ", the output is piped to the command given by ' <code>cmd</code> '. For use with <code>onefile=FALSE</code> give a <code>printf</code> format such as " <code>Rplot%03d.ps</code> " (the default in that case).
paper	the size of paper in the printer. The choices are "a4", "letter", "legal" and "executive" (and these can be capitalized). Also, "special" can be used, when the <code>width</code> and <code>height</code> specify the paper size. A further choice is "default", which is the default. If this is selected, the <code>papersize</code> is taken from the option " <code>papersize</code> " if that is set and to "a4" if it is unset or empty.
horizontal	the orientation of the printed image, a logical. Defaults to true, that is landscape orientation on paper sizes with width less than height.
width, height	the width and height of the graphics region in inches. The default is to use the entire page less a 0.25 inch border on each side.
family	the font family to be used. EITHER a single character string OR a character vector of length four or five. See the section 'Families'.
encoding	the name of an encoding file. Defaults to "ISOLatin1.enc" in the ' <code>R_HOME/afm</code> ' directory, which is used if the path does not contain a path separator. An extension ".enc" can be omitted.

<code>pointsize</code>	the default point size to be used.
<code>bg</code>	the default background color to be used. If "transparent" (or an equivalent specification), no background is painted.
<code>fg</code>	the default foreground color to be used.
<code>onefile</code>	logical: if true (the default) allow multiple figures in one file. If false, generate a file name containing the page number and use an EPSF header and no DocumentMedia comment.
<code>pagecentre</code>	logical: should the device region be centred on the page: defaults to true.
<code>print.it</code>	logical: should the file be printed when the device is closed? (This only applies if file is a real file name.)
<code>command</code>	the command to be used for "printing". Defaults to option "printcmd"; this can also be selected as "default".
<code>append</code>	logical; currently <b>disregarded</b> ; just there for compatibility reasons.
<code>reset, override.check</code>	logical arguments passed to <code>check.options</code> . See the Examples.
<code>title</code>	title string to embed in the file.
<code>fonts</code>	a character vector specifying R graphics (device-independent) font family names for fonts which must be included in the PostScript file.

## Details

`postscript` opens the file `file` and the PostScript commands needed to plot any graphics requested are stored in that file. This file can then be printed on a suitable device to obtain hard copy.

A postscript plot can be printed via `postscript` in two ways.

1. Setting `print.it = TRUE` causes the command given in argument `command` to be called with argument "file" when the device is closed. Note that the plot file is not deleted unless `command` arranges to delete it.
2. `file=""` or `file="|cmd"` can be used to print using a pipe on systems that support 'popen'. Failure to open the command will probably be reported to the terminal but not to 'popen', in which case close the device by `dev.off` immediately.

The postscript produced by R is EPS (*Encapsulated PostScript*) compatible, and can be included into other documents, e.g., into LaTeX, using `includegraphics{<filename>}`. For use in this way you will probably want to set `horizontal = FALSE, onefile = FALSE, paper = "special"`.

Most of the PostScript prologue used is taken from the R character vector `.ps.prolog`. This is marked in the output, and can be changed by changing that vector. (This is only advisable for PostScript experts: the standard version is in namespace:grDevices.)

`ps.options` needs to be called before calling `postscript`, and the default values it sets can be overridden by supplying arguments to `postscript`.

A PostScript device has a default font, which can be set by the user via `family`. If other fonts are to be used when drawing to the PostScript device, these must be declared when the device is created via `fonts`; the font family names for this argument are device-independent R graphics font family names (see the documentation for `postscriptFonts`).

## Families

Font families may be specified in several ways. The `family` argument specifies an initial/default font family for the device. This may be a device-independent R graphics font family (see `postscriptFonts`) or a PostScript-specific font family specification (see below). The `fonts` argument specifies a set of device-independent font families that are mapped to PostScript-specific fonts via a font database (see `postscriptFonts`).

The argument `family` specifies the initial/default font family to be used. In normal use it is one of "AvantGarde", "Bookman", "Courier", "Helvetica", "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" or "Times", and refers to the standard Adobe PostScript fonts of those names which are included (or cloned) in all common PostScript devices.

Many PostScript emulators (including those based on `ghostscript`) use the URW equivalents of these fonts, which are "URWGothic", "URWBookman", "NimbusMon", "NimbusSan", "NimbusSanCond", "CenturySch", "URWPalladio" and "NimbusRom" respectively. If your PostScript device is using URW fonts, you will obtain access to more characters and more appropriate metrics by using these names. To make these easier to remember, "URWHelvetica" == "NimbusSan" and "URWTimes" == "NimbusRom" are also supported.

It is also possible to specify `family = "ComputerModern"`. This is intended to use with the Type 1 versions of the TeX CM fonts. It will normally be possible to include such output in TeX or LaTeX provided it is processed with `dvips -Ppfb -j0` or the equivalent on your system. (`-j0` turns off font subsetting.)

If the second form of argument "family" is used, it should be a character vector of four or five paths to Adobe Font Metric files for the regular, bold, italic, bold italic and (optionally) symbol fonts to be used. If these paths do not contain the file separator, they are taken to refer to files in the R directory 'R\_HOME/afm'. Thus the default Helvetica family can be specified by `family = c("hv____.afm", "hvb____.afm", "hvo____.afm", "hvbo____.afm", "sy____.afm")`. It is the user's responsibility to check that suitable fonts are made available, and that they contain the needed characters when re-encoded. The fontnames used are taken from the `FontName` fields of the afm files. The software including the PostScript plot file should either embed the font outlines (usually from '.pfb' or '.pfa' files) or use DSC comments to instruct the print spooler to do so.

The .afm files for the first four fonts do not need to be in the correct encoding, but that for the symbol font must be.

When `family = "ComputerModern"` is used, the italic/bold-italic fonts used are slanted fonts (`cmsl10` and `cmbxsl10`). To use text italic fonts instead, use `family = c("CM_regular_10.afm", "CM_boldx_10.afm", "cmti10.afm", "cmbxti10.afm", "CM_symbol_10.afm")`.

## Encodings

Encodings describe which glyphs are used to display the character codes (in the range 0–255). By default R uses ISOLatin1 encoding, and the examples for `text` are in that encoding. However, the encoding used on machines running R may well be different, and by using the `encoding` argument the glyphs can be matched to encoding in use.

None of this will matter if only ASCII characters (codes 32–126) are used as all the encodings agree over that range. Some encodings are supersets of ISOLatin1, too. However, if accented and special characters do not come out as you expect, you may need to change the encoding. Three

other encodings are supplied with R: "WinAnsi.enc" and "MacRoman.enc" correspond to the encodings normally used on Windows and MacOS (at least by Adobe), and "PDFDoc.enc" is the first 256 characters of the Unicode encoding, the standard for PDF.

If you change the encoding, it is your responsibility to ensure that the PostScript font contains the glyphs used. One issue here is the Euro symbol which is in the WinAnsi and MacRoman encodings but may well not be in the PostScript fonts. (It is in the URW variants; it is not in the supplied Adobe Font Metric files.)

There is one exception. Character 45 ("-") is always set as minus (its value in Adobe ISOLatin1) even though it is hyphen in the other encodings. Hyphen is available as character 173 (octal 0255) in ISOLatin1.

### Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso (<durso@hussle.harvard.edu>).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[postscriptFonts](#), [Devices](#), [check.options](#) which is called from both `ps.options` and `postscript`.

### Examples

```
## Not run:
# open the file "foo.ps" for graphics output
postscript("foo.ps")
# produce the desired graph(s)
dev.off() # turn off the postscript device
postscript("|lp -dlw")
# produce the desired graph(s)
dev.off() # plot will appear on printer

# for URW PostScript devices
postscript("foo.ps", family = "NimbusSan")

## for inclusion in Computer Modern TeX documents, perhaps
postscript("cm_test.eps", width = 4.0, height = 3.0,
          horizontal = FALSE, onefile = FALSE, paper = "special",
          family = "ComputerModern")
## The resultant postscript file can be used by dvips -Ppfb -j0.

## To test out encodings, you can use
TestChars <- function(encoding="ISOLatin1", family="URWHelvetica")
{
  postscript(encoding=encoding, family=family)
  par(pty="s")
}
```

```

plot(c(-1,16), c(-1,16), type="n", xlab="", ylab="", xaxs="i", yaxs="i")
title(paste("Centred chars in encoding", encoding))
grid(17, 17, lty=1)
for(i in c(32:255)) {
  x <- i
  y <- i
  points(x, y, pch=i)
}
dev.off()
}
## there will be many warnings. We use URW to get a complete enough
## set of font metrics.
TestChars()
TestChars("ISOLatin2")
TestChars("WinAnsi")
## End(Not run)

ps.options(bg = "pink")
utils::str(ps.options(reset = TRUE))

### ---- error checking of arguments: ----
ps.options(width=0:12, onefile=0, bg=pi)
# override the check for 'onefile', but not the others:
utils::str(ps.options(width=0:12, onefile=1, bg=pi,
                      override.check = c(FALSE,TRUE,FALSE)))

```

---

postscriptFonts      *PostScript Fonts*

---

### Description

These functions handle the translation of a device-independent R graphics font family name to a PostScript font description.

### Usage

```

postscriptFont(family, metrics, encoding = "default")

postscriptFonts(...)

```

### Arguments

family	a character string giving the name of an Adobe Type 1 font family.
metrics	a vector of four or five strings giving paths to the afm (font metric) files for the Type 1 font.
encoding	the name of an encoding file. Defaults to "ISOLatin1.enc" in the 'R_HOME/afm' directory, which is used if the path does not contain a path separator. An extension ".enc" can be omitted.

... either character strings naming mappings to display, or new (named) mappings to define.

### Details

A PostScript device is created with a default font (see the documentation for `postscript`), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for `gpar` in the `grid` package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to PostScript fonts. A list of mappings is maintained and can be modified by the user.

The `postscriptFonts` function can be used to list existing translations and to define new mappings. The `postscriptFont` function can be used to create a new mapping.

The argument `family` specifies the font family to be used. Default mappings are provided for four device-independent family names: `"sans"` for a sans-serif font, `"serif"` for a serif font, `"mono"` for a monospaced font, and `"symbol"` for a symbol font.

Mappings for a number of standard Adobe fonts (and URW equivalents) are also provided: `"AvantGarde"`, `"Bookman"`, `"Courier"`, `"Helvetica"`, `"Helvetica-Narrow"`, `"NewCenturySchoolbook"`, `"Palatino"` and `"Times"`; `"URWGothic"`, `"URWBookman"`, `"NimbusMon"`, `"NimbusSan"`, `"NimbusSanCond"`, `"CenturySch"`, `"URWPalladio"` and `"NimbusRom"`.

There is also a mapping for `"ComputerModern"`.

The specification of font metrics and encodings is described in the help for the `postscript` function.

### Value

`postscriptFont` returns a PostScript font description.

`postscriptFonts` returns one or more font mappings.

### Author(s)

Support for Computer Modern fonts is based on a contribution by Brian D'Urso ([durso@hussle.harvard.edu](mailto:durso@hussle.harvard.edu)).

### See Also

[postscript](#)

### Examples

```
postscriptFonts()
CMitalic <- postscriptFont("ComputerModern",
                          c("CM_regular_10.afm", "CM_boldx_10.afm",
                            "cmti10.afm", "cmbxti10.afm",
                            "CM_symbol_10.afm"))
postscriptFonts(CMitalic=CMitalic)
```

---

`quartz`*MacOS X Quartz device*

---

**Description**

`quartz` starts a graphics device driver for the MacOS X System. This can only be done on machines that run MacOS X.

**Usage**

```
quartz(display = "", width = 5, height = 5, pointsize = 12,  
        family = "Helvetica", antialias = TRUE, autorefresh = TRUE)
```

**Arguments**

<code>display</code>	the display on which the graphics window will appear. The default is to use the value in the user's environment variable <code>DISPLAY</code> .
<code>width</code>	the width of the plotting window in inches.
<code>height</code>	the height of the plotting window in inches.
<code>pointsize</code>	the default pointsize to be used.
<code>family</code>	this is the family name of the Postscript font that will be used by the device.
<code>antialias</code>	whether to use antialiasing. It is never the case to set it <code>FALSE</code>
<code>autorefresh</code>	logical specifying if realtime refreshing should be done. If <code>FALSE</code> , the system is charged to refresh the context of the device window.

**Details**

Quartz is the graphic engine based on the PDF format. It is used by the graphic interface of MacOS X to render high quality graphics. As PDF it is device independent and can be rescaled without loss of definition.

If a device-independent R graphics font family is specified (e.g., via `par(family=)` in the graphics package), the Quartz device makes use of the Quartz font database (see `quartzFonts`) to convert the R graphics font family to a Quartz-specific font family description.

Calling `quartz()` sets `.Device` to "quartz".

**See Also**

[quartzFonts](#), [Devices](#).

---

`quartzFonts`*quartz Fonts*

---

### Description

These functions handle the translation of a device-independent R graphics font family name to a quartz font description.

### Usage

```
quartzFont(family)
```

```
quartzFonts(...)
```

### Arguments

<code>family</code>	a character vector containing the four PostScript font names for plain, bold, italic, and bolditalic versions of a font family.
<code>...</code>	either character strings naming mappings to display, or new (named) mappings to define.

### Details

A quartz device is created with a default font (see the documentation for `quartz`), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for `gpar` in the `grid` package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to quartz fonts. A list of mappings is maintained and can be modified by the user.

The `quartzFonts` function can be used to list existing mappings and to define new mappings. The `quartzFont` function can be used to create a new mapping.

Default mappings are provided for four device-independent font family names: "sans" for a sans-serif font, "serif" for a serif font, "mono" for a monospaced font, and "symbol" for a symbol font.

### See Also

[quartz](#)

### Examples

```
quartzFonts()  
quartzFonts("mono")
```

---

recordPlot	<i>Record and Replay Plots</i>
------------	--------------------------------

---

### Description

Functions to save the current plot in an R variable, and to replay it.

### Usage

```
recordPlot()  
replayPlot(x)
```

### Arguments

x                    A saved plot.

### Details

These functions record and replay the displaylist of the current graphics device. The returned object is of class "recordedplot", and `replayPlot` acts as a `print` method for that class.

### Value

`recordPlot` returns an object of class "recordedplot".  
`replayPlot` has no return value.

### WARNING

The format of recorded plots may change between R versions. Recorded plots should **not** be used as a permanent storage format for R plots.

R will always attempt to replay a recorded plot, but if the plot was recorded with a different R version then bad things may happen.

---

rgb	<i>RGB Color Specification</i>
-----	--------------------------------

---

### Description

This function creates "colors" corresponding to the given intensities (between 0 and `max`) of the red, green and blue primaries.

An alpha transparency value can also be specified (0 means fully transparent and `max` means opaque). If `alpha` is not specified, an opaque colour is generated.

The `names` argument may be used to provide names for the colors.

The values returned by these functions can be used with a `col=` specification in graphics functions or in `par`.

**Usage**

```
rgb(red, green, blue, alpha, names = NULL, maxColorValue = 1)
```

**Arguments**

`red`, `blue`, `green`, `alpha`  
 vectors of same length with values in  $[0, M]$  where  $M$  is `maxColorValue`.  
 When this is 255, the `red`, `blue`, `green`, and `alpha` values are coerced to integers in  $0 : 255$  and the result is computed most efficiently.

`names`  
 character. The names for the resulting vector.

`maxColorValue`  
 number giving the maximum of the color values range, see above.

**See Also**

[col2rgb](#) the “inverse” for translating R colors to RGB vectors; [rainbow](#), [hsv](#), [gray](#).

**Examples**

```
rgb(0,1,0)
(u01 <- seq(0,1, length=11))
stopifnot(rgb(u01,u01,u01) == gray(u01))
reds <- rgb((0:15)/15, g=0,b=0, names=paste("red",0:15,sep="."))
reds

rgb(0, 0:12, 0, max = 255)# integer input
```

---

rgb2hsv

*RGB to HSV Conversion*

---

**Description**

`rgb2hsv` transforms colors from RGB space (red/green/blue) into HSV space (hue/saturation/value).

**Usage**

```
rgb2hsv(r, g = NULL, b = NULL, gamma = 1, maxColorValue = 255)
```

**Arguments**

`r`  
 vector of “red” values in  $[0, M]$ , ( $M = \text{maxColorValue}$ ) or 3-row `rgb` matrix.

`g`  
 vector of “green” values, or `NULL` when `r` is a matrix.

`b`  
 vector of “blue” values, or `NULL` when `r` is a matrix.

`gamma`  
 a “gamma correction” (supposedly applied to the `r,g,b` values previously), see [hsv\(..., gamma\)](#).

`maxColorValue`  
 number giving the maximum of the RGB color values range. The default 255 corresponds to the typical  $0 : 255$  RGB coding as in [col2rgb\(\)](#).

**Details**

Value (brightness) gives the amount of light in the color.  
 Hue describes the dominant wavelength.  
 Saturation is the amount of Hue mixed into the color.

**Value**

A matrix with a column for each color. The three rows of the matrix indicate hue, saturation and value and are named "h", "s", and "v" accordingly.

**Author(s)**

R interface by Wolfram Fischer <wolfram@fischer-zim.ch>;  
 C code mainly by Nicholas Lewin-Koh <nikko@hailmail.net>.

**See Also**

[hsv](#), [col2rgb](#), [rgb](#).

**Examples**

```
## These (saturated, bright ones) only differ by hue
(rc <- col2rgb(c("red", "yellow", "green", "cyan", "blue", "magenta")))
(hc <- rgb2hsv(rc))
6 * hc["h",] # the hues are equispaced

(rgb3 <- floor(256 * matrix(runif(3*12), 3,12)))
(hsv3 <- rgb2hsv(rgb3))
## Consistency :
stopifnot(rgb3 == col2rgb(hsv(h=hsv3[1,], s=hsv3[2,], v=hsv3[3,])),
           all.equal(hsv3, rgb2hsv(rgb3/255, maxC = 1)))

## A (simplified) pure R version -- originally by Wolfram Fischer --
## showing the exact algorithm:
rgb2hsvR <- function(rgb, gamma = 1, maxColorValue = 255)
{
  if(!is.numeric(rgb)) stop("rgb matrix must be numeric")
  d <- dim(rgb)
  if(d[1] != 3) stop("rgb matrix must have 3 rows")
  n <- d[2]
  if(n == 0) return(cbind(c(h=1,s=1,v=1))[,0])
  rgb <- rgb/maxColorValue
  if(gamma != 1) rgb <- rgb ^ (1/gamma)

  ## get the max and min
  v <- apply( rgb, 2, max)
  s <- apply( rgb, 2, min)
  D <- v - s # range

  ## set hue to zero for undefined values (gray has no hue)
```

```

h <- numeric(n)
notgray <- ( s != v )

## blue hue
idx <- (v == rgb[3,] & notgray )
if (any (idx))
  h[idx] <- 2/3 + 1/6 * (rgb[1,idx] - rgb[2,idx]) / D[idx]
## green hue
idx <- (v == rgb[2,] & notgray )
if (any (idx))
  h[idx] <- 1/3 + 1/6 * (rgb[3,idx] - rgb[1,idx]) / D[idx]
## red hue
idx <- (v == rgb[1,] & notgray )
if (any (idx))
  h[idx] <- 1/6 * (rgb[2,idx] - rgb[3,idx]) / D[idx]

## correct for negative red
idx <- (h < 0)
h[idx] <- 1+h[idx]

## set the saturation
s[! notgray] <- 0;
s[notgray] <- 1 - s[notgray] / v[notgray]

rbind( h=h, s=s, v=v )
}

## confirm the equivalence:
all.equal(rgb2hsv (rgb3),
          rgb2hsvR(rgb3), tol=1e-14) # TRUE

```

---

units

*Graphical Units*


---

### Description

`xinch` and `yinch` convert the specified number of inches given as their arguments into the correct units for plotting with graphics functions. Usually, this only makes sense when normal coordinates are used, i.e., *no* log scale (see the `log` argument to `par`).

`xyinch` does the same for a pair of numbers `xy`, simultaneously.

`cm` translates inches in to cm (centimeters).

### Usage

```

xinch(x = 1, warn.log = TRUE)
yinch(y = 1, warn.log = TRUE)
xyinch(xy = 1, warn.log = TRUE)
cm(x)

```

**Arguments**

<code>x,y</code>	numeric vector
<code>xy</code>	numeric of length 1 or 2.
<code>warn.log</code>	logical; if TRUE, a warning is printed in case of active log scale.

**Examples**

```

all(c(xinch(),yinch()) == xyinch()) # TRUE
xyinch()
xyinch #- to see that is really    delta{"usr"} / "pin"

cm(1)# = 2.54

## plot labels offset 0.12 inches to the right
## of plotted symbols in a plot
with(mtcars, {
  plot(mpg, disp, pch=19, main= "Motor Trend Cars")
  text(mpg + xinch(0.12), disp, row.names(mtcars),
       adj = 0, cex = .7, col = 'blue')
})

```

x11

*X Window System Graphics***Description**

X11 starts a graphics device driver for the X Window System (version 11). This can only be done on machines that run X. x11 is recognized as a synonym for X11.

**Usage**

```

X11(display = "", width = 7, height = 7, pointsize = 12,
     gamma = 1, colortype = getOption("X11colortype"),
     maxcubsize = 256, bg = "transparent", canvas = "white",
     fonts = getOption("X11fonts"))

```

**Arguments**

<code>display</code>	the display on which the graphics window will appear. The default is to use the value in the user's environment variable DISPLAY.
<code>width</code>	the width of the plotting window in inches.
<code>height</code>	the height of the plotting window in inches.
<code>pointsize</code>	the default pointsize to be used.
<code>gamma</code>	the gamma correction factor. This value is used to ensure that the colors displayed are linearly related to RGB values. A value of around 0.5 is appropriate for many PC displays. A value of 1.0 (no correction) is usually appropriate for high-end displays or Macintoshes.

<code>colortype</code>	the kind of color model to be used. The possibilities are "mono", "gray", "pseudo", "pseudo.cube" and "true". Ignored if an X11 is already open.
<code>maxcubesize</code>	can be used to limit the size of color cube allocated for pseudocolor devices.
<code>bg</code>	color. The default background color.
<code>canvas</code>	color. The color of the canvas, which is visible only when the background color is transparent.
<code>fonts</code>	X11 font description strings into which weight, slant and size will be substituted. There are two, the first for fonts 1 to 4 and the second for font 5, the symbol font.

### Details

By default, an X11 device will use the best color rendering strategy that it can. The choice can be overridden with the `colortype` parameter. A value of "mono" results in black and white graphics, "gray" in grayscale and "true" in truecolor graphics (if this is possible). The values "pseudo" and "pseudo.cube" provide color strategies for pseudocolor displays. The first strategy provides on-demand color allocation which produces exact colors until the color resources of the display are exhausted. The second causes a standard color cube to be set up, and requested colors are approximated by the closest value in the cube. The default strategy for pseudocolor displays is "pseudo".

An initial/default font family for the device can be specified via the `fonts` argument, but if a device-independent R graphics font family is specified (e.g., via `par(family=)` in the graphics package), the X11 device makes use of the X11 font database (see `X11Fonts`) to convert the R graphics font family to an X11-specific font family description.

**Note:** All X11 devices share a `colortype` which is set by the first device to be opened. To change the `colortype` you need to close *all* open X11 devices then open one with the desired `colortype`.

With `colortype` equal to "pseudo.cube" or "gray" successively smaller palettes are tried until one is completely allocated. If allocation of the smallest attempt fails the device will revert to "mono".

### See Also

[Devices](#).

---

X11Fonts

*X11 Fonts*

---

### Description

These functions handle the translation of a device-independent R graphics font family name to an X11 font description.

**Usage**

```
X11Font ( font )
```

```
X11Fonts ( ... )
```

**Arguments**

font	a character string containing an X11 font description.
...	either character strings naming mappings to display, or new (named) mappings to define.

**Details**

An X11 device is created with a default font (see the documentation for X11), but it is also possible to specify a font family when drawing to the device (for example, see the documentation for `gpar` in the `grid` package).

The font family sent to the device is a simple string name, which must be mapped to something more specific to X11 fonts. A list of mappings is maintained and can be modified by the user.

The `X11Fonts` function can be used to list existing mappings and to define new mappings. The `X11Font` function can be used to create a new mapping.

Default mappings are provided for four device-independent font family names: "sans" for a sans-serif font, "serif" for a serif font, "mono" for a monospaced font, and "symbol" for a symbol font.

**See Also**

[X11](#)

**Examples**

```
X11Fonts()
X11Fonts("mono")
utopia <- X11Font("-*-utopia-*-*-*-*-*-*-*-*-*-*")
X11Fonts(utopia=utopia)
```

---

xfig

*XFig Graphics Device*

---

**Description**

`xfig` starts the graphics device driver for producing XFig (version 3.2) graphics.

The auxiliary function `ps.options` can be used to set and view (if called without arguments) default values for the arguments to `xfig` and `postscript`.

**Usage**

```
xfig(file = ifelse(onefile, "Rplots.fig", "Rplot%03d.fig"),
      onefile = FALSE, ...)
```

**Arguments**

<code>file</code>	a character string giving the name of the file. If it is "", the output is piped to the command given by the argument <code>command</code> . For use with <code>onefile = FALSE</code> give a <code>printf</code> format such as "Rplot%d.fig" (the default in that case).
<code>onefile</code>	logical: if true allow multiple figures in one file. If false, assume only one page per file and generate a file number containing the page number.
<code>...</code>	further arguments to <code>ps.options</code> accepted by <code>xfig()</code> : <ul style="list-style-type: none"> <li><b>paper</b> the size of paper in the printer. The choices are "A4", "Letter" and "Legal" (and these can be lowercase). A further choice is "default", which is the default. If this is selected, the <code>papersize</code> is taken from the option "papersize" if that is set and to "A4" if it is unset or empty.</li> <li><b>horizontal</b> the orientation of the printed image, a logical. Defaults to true, that is landscape orientation.</li> <li><b>width,height</b> the width and height of the graphics region in inches. The default is to use the entire page less a 0.25 inch border.</li> <li><b>family</b> the font family to be used. This must be one of "AvantGarde", "Bookman", "Courier", "Helvetica", "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" or "Times".</li> <li><b>pointsize</b> the default point size to be used.</li> <li><b>bg</b> the default background color to be used.</li> <li><b>fg</b> the default foreground color to be used.</li> <li><b>pagecentre</b> logical: should the device region be centred on the page: defaults to TRUE.</li> </ul>

**Details**

Although `xfig` can produce multiple plots in one file, the XFig format does not say how to separate or view them. So `onefile = FALSE` is the default.

**Note**

Only some line textures ( $0 \leq lty < 4$ ) are used. Eventually this will be partially remedied, but the XFig file format does not allow as general line textures as the R model. Unimplemented line textures are displayed as *dash-double-dotted*.

There is a limit of 512 colours (plus white and black) per file.

**See Also**

[Devices](#), [postscript](#), [ps.options](#).

## Chapter 4

# The `graphics` package

---

<code>abline</code>	<i>Add a Straight Line to a Plot</i>
---------------------	--------------------------------------

---

### Description

This function adds one or more straight lines through the current plot.

### Usage

```
abline(a, b, untf = FALSE, ...)  
abline(h=, untf = FALSE, ...)  
abline(v=, untf = FALSE, ...)  
abline(coef=, untf = FALSE, ...)  
abline(reg=, untf = FALSE, ...)
```

### Arguments

<code>a, b</code>	the intercept and slope.
<code>untf</code>	logical asking to <i>untransform</i> . See Details.
<code>h</code>	the y-value for a horizontal line.
<code>v</code>	the x-value for a vertical line.
<code>coef</code>	a vector of length two giving the intercept and slope.
<code>reg</code>	an object with a <code>coef</code> component. See Details.
<code>...</code>	graphical parameters.

**Details**

The first form specifies the line in intercept/slope form (alternatively `a` can be specified on its own and is taken to contain the slope and intercept in vector form).

The `h=` and `v=` forms draw horizontal and vertical lines at the specified coordinates.

The `coef` form specifies the line by a vector containing the slope and intercept.

`reg` is a regression object which contains `reg$coef`. If it is of length 1 then the value is taken to be the slope of a line through the origin, otherwise, the first 2 values are taken to be the intercept and slope.

If `untf` is true, and one or both axes are log-transformed, then a curve is drawn corresponding to a line in original coordinates, otherwise a line is drawn in the transformed coordinate system. The `h` and `v` parameters always refer to original coordinates.

The graphical parameters `col` and `lty` can be specified as arguments to `abline`; see `par` for details.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[lines](#) and [segments](#) for connected and arbitrary lines given by their *endpoints*. [par](#).

**Examples**

```
z <- lm(dist ~ speed, data = cars)
plot(cars)
abline(z)
```

---

arrows

*Add Arrows to a Plot*


---

**Description**

Draw arrows between pairs of points.

**Usage**

```
arrows(x0, y0, x1, y1, length = 0.25, angle = 30, code = 2,
       col = par("fg"), lty = NULL, lwd = par("lwd"), xpd = NULL)
```

**Arguments**

<code>x0, y0</code>	coordinates of points <b>from</b> which to draw.
<code>x1, y1</code>	coordinates of points <b>to</b> which to draw.
<code>length</code>	length of the edges of the arrow head (in inches).
<code>angle</code>	angle from the shaft of the arrow to the edge of the arrow head.
<code>code</code>	integer code, determining <i>kind</i> of arrows to be drawn.
<code>col, lty, lwd, xpd</code>	usual graphical parameters as in <a href="#">par</a> .

**Details**

For each `i`, an arrow is drawn between the point  $(x0[i], y0[i])$  and the point  $(x1[i], y1[i])$ .

If `code=2` an arrowhead is drawn at  $(x0[i], y0[i])$  and if `code=1` an arrowhead is drawn at  $(x1[i], y1[i])$ . If `code=3` a head is drawn at both ends of the arrow. Unless `length = 0`, when no head is drawn.

The graphical parameters `col` and `lty` can be used to specify a color and line texture for the line segments which make up the arrows (`col` may be a vector).

The direction of a zero-length arrow is indeterminate, and hence so is the direction of the arrowheads. To allow for rounding error, arrowheads are omitted (with a warning) on any arrow of length less than 1/1000 inch.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[segments](#) to draw segments.

**Examples**

```
x <- runif(12); y <- rnorm(12)
i <- order(x,y); x <- x[i]; y <- y[i]
plot(x,y, main="arrows(.) and segments(.)")
## draw arrows from point to point :
s <- seq(length(x)-1)# one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3)
s <- s[-length(s)]
segments(x[s], y[s], x[s+2], y[s+2], col= 'pink')
```

---

 assocplot

 Association Plots
 

---

### Description

Produce a Cohen-Friendly association plot indicating deviations from independence of rows and columns in a 2-dimensional contingency table.

### Usage

```
assocplot(x, col = c("black", "red"), space = 0.3,
          main = NULL, xlab = NULL, ylab = NULL)
```

### Arguments

<code>x</code>	a two-dimensional contingency table in matrix form.
<code>col</code>	a character vector of length two giving the colors used for drawing positive and negative Pearson residuals, respectively.
<code>space</code>	the amount of space (as a fraction of the average rectangle width and height) left between each rectangle.
<code>main</code>	overall title for the plot.
<code>xlab</code>	a label for the x axis. Defaults to the name of the row variable in <code>x</code> if non-NULL.
<code>ylab</code>	a label for the y axis. Defaults to the column names of the column variable in <code>x</code> if non-NULL.

### Details

For a two-way contingency table, the signed contribution to Pearson's  $\chi^2$  for cell  $i, j$  is  $d_{ij} = (f_{ij} - e_{ij})/\sqrt{e_{ij}}$ , where  $f_{ij}$  and  $e_{ij}$  are the observed and expected counts corresponding to the cell. In the Cohen-Friendly association plot, each cell is represented by a rectangle that has (signed) height proportional to  $d_{ij}$  and width proportional to  $\sqrt{e_{ij}}$ , so that the area of the box is proportional to the difference in observed and expected frequencies. The rectangles in each row are positioned relative to a baseline indicating independence ( $d_{ij} = 0$ ). If the observed frequency of a cell is greater than the expected one, the box rises above the baseline and is shaded in the color specified by the first element of `col`, which defaults to black; otherwise, the box falls below the baseline and is shaded in the color specified by the second element of `col`, which defaults to red.

### References

Cohen, A. (1980), On the graphical display of the significant components in a two-way contingency table. *Communications in Statistics—Theory and Methods*, **A9**, 1025–1041.

Friendly, M. (1992), Graphical methods for categorical data. *SAS User Group International Conference Proceedings*, **17**, 190–200. <http://www.math.yorku.ca/SCS/sugi/sugi17-paper.html>

**See Also**

[mosaicplot](#); [chisq.test](#).

**Examples**

```
## Aggregate over sex:
x <- margin.table(HairEyeColor, c(1, 2))
x
assocplot(x, main = "Relation between hair and eye color")
```

---

axis

---

*Add an Axis to a Plot*


---

**Description**

Adds an axis to the current plot, allowing the specification of the side, position, labels, and other options.

**Usage**

```
axis(side, at = NULL, labels = TRUE, tick = TRUE, line = NA,
      pos = NA, outer = FALSE, font = NA, vfont = NULL,
      lty = "solid", lwd = 1, col = NULL, padj = NA, ...)
```

**Arguments**

<code>side</code>	an integer specifying which side of the plot the axis is to be drawn on. The axis is placed as follows: 1=below, 2=left, 3=above and 4=right.
<code>at</code>	the points at which tick-marks are to be drawn. Non-finite (infinite, NaN or NA) values are omitted. By default, when NULL, tickmark locations are computed, see Details below.
<code>labels</code>	this can either be a logical value specifying whether (numerical) annotations are to be made at the tickmarks, or a vector of character strings to be placed at the tickpoints.
<code>tick</code>	a logical value specifying whether tickmarks should be drawn
<code>line</code>	the number of lines into the margin which the axis will be drawn. This overrides the value of the graphical parameter <code>mgp[3]</code> . The relative placing of tickmarks and tick labels is unchanged.
<code>pos</code>	the coordinate at which the axis line is to be drawn: this overrides the values of both <code>line</code> and <code>mgp[3]</code> .
<code>outer</code>	a logical value indicating whether the axis should be drawn in the outer plot margin, rather than the standard plot margin.
<code>font</code>	font for text.
<code>vfont</code>	vector font for text.

<code>lty, lwd</code>	line type, width for the axis line and the tick marks.
<code>col</code>	color for the axis line and the tick marks. The default <code>NULL</code> means to use <code>par("fg")</code> .
<code>padj</code>	adjustment for each string perpendicular to the reading direction. For strings parallel to the axes, <code>padj=0</code> means right or top alignment, and <code>padj=1</code> means left or bottom alignment. If <code>padj</code> is not a finite value (the default), the value of <code>par("las")</code> determines the adjustment. For strings plotted perpendicular to the axis the default is to centre the string.
<code>...</code>	other graphical parameters may also be passed as arguments to this function, particularly, <code>cex.axis</code> , <code>col.axis</code> and <code>font.axis</code> for axis annotation, <code>las</code> for vertical/horizontal label orientation, or <code>fg</code> instead of <code>col</code> , see <a href="#">par</a> on these.

### Details

The axis line is drawn from the lowest to the highest value of `at`, but will be clipped at the plot region. Only ticks which are drawn from points within the plot region (up to a tolerance for rounding error) are plotted, but the ticks and their labels may well extend outside the plot region.

When `at = NULL`, pretty tick mark locations are computed internally (the same way `axTicks(side)` would) from `par("usr", "lab")` and `par("xlog")` (or `"ylog"`).

Several of the graphics parameters affect the way axes are drawn. The vertical (for sides 1 and 3) positions of the axis and the tick labels are controlled by `mgp`, the size of the ticks is controlled by `tck` and `tcl` and the appearance of the tick labels by `cex.axis`, `col.axis` and `font.axis` with orientation controlled by `las` (but not `srt`, unlike `S` which uses `srt` if `at` is supplied and `las` if it is not).

### Value

This function is invoked for its side effect, which is to add an axis to an already existing plot.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`axTicks` returns the axis tick locations corresponding to `at=NULL`; `pretty` is more flexible for computing pretty tick coordinates and does *not* depend on (nor adapt to) the coordinate system in use.

### Examples

```
plot(1:4, rnorm(4), axes=FALSE)
axis(1, 1:4, LETTERS[1:4])
axis(2)
box() #- to make it look "as usual"

plot(1:7, rnorm(7), main = "axis() examples",
     type = "s", xaxt = "n", frame = FALSE, col = "red")
```

```
axis(1, 1:7, LETTERS[1:7], col.axis = "blue")
# unusual options:
axis(4, col = "violet", col.axis="dark violet", lwd = 2)
axis(3, col = "gold", lty = 2, lwd = 0.5)
```

---

axis.POSIXct                      *Date and Date-time Plotting Functions*

---

## Description

Functions to plot objects of classes "POSIXlt", "POSIXct" and "Date" representing calendar dates and times.

## Usage

```
axis.POSIXct(side, x, at, format, ...)
axis.Date(side, x, at, format, ...)

## S3 method for class 'POSIXct':
plot(x, y, xlab = "", ...)

## S3 method for class 'POSIXlt':
plot(x, y, xlab = "", ...)

## S3 method for class 'Date':
plot(x, y, xlab = "", ...)
```

## Arguments

<code>x, at</code>	A date-time object.
<code>y</code>	numeric values to be plotted against <code>x</code> .
<code>xlab</code>	a character string giving the label for the <code>x</code> axis.
<code>side</code>	See <a href="#">axis</a> .
<code>format</code>	See <a href="#">strptime</a> .
<code>...</code>	Further arguments to be passed from or to other methods, typically graphical parameters or arguments of <a href="#">plot.default</a> .

## Details

The functions plot against an `x`-axis of date-times. `axis.POSIXct` and `axis.Date` work quite hard to choose suitable time units (years, months, days, hours, minutes or seconds) and a sensible output format, but this can be overridden by supplying a `format` specification.

If `at` is supplied it specifies the locations of the ticks and labels whereas if `x` is specified a suitable grid of labels is chosen.

**See Also**

[DateTimeClasses](#), [Dates](#) for details of the classes.

**Examples**

```
attach(beaver1)
time <- strptime(paste(1990, day, time %% 100, time %% 100),
               "%Y %j %H %M")
plot(time, temp, type="l") # axis at 4-hour intervals.
# now label every hour on the time axis
plot(time, temp, type="l", xaxt="n")
r <- as.POSIXct(round(range(time), "hours"))
axis.POSIXct(1, at=seq(r[1], r[2], by="hour"), format="%H")
rm(time)
detach(beaver1)

plot(.leap.seconds, 1:22, type="n", yaxt="n",
     xlab="leap seconds", ylab="", bty="n")
rug(.leap.seconds)
## or as dates
lps <- as.Date(.leap.seconds)
plot(lps, 1:22, type="n", yaxt="n", xlab="leap seconds", ylab="", bty="n")
rug(lps)

## 100 random dates in a 10-week period
random.dates <- as.Date("2001/1/1") + 70*sort(runif(100))
plot(random.dates, 1:100)
# or for a better axis labelling
plot(random.dates, 1:100, xaxt="n")
axis.Date(1, at=seq(as.Date("2001/1/1"), max(random.dates)+6, "weeks"))
```

---

axTicks

*Compute Axis Tickmark Locations*


---

**Description**

Compute pretty tickmark locations, the same way as R does internally. This is only non-trivial when **log** coordinates are active. By default, gives the `at` values which `axis(side)` would use.

**Usage**

```
axTicks(side, axp = NULL, usr = NULL, log = NULL)
```

**Arguments**

`side` integer in 1:4, as for [axis](#).

`axp` numeric vector of length three, defaulting to `par("Zaxp")` where “Z” is “x” or “y” depending on the `side` argument.

usr	numeric vector of length four, defaulting to <code>par("usr")</code> giving horizontal ('x') and vertical ('y') user coordinate limits.
log	logical indicating if log coordinates are active; defaults to <code>par("Zlog")</code> where 'Z' is as for the <code>axp</code> argument above.

### Details

The `axp`, `usr`, and `log` arguments must be consistent as their default values (the `par(...)` results) are. If you specify all three (as non-NULL), the graphics environment is not used at all. Note that the meaning of `axp` alters very much when `log` is TRUE, see the documentation on `par(xaxp=.)`.

`axTicks()` can be regarded as an R implementation of the C function `CreateAtVector()` in `'.../src/main/plot.c'` which is called by `axis(side,*)` when no argument `at` is specified.

### Value

numeric vector of coordinate values at which axis tickmarks can be drawn. By default, when only the first argument is specified, these values should be identical to those that `axis(side)` would use or has used.

### See Also

`axis`, `par.pretty` uses the same algorithm (but independently of the graphics environment) and has more options. However it is not available for `log = TRUE`.

### Examples

```
plot(1:7, 10*21:27)
axTicks(1)
axTicks(2)
stopifnot(identical(axTicks(1), axTicks(3)),
           identical(axTicks(2), axTicks(4)))

## Show how axTicks() and axis() correspond :
op <- par(mfrow = c(3,1))
for(x in 9999*c(1,2,8)) {
  plot(x,9, log = "x")
  cat(formatC(par("xaxp"),wid=5),";",T <- axTicks(1),"\n")
  rug(T, col="red")
}
par(op)
```

---

barplot

*Bar Plots*


---

### Description

Creates a bar plot with vertical or horizontal bars.

**Usage**

```
## Default S3 method:
barplot(height, width = 1, space = NULL,
        names.arg = NULL, legend.text = NULL, beside = FALSE,
        horiz = FALSE, density = NULL, angle = 45,
        col = NULL, border = par("fg"),
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
        xlim = NULL, ylim = NULL, xpd = TRUE,
        axes = TRUE, axisnames = TRUE,
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),
        inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0, ...)
```

**Arguments**

<code>height</code>	either a vector or matrix of values describing the bars which make up the plot. If <code>height</code> is a vector, the plot consists of a sequence of rectangular bars with heights given by the values in the vector. If <code>height</code> is a matrix and <code>beside</code> is <code>FALSE</code> then each bar of the plot corresponds to a column of <code>height</code> , with the values in the column giving the heights of stacked “sub-bars” making up the bar. If <code>height</code> is a matrix and <code>beside</code> is <code>TRUE</code> , then the values in each column are juxtaposed rather than stacked.
<code>width</code>	optional vector of bar widths. Re-cycled to length the number of bars drawn. Specifying a single value will no visible effect unless <code>xlim</code> is specified.
<code>space</code>	the amount of space (as a fraction of the average bar width) left before each bar. May be given as a single number or one number per bar. If <code>height</code> is a matrix and <code>beside</code> is <code>TRUE</code> , <code>space</code> may be specified by two numbers, where the first is the space between bars in the same group, and the second the space between the groups. If not given explicitly, it defaults to <code>c(0, 1)</code> if <code>height</code> is a matrix and <code>beside</code> is <code>TRUE</code> , and to 0.2 otherwise.
<code>names.arg</code>	a vector of names to be plotted below each bar or group of bars. If this argument is omitted, then the names are taken from the <code>names</code> attribute of <code>height</code> if this is a vector, or the column names if it is a matrix.
<code>legend.text</code>	a vector of text used to construct a legend for the plot, or a logical indicating whether a legend should be included. This is only useful when <code>height</code> is a matrix. In that case given legend labels should correspond to the rows of <code>height</code> ; if <code>legend.text</code> is true, the row names of <code>height</code> will be used as labels if they are non-null.
<code>beside</code>	a logical value. If <code>FALSE</code> , the columns of <code>height</code> are portrayed as stacked bars, and if <code>TRUE</code> the columns are portrayed as juxtaposed bars.
<code>horiz</code>	a logical value. If <code>FALSE</code> , the bars are drawn vertically with the first bar to the left. If <code>TRUE</code> , the bars are drawn horizontally with the first at the bottom.
<code>density</code>	a vector giving the density of shading lines, in lines per inch, for the bars or bar components. The default value of <code>NULL</code> means that no shading lines are drawn. Non-positive values of <code>density</code> also inhibit the drawing of shading lines.
<code>angle</code>	the slope of shading lines, given as an angle in degrees (counter-clockwise), for the bars or bar components.

<code>col</code>	a vector of colors for the bars or bar components. By default, grey is used if <code>height</code> is a vector, and a gamma-corrected grey palette if <code>height</code> is a matrix.
<code>border</code>	the color to be used for the border of the bars.
<code>main,sub</code>	overall and sub title for the plot.
<code>xlab</code>	a label for the x axis.
<code>ylab</code>	a label for the y axis.
<code>xlim</code>	limits for the x axis.
<code>ylim</code>	limits for the y axis.
<code>xpd</code>	logical. Should bars be allowed to go outside region?
<code>axes</code>	logical. If TRUE, a vertical (or horizontal, if <code>horiz</code> is true) axis is drawn.
<code>axisnames</code>	logical. If TRUE, and if there are <code>names.arg</code> (see above), the other axis is drawn (with <code>lty=0</code> ) and labeled.
<code>cex.axis</code>	expansion factor for numeric axis labels.
<code>cex.names</code>	expansion factor for axis names (bar labels).
<code>inside</code>	logical. If TRUE, the lines which divide adjacent (non-stacked!) bars will be drawn. Only applies when <code>space = 0</code> (which it partly is when <code>beside = TRUE</code> ).
<code>plot</code>	logical. If FALSE, nothing is plotted.
<code>axis.lty</code>	the graphics parameter <code>lty</code> applied to the axis and tick marks of the categorical (default horizontal) axis. Note that by default the axis is suppressed.
<code>offset</code>	a vector indicating how much the bars should be shifted relative to the x axis.
<code>...</code>	further graphical parameters ( <code>par</code> ) are passed to <code>plot.window()</code> , <code>title()</code> and <code>axis</code> .

### Details

This is a generic function, it currently only has a default method. A formula interface may be added eventually.

### Value

A numeric vector (or matrix, when `beside = TRUE`), say `mp`, giving the coordinates of *all* the bar midpoints drawn, useful for adding to the graph.

If `beside` is true, use `colMeans(mp)` for the midpoints of each *group* of bars, see example.

### Note

Prior to R 1.6.0, `barplot` behaved as if `axis.lty = 1`, unintentionally.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`plot(..., type="h")`, `dotchart`, `hist`.

**Examples**

```
tN <- table(Ni <- rpois(100, lambda=5))
r <- barplot(tN, col='gray')
#- type = "h" plotting *is* 'bar'plot
lines(r, tN, type='h', col='red', lwd=2)

barplot(tN, space = 1.5, axisnames=FALSE,
        sub = "barplot(..., space= 1.5, axisnames = FALSE)")

barplot(VADeaths, plot = FALSE)
barplot(VADeaths, plot = FALSE, beside = TRUE)

mp <- barplot(VADeaths) # default
tot <- colMeans(VADeaths)
text(mp, tot + 3, format(tot), xpd = TRUE, col = "blue")
barplot(VADeaths, beside = TRUE,
        col = c("lightblue", "mistyrose", "lightcyan",
                "lavender", "cornsilk"),
        legend = rownames(VADeaths), ylim = c(0, 100))
title(main = "Death Rates in Virginia", font.main = 4)

hh <- t(VADeaths)[, 5:1]
mybarcol <- "gray20"
mp <- barplot(hh, beside = TRUE,
              col = c("lightblue", "mistyrose",
                    "lightcyan", "lavender"),
              legend = colnames(VADeaths), ylim= c(0,100),
              main = "Death Rates in Virginia", font.main = 4,
              sub = "Faked upper 2*sigma error bars", col.sub = mybarcol,
              cex.names = 1.5)
segments(mp, hh, mp, hh + 2*sqrt(1000*hh/100), col = mybarcol, lwd = 1.5)
stopifnot(dim(mp) == dim(hh))# corresponding matrices
mtext(side = 1, at = colMeans(mp), line = -2,
      text = paste("Mean", formatC(colMeans(hh))), col = "red")

# Bar shading example
barplot(VADeaths, angle = 15+10*1:5, density = 20, col = "black",
        legend = rownames(VADeaths))
title(main = list("Death Rates in Virginia", font = 4))

# border :
barplot(VADeaths, border = "dark blue")
```

**Description**

This function draws a box around the current plot in the given color and linetype. The `bty` parameter determines the type of box drawn. See [par](#) for details.

**Usage**

```
box(which = "plot", lty = "solid", ...)
```

**Arguments**

`which` character, one of "plot", "figure", "inner" and "outer".  
`lty` line type of the box.  
`...` further graphical parameters, such as `bty`, `col`, or `lwd`, see [par](#).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[rect](#) for drawing of arbitrary rectangles.

**Examples**

```
plot(1:7,abs(rnorm(7)), type='h', axes = FALSE)
axis(1, labels = letters[1:7])
box(lty='1373', col = 'red')
```

---

boxplot

*Box Plots*

---

**Description**

Produce box-and-whisker plot(s) of the given (grouped) values.

**Usage**

```
boxplot(x, ...)

## S3 method for class 'formula':
boxplot(formula, data = NULL, ..., subset, na.action = NULL)

## Default S3 method:
boxplot(x, ..., range = 1.5, width = NULL, varwidth = FALSE,
        notch = FALSE, outline = TRUE, names, plot = TRUE,
        border = par("fg"), col = NULL, log = "",
        pars = list(boxwex = 0.8, staplewex = 0.5, outwex = 0.5),
        horizontal = FALSE, add = FALSE, at = NULL)
```

**Arguments**

<code>formula</code>	a formula, such as <code>y ~ grp</code> , where <code>y</code> is a numeric vector of data values to be split into groups according to the grouping variable <code>grp</code> (usually a factor).
<code>data</code>	a <code>data.frame</code> (or list) from which the variables in <code>formula</code> should be taken.
<code>subset</code>	an optional vector specifying a subset of observations to be used for plotting.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. The default is to ignore missing values in either the response or the group.
<code>x</code>	for specifying data from which the boxplots are to be produced. Either a numeric vector, or a single list containing such vectors. Additional unnamed arguments specify further data as separate vectors (each corresponding to a component boxplot). NAs are allowed in the data.
<code>...</code>	For the <code>formula</code> method, arguments to the default method and graphical parameters.  For the default method, unnamed arguments are additional data vectors (unless <code>x</code> is a list when they are ignored), and named arguments are graphical parameters in addition to the ones given by argument <code>pars</code> .
<code>range</code>	this determines how far the plot whiskers extend out from the box. If <code>range</code> is positive, the whiskers extend to the most extreme data point which is no more than <code>range</code> times the interquartile range from the box. A value of zero causes the whiskers to extend to the data extremes.
<code>width</code>	a vector giving the relative widths of the boxes making up the plot.
<code>varwidth</code>	if <code>varwidth</code> is <code>TRUE</code> , the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
<code>notch</code>	if <code>notch</code> is <code>TRUE</code> , a notch is drawn in each side of the boxes. If the notches of two plots do not overlap this is ‘strong evidence’ that the two medians differ (Chambers <i>et al.</i> , 1983, p. 62). See <code>boxplot.stats</code> for the calculations used.
<code>outline</code>	if <code>outline</code> is not true, the outliers are not drawn (as points whereas S+ uses lines).
<code>names</code>	group labels which will be printed under each boxplot.
<code>boxwex</code>	a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower.
<code>staplewex</code>	staple line width expansion, proportional to box width.
<code>outwex</code>	outlier line width expansion, proportional to box width.
<code>plot</code>	if <code>TRUE</code> (the default) then a boxplot is produced. If not, the summaries which the boxplots are based on are returned.
<code>border</code>	an optional vector of colors for the outlines of the boxplots. The values in <code>border</code> are recycled if the length of <code>border</code> is less than the number of plots.
<code>col</code>	if <code>col</code> is non-null it is assumed to contain colors to be used to colour the bodies of the box plots.
<code>log</code>	character indicating if <code>x</code> or <code>y</code> or both coordinates should be plotted in log scale.

<code>pars</code>	a list of (potentially many) more graphical parameters, e.g., <code>boxwex</code> or <code>outpch</code> ; these are passed to <code>bxp</code> (if <code>plot</code> is true); for details, see there.
<code>horizontal</code>	logical indicating if the boxplots should be horizontal; default <code>FALSE</code> means vertical boxes.
<code>add</code>	logical, if true <i>add</i> boxplot to current plot.
<code>at</code>	numeric vector giving the locations where the boxplots should be drawn, particularly when <code>add = TRUE</code> ; defaults to <code>1:n</code> where <code>n</code> is the number of boxes.

### Details

The generic function `boxplot` currently has a default method (`boxplot.default`) and a formula interface (`boxplot.formula`).

Missing values are ignored when forming boxplots.

### Value

List with the following components:

<code>stats</code>	a matrix, each column contains the extreme of the lower whisker, the lower hinge, the median, the upper hinge and the extreme of the upper whisker for one group/plot.
<code>n</code>	a vector with the number of observations in each group.
<code>conf</code>	a matrix where each column contains the lower and upper extremes of the notch.
<code>out</code>	the values of any data points which lie beyond the extremes of the whiskers.
<code>group</code>	a vector of the same length as <code>out</code> whose elements indicate which group the outlier belongs to
<code>names</code>	a vector of names for the groups

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole.

See also `boxplot.stats`.

### See Also

`boxplot.stats` which does the computation, `bxp` for the plotting and more examples; and `stripchart` for an alternative (with small data sets).

**Examples**

```
## boxplot on a formula:
boxplot(count ~ spray, data = InsectSprays, col = "lightgray")
# *add* notches (somewhat funny here):
boxplot(count ~ spray, data = InsectSprays,
        notch = TRUE, add = TRUE, col = "blue")

boxplot(decrease ~ treatment, data = OrchardSprays,
        log = "y", col="bisque")

rb <- boxplot(decrease ~ treatment, data = OrchardSprays, col="bisque")
title("Comparing boxplot()s and non-robust mean +/- SD")

mn.t <- tapply(OrchardSprays$decrease, OrchardSprays$treatment, mean)
sd.t <- tapply(OrchardSprays$decrease, OrchardSprays$treatment, sd)
xi <- 0.3 + seq(rb$n)
points(xi, mn.t, col = "orange", pch = 18)
arrows(xi, mn.t - sd.t, xi, mn.t + sd.t,
       code = 3, col = "pink", angle = 75, length = .1)

## boxplot on a matrix:
mat <- cbind(Uni05 = (1:100)/21, Norm = rnorm(100),
            T5 = rt(100, df = 5), Gam2 = rgamma(100, shape = 2))
boxplot(data.frame(mat), main = "boxplot(data.frame(mat), main = ...)")
par(las=1)# all axis labels horizontal
boxplot(data.frame(mat), main = "boxplot(*, horizontal = TRUE)",
        horizontal = TRUE)

## Using 'at = ' and adding boxplots -- example idea by Roger Bivand :

boxplot(len ~ dose, data = ToothGrowth,
        boxwex = 0.25, at = 1:3 - 0.2,
        subset= supp == "VC", col="yellow",
        main="Guinea Pigs' Tooth Growth",
        xlab="Vitamin C dose mg",
        ylab="tooth length", ylim=c(0,35))
boxplot(len ~ dose, data = ToothGrowth, add = TRUE,
        boxwex = 0.25, at = 1:3 + 0.2,
        subset= supp == "OJ", col="orange")
legend(2, 9, c("Ascorbic acid", "Orange juice"),
      fill = c("yellow", "orange"))

## more examples in help(bxp)
```

---

boxplot.stats

*Box Plot Statistics*


---

**Description**

This function is typically called by `boxplot` to gather the statistics necessary for producing box plots, but may be invoked separately.

**Usage**

```
boxplot.stats(x, coef = 1.5, do.conf = TRUE, do.out = TRUE)
```

**Arguments**

`x` a numeric vector for which the boxplot will be constructed (NAs and NaNs are allowed and omitted).

`coef` this determines how far the plot “whiskers” extend out from the box. If `coef` is positive, the whiskers extend to the most extreme data point which is no more than `coef` times the length of the box away from the box. A value of zero causes the whiskers to extend to the data extremes (and no outliers be returned).

`do.conf, do.out` logicals; if FALSE, the `conf` or `out` component respectively will be empty in the result.

**Details**

The two “hinges” are versions of the first and third quartile, i.e., close to `quantile(x, c(1, 3)/4)`. The hinges equal the quartiles for odd  $n$  (where  $n <- \text{length}(x)$ ) and differ for even  $n$ . Where the quartiles only equal observations for  $n \%\% 4 == 1$  ( $n \equiv 1 \pmod{4}$ ), the hinges do so *additionally* for  $n \%\% 4 == 2$  ( $n \equiv 2 \pmod{4}$ ), and are in the middle of two observations otherwise.

The notches (if requested) extend to  $\pm 1.58 \text{ IQR}/\sqrt{n}$ . This seems to be based on same calculations as the formula with 1.57 in Chambers *et al.* (1983, p. 62), given in McGill *et al.* (1978, p. 16). They are based on asymptotic normality of the median and roughly equal sample sizes for the two medians being compared, and are said to be rather insensitive to the underlying distributions of the samples. The idea appears to be to give roughly a 95% confidence interval for the difference in two medians.

**Value**

List with named components as follows:

`stats` a vector of length 5, containing the extreme of the lower whisker, the lower “hinge”, the median, the upper “hinge” and the extreme of the upper whisker.

`n` the number of non-NA observations in the sample.

`conf` the lower and upper extremes of the “notch” (`if(do.conf)`). See the details.

`out` the values of any data points which lie beyond the extremes of the whiskers (`if(do.out)`).

Note that `$stats` and `$conf` are sorted in *increasing* order, unlike `S`, and that `$n` and `$out` include any  $\pm \text{Inf}$  values.

**References**

Tukey, J. W. (1977) *Exploratory Data Analysis*. Section 2C.

McGill, R., Tukey, J. W. and Larsen, W. A. (1978) Variations of box plots. *The American Statistician* **32**, 12–16.

Velleman, P. F. and Hoaglin, D. C. (1981) *Applications, Basics and Computing of Exploratory Data Analysis*. Duxbury Press.

Emerson, J. D and Strenio, J. (1983). Boxplots and batch comparison. Chapter 3 of *Understanding Robust and Exploratory Data Analysis*, eds. D. C. Hoaglin, F. Mosteller and J. W. Tukey. Wiley.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole.

### See Also

[fivenum](#), [boxplot](#), [bxp](#).

### Examples

```
x <- c(1:100, 1000)
(b1 <- boxplot.stats(x))
(b2 <- boxplot.stats(x, do.conf=FALSE, do.out=FALSE))
stopifnot(b1 $ stats == b2 $ stats) # do.out=F is still robust
boxplot.stats(x, coef = 3, do.conf=FALSE)
## no outlier treatment:
boxplot.stats(x, coef = 0)

boxplot.stats(c(x, NA)) # slight change : n is 101
(r <- boxplot.stats(c(x, -1:1/0)))
stopifnot(r$out == c(1000, -Inf, Inf))
```

---

bxp

*Box Plots from Summaries*

---

### Description

`bxp` draws box plots based on the given summaries in `z`. It is usually called from within `boxplot`, but can be invoked directly.

### Usage

```
bxp(z, notch = FALSE, width = NULL, varwidth = FALSE, outline = TRUE,
    notch.frac = 0.5, log = "", border = par("fg"), col = par("bg"),
    pars = NULL, frame.plot = axes, horizontal = FALSE,
    add = FALSE, at = NULL, show.names = NULL, ...)
```

**Arguments**

<code>z</code>	a list containing data summaries to be used in constructing the plots. These are usually the result of a call to <code>boxplot</code> , but can be generated in any fashion.
<code>notch</code>	if <code>notch</code> is <code>TRUE</code> , a notch is drawn in each side of the boxes. If the notches of two plots do not overlap then the medians are significantly different at the 5 percent level.
<code>width</code>	a vector giving the relative widths of the boxes making up the plot.
<code>varwidth</code>	if <code>varwidth</code> is <code>TRUE</code> , the boxes are drawn with widths proportional to the square-roots of the number of observations in the groups.
<code>outline</code>	if <code>outline</code> is not true, the outliers are not drawn.
<code>notch.frac</code>	numeric in (0,1). When <code>notch=TRUE</code> , the fraction of the box width that the notches should use.
<code>border</code>	character or numeric (vector), the color of the box borders. Is recycled for multiple boxes. Is used as default for the <code>boxcol</code> , <code>medcol</code> , <code>whiskcol</code> , <code>staplecol</code> , and <code>outcol</code> options (see below).
<code>col</code>	character or numeric; the color within the box; recycled for multiple boxes. Is only used as default for <code>boxfill</code> and will be deprecated.
<code>log</code>	character, indicating if any axis should be drawn in logarithmic scale, as in <code>plot.default</code> .
<code>frame.plot</code>	logical, indicating if a “frame” ( <code>box</code> ) should be drawn; defaults to <code>TRUE</code> , unless <code>axes = FALSE</code> is specified.
<code>horizontal</code>	logical indicating if the boxplots should be horizontal; default <code>FALSE</code> means vertical boxes.
<code>add</code>	logical, if true <code>add</code> boxplot to current plot.
<code>at</code>	numeric vector giving the locations where the boxplots should be drawn, particularly when <code>add = TRUE</code> ; defaults to <code>1:n</code> where <code>n</code> is the number of boxes.
<code>show.names</code>	Set to <code>TRUE</code> or <code>FALSE</code> to override the defaults on whether an x-axis label is printed for each group.
<code>pars, ...</code>	graphical parameters can be passed as arguments to this function, either as a list ( <code>pars</code> ) or normally ( <code>...</code> ), see the following. Currently, <code>ylim</code> is used ‘along the boxplot’, i.e., vertically, when <code>horizontal</code> is false. <code>xaxt</code> , <code>yaxt</code> , <code>las</code> , <code>cex.axis</code> , and <code>col.axis</code> are passed to <code>axis</code> , and <code>main</code> , <code>cex.main</code> , <code>col.main</code> , <code>sub</code> , <code>cex.sub</code> , <code>col.sub</code> , <code>xlab</code> , <code>ylab</code> , <code>cex.lab</code> , and <code>col.lab</code> are passed to <code>title</code> . The following arguments (or <code>pars</code> components) allow further customization of the boxplot graphics: <b>boxwex:</b> a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower. <b>staplewex, outwex:</b> staple and outlier line width expansion, proportional to box width. <b>boxlty, boxlwd, boxcol, boxfill:</b> box outline type, width, color, and fill color.

**medlty, medlwd, medpch, medcex, medcol, medbg:** median line type, line width, point character, point size expansion, color, and background color. The default `medpch= NA` suppresses the point, and `medlty="blank"` does so for the line.

**whisklty, whisklwd, whiskcol:** whisker line type, width, and color.

**staplelty, staplelwd, staplecol:** staple (= end of whisker) line type, width, and color.

**outlty, outlwd, outpch, outcex, outcol, outbg:** outlier line type, line width, point character, point size expansion, color, and background color. The default `outlty= "blank"` suppresses the lines and `outpch= " "` suppresses points.

### Value

An invisible vector, actually identical to the `at` argument, with the coordinates ("x" if horizontal is false, "y" otherwise) of box centers, useful for adding to the plot.

### Author(s)

The R Core development team and Arni Magnusson ([arnima@u.washington.edu](mailto:arnima@u.washington.edu)) who has provided most changes for the `box*`, `med*`, `whisk*`, `staple*`, and `out*` arguments.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### Examples

```
set.seed(753)
(bx.p <- boxplot(split(rt(100, 4), gl(5,20))))
op <- par(mfrow= c(2,2))
bxp(bx.p, xaxt = "n")
bxp(bx.p, notch = TRUE, axes = FALSE, pch = 4, boxfill=1:5)
bxp(bx.p, notch = TRUE, boxfill= "lightblue", frame= FALSE, outl= FALSE,
    main = "bxp(*, frame= FALSE, outl= FALSE)")
bxp(bx.p, notch = TRUE, boxfill= "lightblue", border= 2:6, ylim = c(-4,4),
    pch = 22, bg = "green", log = "x", main = "... log='x', ylim=*")
par(op)
op <- par(mfrow= c(1,2))

## single group -- no label
boxplot (weight ~ group, data = PlantGrowth, subset = group=="ctrl")
## with label
bx <- boxplot(weight ~ group, data = PlantGrowth,
              subset = group=="ctrl", plot = FALSE)
bxp(bx, show.names=TRUE)
par(op)

z <- split(rnorm(1000), rpois(1000,2.2))
boxplot(z, whisklty=3, main="boxplot(z, whisklty = 3)")
```

```

## Colour support similar to plot.default:
op <- par(mfrow=1:2, bg="light gray", fg="midnight blue")
  boxplot(z, col.axis="skyblue3", main="boxplot(*, col.axis=..,main=..)")
  plot(z[[1]], col.axis="skyblue3", main="plot(*, col.axis=..,main=..)")
  mtext("par(bg=\"light gray\", fg=\"midnight blue\"),
        outer = TRUE, line = -1.2)
par(op)

## Mimic S-Plus:
splus <- list(boxwex=0.4, staplewex=1, outwex=1, boxfill="grey40",
             medlwd=3, medcol="white", whisklty=3, outlty=1, outpch=" ")
boxplot(z, pars=splus)
## Recycled and "sweeping" parameters
op <- par(mfrow=c(1,2))
  boxplot(z, border=1:5, lty = 3, medlty = 1, medlwd = 2.5)
  boxplot(z, boxfill=1:3, pch=1:5, lwd = 1.5, medcol="white")
par(op)
## too many possibilities
boxplot(z, boxfill= "light gray", outpch = 21:25, outlty = 2,
        bg = "pink", lwd = 2, medcol = "dark blue", medcex = 2, medpch=20)

```

---

chull

*Compute Convex Hull of a Set of Points*


---

### Description

Computes the subset of points which lie on the convex hull of the set of points specified.

### Usage

```
chull(x, y = NULL)
```

### Arguments

*x*, *y* coordinate vectors of points. This can be specified as two vectors *x* and *y*, a 2-column matrix *x*, a list *x* with two components, etc, see [xy.coords](#).

### Details

[xy.coords](#) is used to interpret the specification of the points. The algorithm is that given by Eddy (1977).

‘Peeling’ as used in the S function `chull` can be implemented by calling `chull` recursively.

### Value

An integer vector giving the indices of the points lying on the convex hull, in clockwise order.

## References

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
- Eddy, W. F. (1977) A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, **3**, 398–403.
- Eddy, W. F. (1977) Algorithm 523. CONVEX, A new convex hull algorithm for planar sets[Z]. *ACM Transactions on Mathematical Software*, **3**, 411–412.

## See Also

[xy.coords](#), [polygon](#)

## Examples

```
X <- matrix(rnorm(2000), ncol = 2)
plot(X, cex = 0.5)
hpts <- chull(X)
hpts <- c(hpts, hpts[1])
lines(X[hpts, ])
```

---

contour

*Display Contours*

---

## Description

Create a contour plot, or add contour lines to an existing plot.

## Usage

```
contour(x, ...)

## Default S3 method:
contour(x = seq(0, 1, len = nrow(z)),
        y = seq(0, 1, len = ncol(z)),
        z,
        nlevels = 10, levels = pretty(zlim, nlevels), labels = NULL,
        xlim = range(x, finite = TRUE),
        ylim = range(y, finite = TRUE),
        zlim = range(z, finite = TRUE),
        labcex = 0.6, drawlabels = TRUE, method = "flattest",
        vfont = c("sans serif", "plain"),
        axes = TRUE, frame.plot = axes,
        col = par("fg"), lty = par("lty"), lwd = par("lwd"),
        add = FALSE, ...)
```

```
contourLines(x = seq(0, 1, len = nrow(z)),
             y = seq(0, 1, len = ncol(z)),
```

```
z, nlevels = 10,
levels = pretty(range(z, na.rm=TRUE), nlevels))
```

### Arguments

<code>x, y</code>	locations of grid lines at which the values in <code>z</code> are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If <code>x</code> is a <code>list</code> , its components <code>x\$x</code> and <code>x\$y</code> are used for <code>x</code> and <code>y</code> , respectively. If the list has component <code>z</code> this is used for <code>z</code> .
<code>z</code>	a matrix containing the values to be plotted (NAs are allowed). Note that <code>x</code> can be used instead of <code>z</code> for convenience.
<code>nlevels</code>	number of contour levels desired <b>iff</b> <code>levels</code> is not supplied.
<code>levels</code>	numeric vector of levels at which to draw contour lines.
<code>labels</code>	a vector giving the labels for the contour lines. If <code>NULL</code> then the levels are used as labels.
<code>labcex</code>	<code>cex</code> for contour labelling.
<code>drawlabels</code>	logical. Contours are labelled if <code>TRUE</code> .
<code>method</code>	character string specifying where the labels will be located. Possible values are "simple", "edge" and "flattest" (the default). See the Details section.
<code>vfont</code>	if a character vector of length 2 is specified, then Hershey vector fonts are used for the contour labels. The first element of the vector selects a typeface and the second element selects a fontindex (see <a href="#">text</a> for more information).
<code>xlim, ylim, zlim</code>	x-, y- and z-limits for the plot.
<code>axes, frame.plot</code>	logical indicating whether axes or a box should be drawn, see <a href="#">plot.default</a> .
<code>col</code>	color for the lines drawn.
<code>lty</code>	line type for the lines drawn.
<code>lwd</code>	line width for the lines drawn.
<code>add</code>	logical. If <code>TRUE</code> , add to a current plot.
<code>...</code>	additional graphical parameters (see <a href="#">par</a> ). The plot aspect ratio <code>asp</code> (see <a href="#">plot.window</a> ) and the arguments to <a href="#">title</a> may also be supplied.

### Details

`contour` is a generic function with only a default method in base R.

`contourLines` draws nothing, but returns a set of contour lines.

There is currently no documentation about the algorithm. The source code is in `'$R_HOME/src/main/plot3d.c'`.

The methods for positioning the labels on contours are "simple" (draw at the edge of the plot, overlaying the contour line), "edge" (draw at the edge of the plot, embedded in the contour line, with no labels overlapping) and "flattest" (draw on the flattest section of the contour, embedded in the contour line, with no labels overlapping). The second and third may not draw a label on every contour line.

For information about vector fonts, see the help for [text](#) and [Hershey](#).

Notice that `contour` interprets the `z` matrix as a table of  $f(x[i], y[j])$  values, so that the `x` axis corresponds to row number and the `y` axis to column number, with column 1 at the bottom, i.e. a 90 degree clockwise rotation of the conventional textual layout.

### Value

`contourLines` returns a list of contours. Each contour is a list with elements:

<code>level</code>	The contour level.
<code>x</code>	The x-coordinates of the contour.
<code>y</code>	The y-coordinates of the contour.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[filled.contour](#) for “color-filled” contours, [image](#) and the graphics demo which can be invoked as `demo(graphics)`.

### Examples

```
x <- -6:16
op <- par(mfrow = c(2, 2))
contour(outer(x, x), method = "edge", vfont = c("sans serif", "plain"))
z <- outer(x, sqrt(abs(x)), FUN = "/")
## Should not be necessary:
z[!is.finite(z)] <- NA
image(x, x, z)
contour(x, x, z, col = "pink", add = TRUE, method = "edge",
        vfont = c("sans serif", "plain"))
contour(x, x, z, ylim = c(1, 6), method = "simple", labcex = 1)
contour(x, x, z, ylim = c(-6, 6), nlev = 20, lty = 2, method = "simple")
par(op)

## Persian Rug Art:
x <- y <- seq(-4*pi, 4*pi, len = 27)
r <- sqrt(outer(x^2, y^2, "+"))
opar <- par(mfrow = c(2, 2), mar = rep(0, 4))
for(f in pi^(0:3))
  contour(cos(r^2)*exp(-r/f),
          drawlabels = FALSE, axes = FALSE, frame = TRUE)

rx <- range(x <- 10*1:nrow(volcano))
ry <- range(y <- 10*1:ncol(volcano))
ry <- ry + c(-1,1) * (diff(rx) - diff(ry))/2
tcol <- terrain.colors(12)
par(opar); opar <- par(pty = "s", bg = "lightcyan")
```

```

plot(x = 0, y = 0, type = "n", xlim = rx, ylim = ry, xlab = "", ylab = "")
u <- par("usr")
rect(u[1], u[3], u[2], u[4], col = tcol[8], border = "red")
contour(x, y, volcano, col = tcol[2], lty = "solid", add = TRUE,
        vfont = c("sans serif", "plain"))
title("A Topographic Map of Maunga Whau", font = 4)
abline(h = 200*0:4, v = 200*0:4, col = "lightgray", lty = 2, lwd = 0.1)

## contourLines produces the same contour lines as contour
line.list <- contourLines(x, y, volcano)
plot(x = 0, y = 0, type = "n", xlim = rx, ylim = ry, xlab = "", ylab = "")
u <- par("usr")
rect(u[1], u[3], u[2], u[4], col = tcol[8], border = "red")
contour(x, y, volcano, col = tcol[2], lty = "solid", add = TRUE,
        vfont = c("sans serif", "plain"))
templines <- function(clines) {
  lines(clines[[2]], clines[[3]])
}
invisible(lapply(line.list, templines))
par(opar)

```

---

coplot

*Conditioning Plots*


---

## Description

This function produces two variants of the **conditioning** plots discussed in the reference below.

## Usage

```

coplot(formula, data, given.values, panel = points, rows, columns,
       show.given = TRUE, col = par("fg"), pch = par("pch"),
       bar.bg = c(num = gray(0.8), fac = gray(0.95)),
       xlab = c(x.name, paste("Given :", a.name)),
       ylab = c(y.name, paste("Given :", b.name)),
       subscripts = FALSE,
       axlabels = function(f) abbreviate(levels(f)),
       number = 6, overlap = 0.5, xlim, ylim, ...)
co.intervals(x, number = 6, overlap = 0.5)

```

## Arguments

**formula** a formula describing the form of conditioning plot. A formula of the form  $y \sim x \mid a$  indicates that plots of  $y$  versus  $x$  should be produced conditional on the variable  $a$ . A formula of the form  $y \sim x \mid a * b$  indicates that plots of  $y$  versus  $x$  should be produced conditional on the two variables  $a$  and  $b$ .

All three or four variables may be either numeric or factors. When  $x$  or  $y$  are factors, the result is almost as if `as.numeric()` was applied, whereas for factor  $a$  or  $b$ , the conditioning (and its graphics if `show.given` is true) are adapted.

<code>data</code>	a data frame containing values for any variables in the formula. By default the environment where <code>coplot</code> was called from is used.
<code>given.values</code>	a value or list of two values which determine how the conditioning on <code>a</code> and <code>b</code> is to take place. When there is no <code>b</code> (i.e., conditioning only on <code>a</code> ), usually this is a matrix with two columns each row of which gives an interval, to be conditioned on, but is can also be a single vector of numbers or a set of factor levels (if the variable being conditioned on is a factor). In this case (no <code>b</code> ), the result of <code>co.intervals</code> can be used directly as <code>given.values</code> argument.
<code>panel</code>	a <code>function(x, y, col, pch, ...)</code> which gives the action to be carried out in each panel of the display. The default is <code>points</code> .
<code>rows</code>	the panels of the plot are laid out in a <code>rows</code> by <code>columns</code> array. <code>rows</code> gives the number of rows in the array.
<code>columns</code>	the number of columns in the panel layout array.
<code>show.given</code>	logical (possibly of length 2 for 2 conditioning variables): should conditioning plots be shown for the corresponding conditioning variables (default <code>TRUE</code> )
<code>col</code>	a vector of colors to be used to plot the points. If too short, the values are recycled.
<code>pch</code>	a vector of plotting symbols or characters. If too short, the values are recycled.
<code>bar.bg</code>	a named vector with components <code>"num"</code> and <code>"fac"</code> giving the background colors for the (shingle) bars, for <b>numeric</b> and <b>factor</b> conditioning variables respectively.
<code>xlab</code>	character; labels to use for the x axis and the first conditioning variable. If only one label is given, it is used for the x axis and the default label is used for the conditioning variable.
<code>ylab</code>	character; labels to use for the y axis and any second conditioning variable.
<code>subscripts</code>	logical: if true the panel function is given an additional (third) argument <code>subscripts</code> giving the subscripts of the data passed to that panel.
<code>axlabels</code>	function for creating axis (tick) labels when x or y are factors.
<code>number</code>	integer; the number of conditioning intervals, for <code>a</code> and <code>b</code> , possibly of length 2. It is only used if the corresponding conditioning variable is not a <code>factor</code> .
<code>overlap</code>	numeric $< 1$ ; the fraction of overlap of the conditioning variables, possibly of length 2 for x and y direction. When <code>overlap &lt; 0</code> , there will be <i>gaps</i> between the data slices.
<code>xlim</code>	the range for the x axis.
<code>ylim</code>	the range for the y axis.
<code>...</code>	additional arguments to the panel function.
<code>x</code>	a numeric vector.

### Details

In the case of a single conditioning variable `a`, when both `rows` and `columns` are unspecified, a “close to square” layout is chosen with `columns >= rows`.

In the case of multiple rows, the *order* of the panel plots is from the bottom and from the left (corresponding to increasing *a*, typically).

A panel function should not attempt to start a new plot, but just plot within a given coordinate system: thus `plot` and `boxplot` are not panel functions.

As from R 2.0.0 the rendering of arguments `xlab` and `ylab` is not controlled by `par` arguments `cex.lab` and `font.lab` even though they are plotted by `mtext` rather than `title`.

## Value

`co.intervals(. , number, .)` returns a  $(\text{number} \times 2)$  *matrix*, say `ci`, where `ci[k, ]` is the *range* of *x* values for the *k*-th interval.

## References

Chambers, J. M. (1992) *Data for models*. Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Cleveland, W. S. (1993) *Visualizing Data*. New Jersey: Summit Press.

## See Also

[pairs](#), [panel.smooth](#), [points](#).

## Examples

```
## Tonga Trench Earthquakes
coplot(lat ~ long | depth, data = quakes)
given.depth <- co.intervals(quakes$depth, number=4, overlap=.1)
coplot(lat ~ long | depth, data = quakes, given.v=given.depth, rows=1)

## Conditioning on 2 variables:
ll.dm <- lat ~ long | depth * mag
coplot(ll.dm, data = quakes)
coplot(ll.dm, data = quakes, number=c(4,7), show.given=c(TRUE,FALSE))
coplot(ll.dm, data = quakes, number=c(3,7),
       overlap=c(-.5,.1)) # negative overlap DROPS values

## given two factors
Index <- seq(length=nrow(warpbreaks)) # to get nicer default labels
coplot(breaks ~ Index | wool * tension, data = warpbreaks, show.given = 0:1)
coplot(breaks ~ Index | wool * tension, data = warpbreaks,
       col = "red", bg = "pink", pch = 21, bar.bg = c(fac = "light blue"))

## Example with empty panels:
attach(data.frame(state.x77))#> don't need 'data' arg. below
coplot(Life.Exp ~ Income | Illiteracy * state.region, number = 3,
       panel = function(x, y, ...) panel.smooth(x, y, span = .8, ...))
## y ~ factor -- not really sensical, but 'show off':
coplot(Life.Exp ~ state.region | Income * state.division,
       panel = panel.smooth)
detach() # data.frame(state.x77)
```

curve

*Draw Function Plots***Description**

Draws a curve corresponding to the given function or expression (in `x`) over the interval `[from,to]`.

**Usage**

```
curve(expr, from, to, n = 101, add = FALSE, type = "l",
      ylab = NULL, log = NULL, xlim = NULL, ...)

## S3 method for class 'function':
plot(x, from = 0, to = 1, xlim = NULL, ...)
```

**Arguments**

<code>expr</code>	an expression written as a function of <code>x</code> , or alternatively the name of a function which will be plotted.
<code>x</code>	a ‘vectorizing’ numeric R function.
<code>from,to</code>	the range over which the function will be plotted.
<code>n</code>	integer; the number of <code>x</code> values at which to evaluate.
<code>add</code>	logical; if <code>TRUE</code> add to already existing plot.
<code>xlim</code>	numeric of length 2; if specified, it serves as default for <code>c(from, to)</code> .
<code>type, ylab, log, ...</code>	graphical parameters can also be specified as arguments. <code>plot.function</code> passes all these to <code>curve</code> .

**Details**

The evaluation of `expr` is at `n` points equally spaced over the range `[from, to]`, possibly adapted to log scale. The points determined in this way are then joined with straight lines. `x(t)` or `expr` (with `x` inside) must return a numeric of the same length as the argument `t` or `x`.

If `add = TRUE`, `c(from,to)` default to `xlim` which defaults to the current `x`-limits. Further, `log` is taken from the current plot when `add` is true.

This used to be a quick hack which now seems to serve a useful purpose, but can give bad results for functions which are not smooth.

For “expensive” expressions, you should use smarter tools.

**See Also**

[splinefun](#) for spline interpolation, [lines](#).

**Examples**

```

op <- par(mfrow=c(2,2))
curve(x^3-3*x, -2, 2)
curve(x^2-2, add = TRUE, col = "violet")

plot(cos, xlim = c(-pi,3*pi), n = 1001, col = "blue")

chippy <- function(x) sin(cos(x)*exp(-x/2))
curve(chippy, -8, 7, n=2001)
curve(chippy, -8, -5)

for(ll in c("", "x", "y", "xy"))
  curve(log(1+x), 1,100, log=ll, sub=paste("log= ",ll,"'",sep=""))
par(op)

```

dotchart

*Cleveland Dot Plots***Description**

Draw a Cleveland dot plot.

**Usage**

```

dotchart(x, labels = NULL, groups = NULL, gdata = NULL,
         cex = par("cex"), pch = 21, gpch = 21, bg = par("bg"),
         color = par("fg"), gcolor = par("fg"), lcolor = "gray",
         xlim = range(x[is.finite(x)]),
         main = NULL, xlab = NULL, ylab = NULL, ...)

```

**Arguments**

<code>x</code>	either a vector or matrix of numeric values (NAs are allowed). If <code>x</code> is a matrix the overall plot consists of juxtaposed dotplots for each row.
<code>labels</code>	a vector of labels for each point. For vectors the default is to use <code>names(x)</code> and for matrices the row labels <code>dimnames(x)[[1]]</code> .
<code>groups</code>	an optional factor indicating how the elements of <code>x</code> are grouped. If <code>x</code> is a matrix, <code>groups</code> will default to the columns of <code>x</code> .
<code>gdata</code>	data values for the groups. This is typically a summary such as the median or mean of each group.
<code>cex</code>	the character size to be used. Setting <code>cex</code> to a value smaller than one can be a useful way of avoiding label overlap.
<code>pch</code>	the plotting character or symbol to be used.
<code>gpch</code>	the plotting character or symbol to be used for group values.
<code>bg</code>	the background color of plotting characters or symbols to be used; use <code>par(bg= *)</code> to set the background color of the whole plot.

color	the color(s) to be used for points and labels.
gcolor	the single color to be used for group labels and values.
lcolor	the color(s) to be used for the horizontal lines.
xlim	horizontal range for the plot, see <code>plot.window</code> , e.g.
main	overall title for the plot, see <code>title</code> .
xlab, ylab	axis annotations as in <code>title</code> .
...	graphical parameters can also be specified as arguments.

**Value**

This function is invoked for its side effect, which is to produce two variants of dotplots as described in Cleveland (1985).

Dot plots are a reasonable substitute for bar plots.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The Elements of Graphing Data*. Monterey, CA: Wadsworth.

**Examples**

```
dotchart(VADeaths, main = "Death Rates in Virginia - 1940")
op <- par(xaxs="i")# 0 -- 100%
dotchart(t(VADeaths), xlim = c(0,100),
         main = "Death Rates in Virginia - 1940")
par(op)
```

---

filled.contour      *Level (Contour) Plots*

---

**Description**

This function produces a contour plot with the areas between the contours filled in solid color (Cleveland calls this a level plot). A key showing how the colors map to z values is shown to the right of the plot.

**Usage**

```
filled.contour(x = seq(0, 1, len = nrow(z)),
              y = seq(0, 1, len = ncol(z)),
              z,
              xlim = range(x, finite=TRUE),
              ylim = range(y, finite=TRUE),
              zlim = range(z, finite=TRUE),
              levels = pretty(zlim, nlevels), nlevels = 20,
```

```

color.palette = cm.colors,
col = color.palette(length(levels) - 1),
plot.title, plot.axes, key.title, key.axes,
asp = NA, xaxs = "i", yaxs = "i", las = 1,
axes = TRUE, frame.plot = axes, ...)

```

### Arguments

<code>x, y</code>	locations of grid lines at which the values in <code>z</code> are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If <code>x</code> is a list, its components <code>x\$x</code> and <code>x\$y</code> are used for <code>x</code> and <code>y</code> , respectively. If the list has component <code>z</code> this is used for <code>z</code> .
<code>z</code>	a matrix containing the values to be plotted (NAs are allowed). Note that <code>x</code> can be used instead of <code>z</code> for convenience.
<code>xlim</code>	<code>x</code> limits for the plot.
<code>ylim</code>	<code>y</code> limits for the plot.
<code>zlim</code>	<code>z</code> limits for the plot.
<code>levels</code>	a set of levels which are used to partition the range of <code>z</code> . Must be <b>strictly</b> increasing (and finite). Areas with <code>z</code> values between consecutive levels are painted with the same color.
<code>nlevels</code>	if <code>levels</code> is not specified, the range of <code>z</code> , values is divided into approximately this many levels.
<code>color.palette</code>	a color palette function to be used to assign colors in the plot.
<code>col</code>	an explicit set of colors to be used in the plot. This argument overrides any palette function specification.
<code>plot.title</code>	statements which add titles to the main plot.
<code>plot.axes</code>	statements which draw axes (and a <a href="#">box</a> ) on the main plot. This overrides the default axes.
<code>key.title</code>	statements which add titles for the plot key.
<code>key.axes</code>	statements which draw axes on the plot key. This overrides the default axis.
<code>asp</code>	the $y/x$ aspect ratio, see <a href="#">plot.window</a> .
<code>xaxs</code>	the <code>x</code> axis style. The default is to use internal labeling.
<code>yaxs</code>	the <code>y</code> axis style. The default is to use internal labeling.
<code>las</code>	the style of labeling to be used. The default is to use horizontal labeling.
<code>axes, frame.plot</code>	logicals indicating if axes and a box should be drawn, as in <a href="#">plot.default</a> .
<code>...</code>	additional graphical parameters, currently only passed to <a href="#">title()</a> .

**Note**

This function currently uses the `layout` function and so is restricted to a full page display. As an alternative consider the `levelplot` function from the **lattice** package which works in multipanel displays.

The output produced by `filled.contour` is actually a combination of two plots; one is the filled contour and one is the legend. Two separate coordinate systems are set up for these two plots, but they are only used internally - once the function has returned these coordinate systems are lost. If you want to annotate the main contour plot, for example to add points, you can specify graphics commands in the `plot.axes` argument. An example is given below.

**Author(s)**

Ross Ihaka.

**References**

Cleveland, W. S. (1993) *Visualizing Data*. Summit, New Jersey: Hobart.

**See Also**

[contour](#), [image](#), [palette](#); [levelplot](#) from package **lattice**.

**Examples**

```
filled.contour(volcano, color = terrain.colors, asp = 1)# simple

x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
filled.contour(x, y, volcano, color = terrain.colors,
  plot.title = title(main = "The Topography of Maunga Whau",
    xlab = "Meters North", ylab = "Meters West"),
  plot.axes = { axis(1, seq(100, 800, by = 100))
    axis(2, seq(100, 600, by = 100)) },
  key.title = title(main="Height\n(meters)"),
  key.axes = axis(4, seq(90, 190, by = 10))# maybe also asp=1
mtext(paste("filled.contour(.) from", R.version.string),
  side = 1, line = 4, adj = 1, cex = .66)

# Annotating a filled contour plot
a <- expand.grid(1:20, 1:20)
b <- matrix(a[,1] + a[,2], 20)
filled.contour(x = 1:20, y = 1:20, z = b,
  plot.axes={ axis(1); axis(2); points(10,10) })

## Persian Rug Art:
x <- y <- seq(-4*pi, 4*pi, len = 27)
r <- sqrt(outer(x^2, y^2, "+"))
filled.contour(cos(r^2)*exp(-r/(2*pi)), axes = FALSE)
## rather, the key *should* be labeled:
filled.contour(cos(r^2)*exp(-r/(2*pi)), frame.plot = FALSE, plot.axes = {})
```

---

fourfoldplot	<i>Fourfold Plots</i>
--------------	-----------------------

---

### Description

Creates a fourfold display of a 2 by 2 by  $k$  contingency table on the current graphics device, allowing for the visual inspection of the association between two dichotomous variables in one or several populations (strata).

### Usage

```
fourfoldplot(x, color = c("#99CCFF", "#6699CC"), conf.level = 0.95,
             std = c("margins", "ind.max", "all.max"),
             margin = c(1, 2), space = 0.2, main = NULL,
             mfrow = NULL, mfc col = NULL)
```

### Arguments

<code>x</code>	a 2 by 2 by $k$ contingency table in array form, or as a 2 by 2 matrix if $k$ is 1.
<code>color</code>	a vector of length 2 specifying the colors to use for the smaller and larger diagonals of each 2 by 2 table.
<code>conf.level</code>	confidence level used for the confidence rings on the odds ratios. Must be a single nonnegative number less than 1; if set to 0, confidence rings are suppressed.
<code>std</code>	a character string specifying how to standardize the table. Must be one of "margins", "ind.max", or "all.max", and can be abbreviated by the initial letter. If set to "margins", each 2 by 2 table is standardized to equate the margins specified by <code>margin</code> while preserving the odds ratio. If "ind.max" or "all.max", the tables are either individually or simultaneously standardized to a maximal cell frequency of 1.
<code>margin</code>	a numeric vector with the margins to equate. Must be one of 1, 2, or <code>c(1, 2)</code> (the default), which corresponds to standardizing the row, column, or both margins in each 2 by 2 table. Only used if <code>std</code> equals "margins".
<code>space</code>	the amount of space (as a fraction of the maximal radius of the quarter circles) used for the row and column labels.
<code>main</code>	character string for the fourfold title.
<code>mfrow</code>	a numeric vector of the form <code>c(nr, nc)</code> , indicating that the displays for the 2 by 2 tables should be arranged in an <code>nr</code> by <code>nc</code> layout, filled by rows.
<code>mfc col</code>	a numeric vector of the form <code>c(nr, nc)</code> , indicating that the displays for the 2 by 2 tables should be arranged in an <code>nr</code> by <code>nc</code> layout, filled by columns.

## Details

The fourfold display is designed for the display of 2 by 2 by  $k$  tables.

Following suitable standardization, the cell frequencies  $f_{ij}$  of each 2 by 2 table are shown as a quarter circle whose radius is proportional to  $\sqrt{f_{ij}}$  so that its area is proportional to the cell frequency. An association (odds ratio different from 1) between the binary row and column variables is indicated by the tendency of diagonally opposite cells in one direction to differ in size from those in the other direction; color is used to show this direction. Confidence rings for the odds ratio allow a visual test of the null of no association; the rings for adjacent quadrants overlap iff the observed counts are consistent with the null hypothesis.

Typically, the number  $k$  corresponds to the number of levels of a stratifying variable, and it is of interest to see whether the association is homogeneous across strata. The fourfold display visualizes the pattern of association. Note that the confidence rings for the individual odds ratios are not adjusted for multiple testing.

## References

Friendly, M. (1994). A fourfold display for 2 by 2 by  $k$  tables. Technical Report 217, York University, Psychology Department. <http://www.math.yorku.ca/SCS/Papers/4fold/4fold.ps.gz>

## See Also

[mosaicplot](#)

## Examples

```
## Use the Berkeley admission data as in Friendly (1995).
x <- aperm(UCBAdmissions, c(2, 1, 3))
dimnames(x)[[2]] <- c("Yes", "No")
names(dimnames(x)) <- c("Sex", "Admit?", "Department")
stats::ftable(x)

## Fourfold display of data aggregated over departments, with
## frequencies standardized to equate the margins for admission
## and sex.
## Figure 1 in Friendly (1994).
fourfoldplot(margin.table(x, c(1, 2)))

## Fourfold display of x, with frequencies in each table
## standardized to equate the margins for admission and sex.
## Figure 2 in Friendly (1994).
fourfoldplot(x)

## Fourfold display of x, with frequencies in each table
## standardized to equate the margins for admission. but not
## for sex.
## Figure 3 in Friendly (1994).
fourfoldplot(x, margin = 2)
```

---

`frame`*Create / Start a New Plot Frame*

---

**Description**

This function (`frame` is an alias for `plot.new`) causes the completion of plotting in the current plot (if there is one) and an advance to a new graphics frame. This is used in all high-level plotting functions and also useful for skipping plots when a multi-figure region is in use.

**Usage**

```
plot.new()  
frame()
```

**Details**

There is a hook called "`plot.new`" (see [setHook](#)) called immediately after advancing the frame, which is used in the testing code to annotate the new page. The hook function(s) are called with no argument. (If the value is a character string, `get` is called on it from within the **graphics** namespace.)

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (`frame`.)

**See Also**

[plot.window](#), [plot.default](#).

---

`grid`*Add Grid to a Plot*

---

**Description**

`grid` adds an `nx` by `ny` rectangular grid to an existing plot.

**Usage**

```
grid(nx = NULL, ny = nx, col = "lightgray", lty = "dotted",  
     lwd = NULL, equilogs = TRUE)
```

**Arguments**

<code>nx,ny</code>	number of cells of the grid in x and y direction. When <code>NULL</code> , as per default, the grid aligns with the tick marks on the corresponding <i>default</i> axis (i.e., tick-marks as computed by <code>axTicks</code> ). When <code>NA</code> , no grid lines are drawn in the corresponding direction.
<code>col</code>	character or (integer) numeric; color of the grid lines.
<code>lty</code>	character or (integer) numeric; line type of the grid lines.
<code>lwd</code>	non-negative numeric giving line width of the grid lines; defaults to <code>par("lwd")</code> .
<code>equilogs</code>	logical, only used when <i>log</i> coordinates and alignment with the axis tick marks are active. Setting <code>equilogs = FALSE</code> in that case gives <i>non equidistant</i> tick aligned grid lines.

**Note**

If more fine tuning is required, use `abline(h = ., v = .)` directly.

**See Also**

`plot`, `abline`, `lines`, `points`.

**Examples**

```
plot(1:3)
grid(NA, 5, lwd = 2) # grid only in y-direction

## maybe change the desired number of tick marks: par(lab=c(mx,my,7))
op <- par(mfcol = 1:2)
with(iris,
  {
    plot(Sepal.Length, Sepal.Width, col = as.integer(Species),
         xlim = c(4, 8), ylim = c(2, 4.5), panel.first = grid(),
         main = "with(iris, plot(..., panel.first = grid(), ..) )")
    plot(Sepal.Length, Sepal.Width, col = as.integer(Species),
         panel.first = grid(3, lty=1,lwd=2),
         main = "... panel.first = grid(3, lty=1,lwd=2), ..")
  }
)
par(op)
```

---

hist

*Histograms*


---

**Description**

The generic function `hist` computes a histogram of the given data values. If `plot=TRUE`, the resulting object of class `"histogram"` is plotted by `plot.histogram`, before it is returned.

**Usage**

```
hist(x, ...)

## Default S3 method:
hist(x, breaks = "Sturges", freq = NULL, probability = !freq,
     include.lowest = TRUE, right = TRUE,
     density = NULL, angle = 45, col = NULL, border = NULL,
     main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL,
     xlab = xname, ylab,
     axes = TRUE, plot = TRUE, labels = FALSE,
     nclass = NULL, ...)
```

**Arguments**

<code>x</code>	a vector of values for which the histogram is desired.
<code>breaks</code>	one of: <ul style="list-style-type: none"> <li>• a vector giving the breakpoints between histogram cells,</li> <li>• a single number giving the number of cells for the histogram,</li> <li>• a character string naming an algorithm to compute the number of cells (see <a href="#">Details</a>),</li> <li>• a function to compute the number of cells.</li> </ul> <p>In the last three cases the number is a suggestion only.</p>
<code>freq</code>	logical; if TRUE, the histogram graphic is a representation of frequencies, the counts component of the result; if FALSE, <i>relative</i> frequencies (“probabilities”), component density, are plotted. Defaults to TRUE <i>iff</i> breaks are equidistant (and probability is not specified).
<code>probability</code>	an <i>alias</i> for ! <code>freq</code> , for S compatibility.
<code>include.lowest</code>	logical; if TRUE, an <code>x[i]</code> equal to the <code>breaks</code> value will be included in the first (or last, for <code>right = FALSE</code> ) bar. This will be ignored (with a warning) unless <code>breaks</code> is a vector.
<code>right</code>	logical; if TRUE, the histograms cells are right-closed (left open) intervals.
<code>density</code>	the density of shading lines, in lines per inch. The default value of NULL means that no shading lines are drawn. Non-positive values of <code>density</code> also inhibit the drawing of shading lines.
<code>angle</code>	the slope of shading lines, given as an angle in degrees (counter-clockwise).
<code>col</code>	a colour to be used to fill the bars. The default of NULL yields unfilled bars.
<code>border</code>	the color of the border around the bars. The default is to use the standard foreground color.
<code>main, xlab, ylab</code>	these arguments to <code>title</code> have useful defaults here.
<code>xlim, ylim</code>	the range of <code>x</code> and <code>y</code> values with sensible defaults. Note that <code>xlim</code> is <i>not</i> used to define the histogram ( <code>breaks</code> ), but only for plotting (when <code>plot = TRUE</code> ).

<code>axes</code>	logical. If TRUE (default), axes are draw if the plot is drawn.
<code>plot</code>	logical. If TRUE (default), a histogram is plotted, otherwise a list of breaks and counts is returned.
<code>labels</code>	logical or character. Additionally draw labels on top of bars, if not FALSE; see <a href="#">plot.histogram</a> .
<code>nclass</code>	numeric (integer). For S(-PLUS) compatibility only, <code>nclass</code> is equivalent to <code>breaks</code> for a scalar or character argument.
<code>...</code>	further graphical parameters to <code>title</code> and <code>axis</code> .

### Details

The definition of “histogram” differs by source (with country-specific biases). R’s default with equi-spaced breaks (also the default) is to plot the counts in the cells defined by `breaks`. Thus the height of a rectangle is proportional to the number of points falling into the cell, as is the area *provided* the breaks are equally-spaced.

The default with non-equi-spaced breaks is to give a plot of area one, in which the *area* of the rectangles is the fraction of the data points falling in the cells.

If `right = TRUE` (default), the histogram cells are intervals of the form  $(a, b]$ , i.e., they include their right-hand endpoint, but not their left one, with the exception of the first cell when `include.lowest` is TRUE.

For `right = FALSE`, the intervals are of the form  $[a, b)$ , and `include.lowest` really has the meaning of “include highest”.

A numerical tolerance of  $10^{-7}$  times the median bin size is applied when counting entries on the edges of bins.

The default for `breaks` is "Sturges": see [nclass.Sturges](#). Other names for which algorithms are supplied are "Scott" and "FD" / "Friedman-Diaconis" (with corresponding functions [nclass.scott](#) and [nclass.FD](#)). Case is ignored and partial matching is used. Alternatively, a function can be supplied which will compute the intended number of breaks as a function of `x`.

### Value

an object of class "histogram" which is a list with components:

<code>breaks</code>	the $n + 1$ cell boundaries (= <code>breaks</code> if that was a vector).
<code>counts</code>	$n$ integers; for each cell, the number of <code>x[ ]</code> inside.
<code>density</code>	values $\hat{f}(x_i)$ , as estimated density values. If <code>all(diff(breaks) == 1)</code> , they are the relative frequencies <code>counts/n</code> and in general satisfy $\sum_i \hat{f}(x_i)(b_{i+1} - b_i) = 1$ , where $b_i = \text{breaks}[i]$ .
<code>intensities</code>	same as <code>density</code> . Deprecated, but retained for compatibility.
<code>mids</code>	the $n$ cell midpoints.
<code>xname</code>	a character string with the actual <code>x</code> argument name.
<code>equidist</code>	logical, indicating if the distances between <code>breaks</code> are all the same.

**Note**

The resulting value does *not* depend on the values of the arguments `freq` (or probability) or `plot`. This is intentionally different from `S`.

Prior to R 1.7.0, the element `breaks` of the result was adjusted for numerical tolerances. The nominal values are now returned even though tolerances are still used when counting.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Springer.

**See Also**

`nclass.Sturges`, `stem`, `density`, `truehist` in package `MASS`.

**Examples**

```
op <- par(mfrow=c(2, 2))
hist(islands)
utils::str(hist(islands, col="gray", labels = TRUE))

hist(sqrt(islands), br = 12, col="lightblue", border="pink")
##-- For non-equidistant breaks, counts should NOT be graphed unscaled:
r <- hist(sqrt(islands), br = c(4*0:5, 10*3:5, 70, 100, 140), col='blue1')
text(r$mid, r$density, r$counts, adj=c(.5, -.5), col='blue3')
sapply(r[2:3], sum)
sum(r$density * diff(r$breaks)) # == 1
lines(r, lty = 3, border = "purple") # -> lines.histogram(*)
par(op)

utils::str(hist(islands, br=12, plot= FALSE)) #-> 10 (~= 12) breaks
utils::str(hist(islands, br=c(12,20,36,80,200,1000,17000), plot = FALSE))

hist(islands, br=c(12,20,36,80,200,1000,17000), freq = TRUE,
      main = "WRONG histogram") # and warning
```

---

hist.POSIXt

*Histogram of a Date or Date-Time Object*


---

**Description**

Method for `hist` applied to date or date-time objects.

**Usage**

```
## S3 method for class 'POSIXt':
hist(x, breaks, ..., plot = TRUE, freq = FALSE,
     start.on.monday = TRUE, format)

## S3 method for class 'Date':
hist(x, breaks, ..., plot = TRUE, freq = FALSE,
     start.on.monday = TRUE, format)
```

**Arguments**

<code>x</code>	an object inheriting from class "POSIXt" or "Date".
<code>breaks</code>	a vector of cut points <i>or</i> number giving the number of intervals which <code>x</code> is to be cut into <i>or</i> an interval specification, one of "days", "weeks", "months" or "years", plus "secs", "mins", "hours" for date-time objects.
<code>...</code>	graphical parameters, or arguments to <code>hist.default</code> such as <code>include.lowest</code> , <code>right</code> and <code>labels</code> .
<code>plot</code>	logical. If TRUE (default), a histogram is plotted, otherwise a list of breaks and counts is returned.
<code>freq</code>	logical; if TRUE, the histogram graphic is a representation of frequencies, i.e. the counts component of the result; if FALSE, <i>relative</i> frequencies ("probabilities") are plotted.
<code>start.on.monday</code>	logical. If <code>breaks = "weeks"</code> , should the week start on Mondays or Sundays?
<code>format</code>	for the x-axis labels. See <code>strptime</code> .

**Value**

An object of class "histogram": see `hist`.

**See Also**

`seq.POSIXt`, `axis.POSIXct`, `hist`

**Examples**

```
hist(.leap.seconds, "years", freq = TRUE)
hist(.leap.seconds,
     seq(ISOdate(1970, 1, 10), ISOdate(2002, 1, 1), "5 years"))

## 100 random dates in a 10-week period
random.dates <- as.Date("2001/1/1") + 70*runif(100)
hist(random.dates, "weeks", format = "%d %b")
```

---

 identify

*Identify Points in a Scatter Plot*


---

### Description

`identify` reads the position of the graphics pointer when the (first) mouse button is pressed. It then searches the coordinates given in `x` and `y` for the point closest to the pointer. If this point is close to the pointer, its index will be returned as part of the value of the call.

### Usage

```
identify(x, ...)
```

```
## Default S3 method:
```

```
identify(x, y = NULL, labels = seq(along = x), pos = FALSE,
         n = length(x), plot = TRUE, offset = 0.5, ...)
```

### Arguments

<code>x, y</code>	coordinates of points in a scatter plot. Alternatively, any object which defines coordinates (a plotting structure, time series etc.) can be given as <code>x</code> and <code>y</code> left undefined.
<code>labels</code>	an optional vector, the same length as <code>x</code> and <code>y</code> , giving labels for the points.
<code>pos</code>	if <code>pos</code> is <code>TRUE</code> , a component is added to the return value which indicates where text was plotted relative to each identified point (1=below, 2=left, 3=above and 4=right).
<code>n</code>	the maximum number of points to be identified.
<code>plot</code>	if <code>plot</code> is <code>TRUE</code> , the labels are printed at the points and if <code>FALSE</code> they are omitted.
<code>offset</code>	the distance (in character widths) which separates the label from identified points.
<code>...</code>	further arguments to <code>par(.)</code> .

### Details

`identify` is only supported on screen devices such as `X11`, `windows` and `quartz`. On other devices the call will do nothing.

If `plot` is `TRUE`, the point is labelled with the corresponding element of `text`. The labels are placed either below, to the left, above or to the right of the identified point, depending on where the cursor was.

For the usual `X11` device the identification process is terminated by pressing any mouse button other than the first. For the `quartz` device the process is terminated by pressing the `ESC` key.

On most devices which support `identify`, successful selection of a point is indicated by a bell sound unless `options(locatorBell=FALSE)` has been set.

If the window is resized or hidden and then exposed before the identification process has terminated, any labels drawn by `identify` will disappear. These will reappear once the identification process has terminated and the window is resized or hidden and exposed again. This is because the labels drawn by `identify` are not recorded in the device's display list until the identification process has terminated.

### Value

If `pos` is `FALSE`, an integer vector containing the indexes of the identified points.

If `pos` is `TRUE`, a list containing a component `ind`, indicating which points were identified and a component `pos`, indicating where the labels were placed relative to the identified points.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[locator](#)

---

image

*Display a Color Image*

---

### Description

Creates a grid of colored or gray-scale rectangles with colors corresponding to the values in `z`. This can be used to display three-dimensional or spatial data aka "images". This is a generic function.

The functions `heat.colors`, `terrain.colors` and `topo.colors` create heat-spectrum (red to white) and topographical color schemes suitable for displaying ordered data, with `n` giving the number of colors desired.

### Usage

```
image(x, ...)

## Default S3 method:
image(x, y, z, zlim, xlim, ylim, col = heat.colors(12),
      add = FALSE, xaxs = "i", yaxs = "i", xlab, ylab,
      breaks, oldstyle = FALSE, ...)
```

### Arguments

`x, y` locations of grid lines at which the values in `z` are measured. These must be in (strictly) ascending order. By default, equally spaced values from 0 to 1 are used. If `x` is a `list`, its components `x$x` and `x$y` are used for `x` and `y`, respectively. If the list has component `z` this is used for `z`.

<code>z</code>	a matrix containing the values to be plotted (NAs are allowed). Note that <code>x</code> can be used instead of <code>z</code> for convenience.
<code>zlim</code>	the minimum and maximum <code>z</code> values for which colors should be plotted. Each of the given colors will be used to color an equispaced interval of this range. The <i>midpoints</i> of the intervals cover the range, so that values just outside the range will be plotted.
<code>xlim, ylim</code>	ranges for the plotted <code>x</code> and <code>y</code> values, defaulting to the range of the finite values of <code>x</code> and <code>y</code> .
<code>col</code>	a list of colors such as that generated by <code>rainbow</code> , <code>heat.colors</code> , <code>topo.colors</code> , <code>terrain.colors</code> or similar functions.
<code>add</code>	logical; if TRUE, add to current plot (and disregard the following arguments). This is rarely useful because <code>image</code> “paints” over existing graphics.
<code>xaxs, yaxs</code>	style of <code>x</code> and <code>y</code> axis. The default “ <code>i</code> ” is appropriate for images. See <code>par</code> .
<code>xlab, ylab</code>	each a character string giving the labels for the <code>x</code> and <code>y</code> axis. Default to the ‘call names’ of <code>x</code> or <code>y</code> , or to “” if these were unspecified.
<code>breaks</code>	a set of breakpoints for the colours: must give one more breakpoint than colour.
<code>oldstyle</code>	logical. If true the midpoints of the colour intervals are equally spaced, and <code>zlim[1]</code> and <code>zlim[2]</code> were taken to be midpoints. (This was the default prior to R 1.1.0.) The current default is to have colour intervals of equal lengths between the limits.
<code>...</code>	graphical parameters for <code>plot</code> may also be passed as arguments to this function, as can the plot aspect ratio <code>asp</code> (see <code>plot.window</code> ).

### Details

The length of `x` should be equal to the `nrow(z)+1` or `nrow(z)`. In the first case `x` specifies the boundaries between the cells: in the second case `x` specifies the midpoints of the cells. Similar reasoning applies to `y`. It probably only makes sense to specify the midpoints of an equally-spaced grid. If you specify just one row or column and a length-one `x` or `y`, the whole user area in the corresponding direction is filled.

If `breaks` is specified then `zlim` is unused and the algorithm used follows `cut`, so intervals are closed on the right and open on the left except for the lowest interval.

Notice that `image` interprets the `z` matrix as a table of  $f(x[i], y[j])$  values, so that the `x` axis corresponds to row number and the `y` axis to column number, with column 1 at the bottom, i.e. a 90 degree clockwise rotation of the conventional textual layout.

### Note

Based on a function by Thomas Lumley (`tlumley@u.washington.edu`).

### See Also

`filled.contour` or `heatmap` which can look nicer (but are less modular), `contour`; `heat.colors`, `topo.colors`, `terrain.colors`, `rainbow`, `hsv`, `par`.

**Examples**

```
x <- y <- seq(-4*pi, 4*pi, len=27)
r <- sqrt(outer(x^2, y^2, "+"))
image(z = z <- cos(r^2)*exp(-r/6), col=gray((0:32)/32))
image(z, axes = FALSE, main = "Math can be beautiful ...",
      xlab = expression(cos(r^2) * e^{-r/6}))
contour(z, add = TRUE, drawlabels = FALSE)

# Volcano data visualized as matrix. Need to transpose and flip
# matrix horizontally.
image(t(volcano)[ncol(volcano):1,])

# A prettier display of the volcano
x <- 10*(1:nrow(volcano))
y <- 10*(1:ncol(volcano))
image(x, y, volcano, col = terrain.colors(100), axes = FALSE)
contour(x, y, volcano, levels = seq(90, 200, by = 5),
       add = TRUE, col = "peru")
axis(1, at = seq(100, 800, by = 100))
axis(2, at = seq(100, 600, by = 100))
box()
title(main = "Maunga Whau Volcano", font.main = 4)
```

---

 layout

*Specifying Complex Plot Arrangements*


---

**Description**

layout divides the device up into as many rows and columns as there are in matrix *mat*, with the column-widths and the row-heights specified in the respective arguments.

**Usage**

```
layout(mat,
       widths = rep(1, dim(mat)[2]),
       heights = rep(1, dim(mat)[1]),
       respect = FALSE)

layout.show(n = 1)
lcm(x)
```

**Arguments**

*mat* a matrix object specifying the location of the next *N* figures on the output device. Each value in the matrix must be 0 or a positive integer. If *N* is the largest positive integer in the matrix, then the integers  $\{1, \dots, N - 1\}$  must also appear at least once in the matrix.

<code>widths</code>	a vector of values for the widths of columns on the device. Relative widths are specified with numeric values. Absolute widths (in centimetres) are specified with the <code>lcm()</code> function (see examples).
<code>heights</code>	a vector of values for the heights of rows on the device. Relative and absolute heights can be specified, see <code>widths</code> above.
<code>respect</code>	either a logical value or a matrix object. If the latter, then it must have the same dimensions as <code>mat</code> and each value in the matrix must be either 0 or 1.
<code>n</code>	number of figures to plot.
<code>x</code>	a dimension to be interpreted as a number of centimetres.

### Details

Figure  $i$  is allocated a region composed from a subset of these rows and columns, based on the rows and columns in which  $i$  occurs in `mat`.

The `respect` argument controls whether a unit column-width is the same physical measurement on the device as a unit row-height.

`layout.show(n)` plots (part of) the current layout, namely the outlines of the next  $n$  figures.

`lcm` is a trivial function, to be used as *the* interface for specifying absolute dimensions for the `widths` and `heights` arguments of `layout()`.

### Value

`layout` returns the number of figures,  $N$ , see above.

### Warnings

These functions are totally incompatible with the other mechanisms for arranging plots on a device: `par(mfrow)`, `par(mfcol)` and `split.screen`.

### Author(s)

Paul R. Murrell

### References

Murrell, P. R. (1999) Layouts: A mechanism for arranging plots on a page. *Journal of Computational and Graphical Statistics*, **8**, 121-134. Chapter 5 of Paul Murrell's Ph.D. thesis.

### See Also

`par` with arguments `mfrow`, `mfcol`, or `mfg`.

**Examples**

```

def.par <- par(no.readonly = TRUE) # save default, for resetting...

## divide the device into two rows and two columns
## allocate figure 1 all of row 1
## allocate figure 2 the intersection of column 2 and row 2
layout(matrix(c(1,1,0,2), 2, 2, byrow = TRUE))
## show the regions that have been allocated to each plot
layout.show(2)

## divide device into two rows and two columns
## allocate figure 1 and figure 2 as above
## respect relations between widths and heights
nf <- layout(matrix(c(1,1,0,2), 2, 2, byrow=TRUE), respect=TRUE)
layout.show(nf)

## create single figure which is 5cm square
nf <- layout(matrix(1), widths=lcm(5), heights=lcm(5))
layout.show(nf)

##-- Create a scatterplot with marginal histograms -----

x <- pmin(3, pmax(-3, rnorm(50)))
y <- pmin(3, pmax(-3, rnorm(50)))
xhist <- hist(x, breaks=seq(-3,3,0.5), plot=FALSE)
yhist <- hist(y, breaks=seq(-3,3,0.5), plot=FALSE)
top <- max(c(xhist$counts, yhist$counts))
xrange <- c(-3,3)
yrange <- c(-3,3)
nf <- layout(matrix(c(2,0,1,3),2,2,byrow=TRUE), c(3,1), c(1,3), TRUE)
layout.show(nf)

par(mar=c(3,3,1,1))
plot(x, y, xlim=xrange, ylim=yrange, xlab="", ylab="")
par(mar=c(0,3,1,1))
barplot(xhist$counts, axes=FALSE, ylim=c(0, top), space=0)
par(mar=c(3,0,1,1))
barplot(yhist$counts, axes=FALSE, xlim=c(0, top), space=0, horiz=TRUE)

par(def.par)#- reset to default

```

---

legend

---

*Add Legends to Plots*


---

**Description**

This function can be used to add legends to plots. Note that a call to the function `locator` can be used in place of the `x` and `y` arguments.

**Usage**

```
legend(x, y = NULL, legend, fill = NULL, col = "black",
       lty, lwd, pch,
       angle = 45, density = NULL, bty = "o", bg = par("bg"),
       pt.bg = NA, cex = 1, pt.cex = cex, pt.lwd = lwd,
       xjust = 0, yjust = 1, x.intersp = 1, y.intersp = 1,
       adj = c(0, 0.5), text.width = NULL, text.col = par("col"),
       merge = do.lines && has.pch, trace = FALSE,
       plot = TRUE, ncol = 1, horiz = FALSE)
```

**Arguments**

<code>x, y</code>	the x and y co-ordinates to be used to position the legend. They can be specified in any way which is accepted by <a href="#">xy.coords</a> : See Details.
<code>legend</code>	a vector of text values or an <a href="#">expression</a> of length $\geq 1$ , or a <a href="#">call</a> (as resulting from <a href="#">substitute</a> ) to appear in the legend.
<code>fill</code>	if specified, this argument will cause boxes filled with the specified colors (or shaded in the specified colors) to appear beside the legend text.
<code>col</code>	the color of points or lines appearing in the legend.
<code>lty, lwd</code>	the line types and widths for lines appearing in the legend. One of these two <i>must</i> be specified for line drawing.
<code>pch</code>	the plotting symbols appearing in the legend, either as vector of 1-character strings, or one (multi character) string. <i>Must</i> be specified for symbol drawing.
<code>angle</code>	angle of shading lines.
<code>density</code>	the density of shading lines, if numeric and positive. If NULL or negative or NA color filling is assumed.
<code>bty</code>	the type of box to be drawn around the legend. The allowed values are "o" (the default) and "n".
<code>bg</code>	the background color for the legend box. (Note that this is only used if <code>bty != "n"</code> .)
<code>pt.bg</code>	the background color for the <a href="#">points</a> .
<code>cex</code>	character expansion factor <b>relative</b> to current <code>par("cex")</code> .
<code>pt.cex</code>	expansion factor(s) for the points.
<code>pt.lwd</code>	line width for the points, defaults to the one for lines.
<code>xjust</code>	how the legend is to be justified relative to the legend x location. A value of 0 means left justified, 0.5 means centered and 1 means right justified.
<code>yjust</code>	the same as <code>xjust</code> for the legend y location.
<code>x.intersp</code>	character interspacing factor for horizontal (x) spacing.
<code>y.intersp</code>	the same for vertical (y) line distances.
<code>adj</code>	numeric of length 1 or 2; the string adjustment for legend text. Useful for y-adjustment when <code>labels</code> are <a href="#">plotmath</a> expressions.
<code>text.width</code>	the width of the legend text in x ("user") coordinates. Defaults to the proper value computed by <a href="#">strwidth</a> ( <code>legend</code> ).

<code>text.col</code>	the color used for the legend text.
<code>merge</code>	logical; if TRUE, “merge” points and lines but not filled boxes. Defaults to TRUE if there are points and lines.
<code>trace</code>	logical; if TRUE, shows how legend does all its magical computations.
<code>plot</code>	logical. If FALSE, nothing is plotted but the sizes are returned.
<code>ncol</code>	the number of columns in which to set the legend items (default is 1, a vertical legend).
<code>horiz</code>	logical; if TRUE, set the legend horizontally rather than vertically (specifying <code>horiz</code> overrides the <code>ncol</code> specification).

### Details

Arguments `x`, `y`, `legend` are interpreted in a non-standard way to allow the coordinates to be specified *via* one or two arguments. If `legend` is missing and `y` is not numeric, it is assumed that the second argument is intended to be `legend` and that the first argument specifies the coordinates.

The coordinates can be specified in any way which is accepted by `xy.coords`. If this gives the coordinates of one point, it is used as the top-left coordinate of the rectangle containing the legend. If it gives the coordinates of two points, these specify opposite corners of the rectangle (either pair of corners, in any order).

“Attribute” arguments such as `col`, `pch`, `lty`, etc, are recycled if necessary. `merge` is not.

Points are drawn *after* lines in order that they can cover the line with their background color `pt.bg`, if applicable.

See the examples for how to right-justify labels.

### Value

A list with list components

<code>rect</code>	a list with components <b>w</b> , <b>h</b> positive numbers giving width and height of the legend’s box. <b>left</b> , <b>top</b> x and y coordinates of upper left corner of the box.
<code>text</code>	a list with components <b>x</b> , <b>y</b> numeric vectors of length <code>length(legend)</code> , giving the x and y coordinates of the legend’s text(s).

returned invisibly.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`plot`, `barplot` which uses `legend()`, and `text` for more examples of math expressions.

## Examples

```

## Run the example in '?matplot' or the following:
leg.txt <- c("Setosa      Petals", "Setosa      Sepals",
            "Versicolor Petals", "Versicolor Sepals")
y.leg <- c(4.5, 3, 2.1, 1.4, .7)
cexv <- c(1.2, 1, 4/5, 2/3, 1/2)
matplot(c(1,8), c(0,4.5), type = "n", xlab = "Length", ylab = "Width",
        main = "Petal and Sepal Dimensions in Iris Blossoms")
for (i in seq(cexv)) {
  text (1, y.leg[i]-.1, paste("cex=",formatC(cexv[i])), cex=.8, adj = 0)
  legend(3, y.leg[i], leg.txt, pch = "sSvV", col = c(1, 3), cex = cexv[i])
}

## 'merge = TRUE' for merging lines & points:
x <- seq(-pi, pi, len = 65)
plot(x, sin(x), type = "l", ylim = c(-1.2, 1.8), col = 3, lty = 2)
points(x, cos(x), pch = 3, col = 4)
lines(x, tan(x), type = "b", lty = 1, pch = 4, col = 6)
title("legend(..., lty = c(2, -1, 1), pch = c(-1,3,4), merge = TRUE)",
      cex.main = 1.1)
legend(-1, 1.9, c("sin", "cos", "tan"), col = c(3,4,6), text.col= "green4",
      lty = c(2, -1, 1), pch = c(-1, 3, 4), merge = TRUE, bg='gray90')

## right-justifying a set of labels: thanks to Uwe Ligges
x <- 1:5; y1 <- 1/x; y2 <- 2/x
plot(rep(x, 2), c(y1, y2), type="n", xlab="x", ylab="y")
lines(x, y1); lines(x, y2, lty=2)
temp <- legend(5, 2, legend = c(" ", " "),
              text.width = strwidth("1,000,000"),
              lty = 1:2, xjust = 1, yjust = 1)
text(temp$rect$left + temp$rect$w, temp$text$y,
      c("1,000", "1,000,000"), pos=2)

##--- log scaled Examples -----
leg.txt <- c("a one", "a two")

par(mfrow = c(2,2))
for(ll in c("", "x", "y", "xy")) {
  plot(2:10, log=ll, main=paste("log = '",ll,"'", sep=""))
  abline(1,1)
  lines(2:3,3:4, col=2) #
  points(2,2, col=3)   #
  rect(2,3,3,2, col=4)
  text(c(3,3),2:3, c("rect(2,3,3,2, col=4)",
                    "text(c(3,3),2:3,\"c(rect(...)\")"), adj = c(0,.3))
  legend(list(x=2,y=8), legend = leg.txt, col=2:3, pch=1:2,
        lty=1, merge=TRUE)#, trace=TRUE)
}
par(mfrow=c(1,1))

##-- Math expressions: -----
x <- seq(-pi, pi, len = 65)

```

```

plot(x, sin(x), type="l", col = 2, xlab = expression(phi),
     ylab = expression(f(phi)))
abline(h=-1:1, v=pi/2*(-6:6), col="gray90")
lines(x, cos(x), col = 3, lty = 2)
ex.cs1 <- expression(plain(sin) * phi, paste("cos", phi))# 2 ways
utils::str(legend(-3, .9, ex.cs1, lty=1:2, plot=FALSE,
                 adj = c(0, .6)))# adj y !
legend(-3, .9, ex.cs1, lty=1:2, col=2:3, adj = c(0, .6))

x <- rexp(100, rate = .5)
hist(x, main = "Mean and Median of a Skewed Distribution")
abline(v = mean(x), col=2, lty=2, lwd=2)
abline(v = median(x), col=3, lty=3, lwd=2)
ex12 <- expression(bar(x) == sum(over(x[i], n), i==1, n),
                  hat(x) == median(x[i], i==1,n))
utils::str(legend(4.1, 30, ex12, col = 2:3, lty=2:3, lwd=2))

## 'Filled' boxes -- for more, see example(plotfactor)
op <- par(bg="white") # to get an opaque box for the legend
plot(cut(weight, 3) ~ group, data = PlantGrowth, col = NULL,
     density = 16*(1:3))
par(op)

## Using 'ncol' :
x <- 0:64/64
matplot(x, outer(x, 1:7, function(x, k) sin(k * pi * x)),
        type = "o", col = 1:7, ylim = c(-1, 1.5), pch = "*")
op <- par(bg="antiquewhite1")
legend(0, 1.5, paste("sin(", 1:7, "pi * x)"), col=1:7, lty=1:7, pch = "*",
      ncol = 4, cex = 0.8)
legend(.8,1.2, paste("sin(", 1:7, "pi * x)"), col=1:7, lty=1:7,
      pch = "*", cex = 0.8)
legend(0, -.1, paste("sin(", 1:4, "pi * x)"), col=1:4, lty=1:4,
      ncol = 2, cex = 0.8)
legend(0, -.4, paste("sin(", 5:7, "pi * x)"), col=4:6, pch=24,
      ncol = 2, cex = 1.5, lwd = 2, pt.bg = "pink", pt.cex = 1:3)
par(op)

## point covering line :
y <- sin(3*pi*x)
plot(x, y, type="l", col="blue", main = "points with bg & legend(*, pt.bg)")
points(x, y, pch=21, bg="white")
legend(.4,1, "sin(c x)", pch=21, pt.bg="white", lty=1, col = "blue")

```

---

lines

---

*Add Connected Line Segments to a Plot*


---

### Description

A generic function taking coordinates given in various ways and joining the corresponding points with line segments.

**Usage**

```
lines(x, ...)

## Default S3 method:
lines(x, y = NULL, type = "l", col = par("col"),
      lty = par("lty"), ...)
```

**Arguments**

<code>x</code> , <code>y</code>	coordinate vectors of points to join.
<code>type</code>	character indicating the type of plotting; actually any of the <code>types</code> as in <a href="#">plot</a> .
<code>col</code>	color to use. This can be vector of length greater than one, but only the first value will be used.
<code>lty</code>	line type to use.
<code>...</code>	Further graphical parameters (see <a href="#">par</a> ) may also be supplied as arguments, particularly, line type, <code>lty</code> and line width, <code>lwd</code> .

**Details**

The coordinates can be passed to `lines` in a plotting structure (a list with `x` and `y` components), a time series, etc. See [xy.coords](#).

The coordinates can contain NA values. If a point contains NA in either its `x` or `y` value, it is omitted from the plot, and lines are not drawn to or from such points. Thus missing values can be used to achieve breaks in lines.

For `type = "h"`, `col` can be a vector and will be recycled as needed.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[points](#), particularly for `type %in% c("p", "b", "o")`, [plot](#), and the underlying “primitive” [plot.xy](#).

[par](#) for how to specify colors.

**Examples**

```
# draw a smooth line through a scatter plot
plot(cars, main="Stopping Distance versus Speed")
lines(lowess(cars))
```

---

 locator

*Graphical Input*


---

### Description

Reads the position of the graphics cursor when the (first) mouse button is pressed.

### Usage

```
locator(n = 512, type = "n", ...)
```

### Arguments

<code>n</code>	the maximum number of points to locate. Valid values start at 1.
<code>type</code>	One of "n", "p", "l" or "o". If "p" or "o" the points are plotted; if "l" or "o" they are joined by lines.
<code>...</code>	additional graphics parameters used if <code>type != "n"</code> for plotting the locations.

### Details

`locator` is only supported on screen devices such as `X11`, `windows` and `quartz`. On other devices the call will do nothing.

Unless the process is terminated prematurely by the user (see below) at most `n` positions are determined.

For the usual `X11` device the identification process is terminated by pressing any mouse button other than the first. For the `quartz` device the process is terminated by pressing the `ESC` key.

The current graphics parameters apply just as if `plot.default` has been called with the same value of `type`. The plotting of the points and lines is subject to clipping, but locations outside the current clipping rectangle will be returned.

On most devices which support `locator`, successful selection of a point is indicated by a bell sound unless `options(locatorBell=FALSE)` has been set.

If the window is resized or hidden and then exposed before the input process has terminated, any lines or points drawn by `locator` will disappear. These will reappear once the input process has terminated and the window is resized or hidden and exposed again. This is because the points and lines drawn by `locator` are not recorded in the device's display list until the input process has terminated.

### Value

A list containing `x` and `y` components which are the coordinates of the identified points in the user coordinate system, i.e., the one specified by `par("usr")`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[identify](#)

---

 matplot

---

*Plot Columns of Matrices*


---

**Description**

Plot the columns of one matrix against the columns of another.

**Usage**

```
matplot(x, y, type = "p", lty = 1:5, lwd = 1, pch = NULL,
        col = 1:6, cex = NULL,
        xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
        ..., add = FALSE, verbose = getOption("verbose"))
```

```
matpoints(x, y, type = "p", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)
```

```
matlines (x, y, type = "l", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)
```

**Arguments**

<code>x, y</code>	vectors or matrices of data for plotting. The number of rows should match. If one of them are missing, the other is taken as <code>y</code> and an <code>x</code> vector of <code>1:n</code> is used. Missing values (NAs) are allowed.
<code>type</code>	character string (length 1 vector) or vector of 1-character strings indicating the type of plot for each column of <code>y</code> , see <a href="#">plot</a> for all possible types. The first character of <code>type</code> defines the first plot, the second character the second, etc. Characters in <code>type</code> are cycled through; e.g., "pl" alternately plots points and lines.
<code>lty, lwd</code>	vector of line types and widths. The first element is for the first column, the second element for the second column, etc., even if lines are not plotted for all columns. Line types will be used cyclically until all plots are drawn.
<code>pch</code>	character string or vector of 1-characters or integers for plotting characters, see <a href="#">points</a> . The first character is the plotting-character for the first plot, the second for the second, etc. The default is the digits (1 through 9, 0) then the letters.
<code>col</code>	vector of colors. Colors are used cyclically.

<code>cex</code>	vector of character expansion sizes, used cyclically.
<code>xlab, ylab</code>	titles for x and y axes, as in <code>plot</code> .
<code>xlim, ylim</code>	ranges of x and y axes, as in <code>plot</code> .
<code>...</code>	Graphical parameters (see <code>par</code> ) and any further arguments of <code>plot</code> , typically <code>plot.default</code> , may also be supplied as arguments to this function. Hence, the high-level graphics control arguments described under <code>par</code> and the arguments to <code>title</code> may be supplied to this function.
<code>add</code>	logical. If TRUE, plots are added to current one, using <code>points</code> and <code>lines</code> .
<code>verbose</code>	logical. If TRUE, write one line of what is done.

### Details

Points involving missing values are not plotted.

The first column of `x` is plotted against the first column of `y`, the second column of `x` against the second column of `y`, etc. If one matrix has fewer columns, plotting will cycle back through the columns again. (In particular, either `x` or `y` may be a vector, against which all columns of the other argument will be plotted.)

The first element of `col`, `cex`, `lty`, `lwd` is used to plot the axes as well as the first line.

Because plotting symbols are drawn with lines and because these functions may be changing the line style, you should probably specify `lty=1` when using plotting symbols.

### Side Effects

Function `matplot` generates a new plot; `matpoints` and `matlines` add to the current one.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`plot`, `points`, `lines`, `matrix`, `par`.

### Examples

```
matplot((-4:5)^2, main = "Quadratic") # almost identical to plot(*)
sines <- outer(1:20, 1:4, function(x, y) sin(x / 20 * pi * y))
matplot(sines, pch = 1:4, type = "o", col = rainbow(ncol(sines)))

x <- 0:50/50
matplot(x, outer(x, 1:8, function(x, k) sin(k*pi * x)),
        ylim = c(-2,2), type = "plobcsSh",
        main= "matplot(type = \"plobcsSh\" )")
## pch & type = vector of 1-chars :
matplot(x, outer(x, 1:4, function(x, k) sin(k*pi * x)),
        pch = letters[1:4], type = c("b","p","o"))
```

```

table(iris$Species) # is data.frame with 'Species' factor
iS <- iris$Species == "setosa"
iV <- iris$Species == "versicolor"
op <- par(bg = "bisque")
matplot(c(1, 8), c(0, 4.5), type= "n", xlab = "Length", ylab = "Width",
        main = "Petal and Sepal Dimensions in Iris Blossoms")
matpoints(iris[iS,c(1,3)], iris[iS,c(2,4)], pch = "sS", col = c(2,4))
matpoints(iris[iV,c(1,3)], iris[iV,c(2,4)], pch = "vV", col = c(2,4))
legend(1, 4, c("    Setosa Petals", "    Setosa Sepals",
              "Versicolor Petals", "Versicolor Sepals"),
      pch = "sSvV", col = rep(c(2,4), 2))

nam.var <- colnames(iris)[-5]
nam.spec <- as.character(iris[1+50*0:2, "Species"])
iris.S <- array(NA, dim = c(50,4,3), dimnames = list(NULL, nam.var, nam.spec))
for(i in 1:3) iris.S[, ,i] <- data.matrix(iris[1:50+50*(i-1), -5])

matplot(iris.S["Petal.Length",], iris.S["Petal.Width",], pch="SCV",
        col = rainbow(3, start = .8, end = .1),
        sub = paste(c("S", "C", "V"), dimnames(iris.S)[[3]],
                  sep = "=", collapse= "  "),
        main = "Fisher's Iris Data")
par(op)

```

---

mosaicplot

*Mosaic Plots*


---

## Description

Plots a mosaic on the current graphics device.

## Usage

```

mosaicplot(x, ...)

## Default S3 method:
mosaicplot(x, main = deparse(substitute(x)),
          sub = NULL, xlab = NULL, ylab = NULL,
          sort = NULL, off = NULL, dir = NULL,
          color = FALSE, shade = FALSE, margin = NULL,
          cex.axis = 0.66, las = par("las"),
          type = c("pearson", "deviance", "FT"), ...)

## S3 method for class 'formula':
mosaicplot(formula, data = NULL, ...,
          main = deparse(substitute(data)), subset,
          na.action = stats::na.omit)

```

**Arguments**

<code>x</code>	a contingency table in array form, with optional category labels specified in the <code>dimnames(x)</code> attribute. The table is best created by the <code>table()</code> command.
<code>main</code>	character string for the mosaic title.
<code>sub</code>	character string for the mosaic sub-title (at bottom).
<code>xlab, ylab</code>	x- and y-axis labels used for the plot; by default, the first and second element of <code>names(dimnames(X))</code> (i.e., the name of the first and second variable in <code>X</code> ).
<code>sort</code>	vector ordering of the variables, containing a permutation of the integers <code>1:length(dim(x))</code> (the default).
<code>off</code>	vector of offsets to determine percentage spacing at each level of the mosaic (appropriate values are between 0 and 20, and the default is 10 at each level). There should be one offset for each dimension of the contingency table.
<code>dir</code>	vector of split directions ("v" for vertical and "h" for horizontal) for each level of the mosaic, one direction for each dimension of the contingency table. The default consists of alternating directions, beginning with a vertical split.
<code>color</code>	logical or (recycling) vector of colors for color shading, used only when <code>shade</code> is <code>FALSE</code> . The default <code>color=FALSE</code> gives empty boxes with no shading.
<code>shade</code>	a logical indicating whether to produce extended mosaic plots, or a numeric vector of at most 5 distinct positive numbers giving the absolute values of the cut points for the residuals. By default, <code>shade</code> is <code>FALSE</code> , and simple mosaics are created. Using <code>shade = TRUE</code> cuts absolute values at 2 and 4.
<code>margin</code>	a list of vectors with the marginal totals to be fit in the log-linear model. By default, an independence model is fitted. See <a href="#">loglin</a> for further information.
<code>cex.axis</code>	The magnification to be used for axis annotation, as a multiple of <code>par("cex")</code> .
<code>las</code>	numeric; the style of axis labels, see <a href="#">par</a> .
<code>type</code>	a character string indicating the type of residual to be represented. Must be one of "pearson" (giving components of Pearson's $\chi^2$ ), "deviance" (giving components of the likelihood ratio $\chi^2$ ), or "FT" for the Freeman-Tukey residuals. The value of this argument can be abbreviated.
<code>formula</code>	a formula, such as <code>y ~ x</code> .
<code>data</code>	a data frame (or list), or a contingency table from which the variables in <code>formula</code> should be taken.
<code>...</code>	further arguments to be passed to or from methods.
<code>subset</code>	an optional vector specifying a subset of observations in the data frame to be used for plotting.
<code>na.action</code>	a function which indicates what should happen when the data contains variables to be cross-tabulated, and these variables contain NAs. The default is to omit cases which have an NA in any variable. Since the tabulation will omit all cases containing missing values, this will only be useful if the <code>na.action</code> function replaces missing values.

## Details

This is a generic function. It currently has a default method (`mosaicplot.default`) and a formula interface (`mosaicplot.formula`).

Extended mosaic displays show the standardized residuals of a loglinear model of the counts from by the color and outline of the mosaic's tiles. (Standardized residuals are often referred to a standard normal distribution.) Negative residuals are drawn in shaded of red and with broken outlines; positive ones are drawn in blue with solid outlines.

For the formula method, if `data` is an object inheriting from classes "table" or "ftable", or an array with more than 2 dimensions, it is taken as a contingency table, and hence all entries should be nonnegative. In this case, the left-hand side of `formula` should be empty, and the variables on the right-hand side should be taken from the names of the `dimnames` attribute of the contingency table. A marginal table of these variables is computed, and a mosaic of this table is produced.

Otherwise, `data` should be a data frame or matrix, list or environment containing the variables to be cross-tabulated. In this case, after possibly selecting a subset of the data as specified by the `subset` argument, a contingency table is computed from the variables given in `formula`, and a mosaic is produced from this.

See Emerson (1998) for more information and a case study with television viewer data from Nielsen Media Research.

Missing values are not supported except via an `na.action` function when `data` contains variables to be cross-tabulated.

## Author(s)

S-PLUS original by John Emerson (`emerson@stat.yale.edu`). Originally modified and enhanced for R by KH.

## References

Hartigan, J.A., and Kleiner, B. (1984) A mosaic of television ratings. *The American Statistician*, **38**, 32–35.

Emerson, J. W. (1998) Mosaic displays in S-PLUS: a general implementation and a case study. *Statistical Computing and Graphics Newsletter (ASA)*, **9**, 1, 17–23.

Friendly, M. (1994) Mosaic displays for multi-way contingency tables. *Journal of the American Statistical Association*, **89**, 190–200.

The home page of Michael Friendly (<http://www.math.yorku.ca/SCS/friendly.html>) provides information on various aspects of graphical methods for analyzing categorical data, including mosaic plots.

## See Also

[assocplot](#), [loglin](#).

## Examples

```
mosaicplot(Titanic, main = "Survival on the Titanic", color = TRUE)
## Formula interface for tabulated data:
```

```

mosaicplot(~ Sex + Age + Survived, data = Titanic, color = TRUE)

mosaicplot(HairEyeColor, shade = TRUE)
## Independence model of hair and eye color and sex. Indicates that
## there are significantly more blue eyed blonde females than expected
## in the case of independence (and too few brown eyed blonde females).

mosaicplot(HairEyeColor, shade = TRUE, margin = list(c(1,2), 3))
## Model of joint independence of sex from hair and eye color. Males
## are underrepresented among people with brown hair and eyes, and are
## overrepresented among people with brown hair and blue eyes, but not
## "significantly".

## Formula interface for raw data: visualize crosstabulation of numbers
## of gears and carburettors in Motor Trend car data.
mosaicplot(~ gear + carb, data = mtcars, color = TRUE, las = 1)
# color recycling
mosaicplot(~ gear + carb, data = mtcars, color = 2:3, las = 1)

```

---

mtext

---

*Write Text into the Margins of a Plot*


---

## Description

Text is written in one of the four margins of the current figure region or one of the outer margins of the device region.

## Usage

```

mtext(text, side = 3, line = 0, outer = FALSE, at = NA,
      adj = NA, padj = NA, cex = NA, col = NA, font = NA, vfont = NULL, ...)

```

## Arguments

text	one or more character strings or expressions.
side	on which side of the plot (1=bottom, 2=left, 3=top, 4=right).
line	on which MARGin line, starting at 0 counting outwards.
outer	use outer margins if available.
at	give location in user-coordinates. If <code>length(at)==0</code> (the default), the location will be determined by <code>adj</code> .
adj	adjustment for each string in reading direction. For strings parallel to the axes, <code>adj=0</code> means left or bottom alignment, and <code>adj=1</code> means right or top alignment. If <code>adj</code> is not a finite value (the default), the value of <code>par("las")</code> determines the adjustment. For strings plotted parallel to the axis the default is to centre the string.

<code>padj</code>	adjustment for each string perpendicular to the reading direction (which is controlled by <code>adj</code> ). For strings parallel to the axes, <code>padj=0</code> means right or top alignment, and <code>padj=1</code> means left or bottom alignment. If <code>padj</code> is not a finite value (the default), the value of <code>par("las")</code> determines the adjustment. For strings plotted perpendicular to the axis the default is to centre the string.
<code>...</code>	Further graphical parameters (see <a href="#">text</a> and <a href="#">par</a> ); currently supported are:
<code>cex</code>	character expansion factor (default = 1).
<code>col</code>	color to use.
<code>font</code>	font for text.
<code>vfont</code>	vector font for text.

### Details

The “user coordinates” in the outer margins always range from zero to one, and are not affected by the user coordinates in the figure region(s) — R is differing here from other implementations of S.

The arguments `side`, `line`, `at`, `adj`, the further graphical parameters and even `outer` can be vectors, and recycling will take place to plot as many strings as the longest of the vector arguments. Note that a vector `adj` has a different meaning from [text](#).

`adj = 0.5` will centre the string, but for `outer=TRUE` on the device region rather than the plot region.

Parameter `las` will determine the orientation of the string(s). For strings plotted perpendicular to the axis the default justification is to place the end of the string nearest the axis on the specified line. (Note that this differs from S, which uses `srt` if `at` is supplied and `las` if it is not.)

Note that if the text is to be plotted perpendicular to the axis, `adj` determines the justification of the string *and* the position along the axis unless `at` is specified.

### Side Effects

The given text is written onto the current plot.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[title](#), [text](#), [plot](#), [par](#); [plotmath](#) for details on mathematical annotation.

### Examples

```
plot(1:10, (-4:5)^2, main="Parabola Points", xlab="xlab")
mtext("10 of them")
for(s in 1:4)
  mtext(paste("mtext(..., line= -1, {side, col, font} = ", s,
             ", cex = ", (1+s)/2, ")", line = -1,
```

```

        side=s, col=s, font=s, cex= (1+s)/2)
mtext("mtext(..., line= -2)", line = -2)
mtext("mtext(..., line= -2, adj = 0)", line = -2, adj =0)
##--- log axis :
plot(1:10, exp(1:10), log='y', main="log='y'", xlab="xlab")
for(s in 1:4) mtext(paste("mtext(...,side=",s,")"), side=s)

```

---

n2mfrow

---

*Compute Default mfrow From Number of Plots*


---

### Description

Easy setup for plotting multiple figures (in a rectangular layout) on one page. This computes a sensible default for [par](#)(mfrow).

### Usage

```
n2mfrow(nr.plots)
```

### Arguments

nr.plots      integer; the number of plot figures you'll want to draw.

### Value

A length two integer vector nr, nc giving the number of rows and columns, fulfilling nr >= nc >= 1 and nr \* nc >= nr.plots.

### Author(s)

Martin Maechler

### See Also

[par](#), [layout](#).

### Examples

```

n2mfrow(8) # 3 x 3

n <- 5 ; x <- seq(-2,2, len=51)
## suppose now that 'n' is not known {inside function}
op <- par(mfrow = n2mfrow(n))
for (j in 1:n)
  plot(x, x^j, main = substitute(x^ exp, list(exp = j)), type = "l",
       col = "blue")

sapply(1:10, n2mfrow)

```

---

`nclass`*Compute the Number of Classes for a Histogram*

---

## Description

Compute the number of classes for a histogram, for use internally in `hist`.

## Usage

```
nclass.Sturges(x)
nclass.scott(x)
nclass.FD(x)
```

## Arguments

`x`                    A data vector.

## Details

`nclass.Sturges` uses Sturges' formula, implicitly basing bin sizes on the range of the data.

`nclass.scott` uses Scott's choice for a normal distribution based on the estimate of the standard error.

`nclass.FD` uses the Freedman-Diaconis choice based on the inter-quartile range.

## Value

The suggested number of classes.

## References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S-PLUS*. Springer, page 112.

Freedman, D. and Diaconis, P. (1981) On the histogram as a density estimator:  $L_2$  theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete* **57**, 453–476.

Scott, D. W. (1979) On optimal and data-based histograms. *Biometrika* **66**, 605–610.

Scott, D. W. (1992) *Multivariate Density Estimation. Theory, Practice, and Visualization*. Wiley.

## See Also

`hist`

pairs

*Scatterplot Matrices***Description**

A matrix of scatterplots is produced.

**Usage**

```

pairs(x, ...)

## S3 method for class 'formula':
pairs(formula, data = NULL, ..., subset, na.action = na.pass)

## Default S3 method:
pairs(x, labels, panel = points, ...,
      lower.panel = panel, upper.panel = panel,
      diag.panel = NULL, text.panel = textPanel,
      label.pos = 0.5 + has.diag/3,
      cex.labels = NULL, font.labels = 1,
      rowlaptop = TRUE, gap = 1)

```

**Arguments**

<code>x</code>	the coordinates of points given as columns of a matrix.
<code>formula</code>	a formula, such as $\sim x + y + z$ . Each term will give a separate variable in the pairs plot, so terms should be numeric vectors. (A response will be interpreted as another variable, but not treated specially, so it is confusing to use one.)
<code>data</code>	a data.frame (or list) from which the variables in <code>formula</code> should be taken.
<code>subset</code>	an optional vector specifying a subset of observations to be used for plotting.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. The default is to pass missing values on to the panel functions, but <code>na.action = na.omit</code> will cause cases with missing values in any of the variables to be omitted entirely.
<code>labels</code>	the names of the variables.
<code>panel</code>	<code>function(x, y, ...)</code> which is used to plot the contents of each panel of the display.
<code>...</code>	graphical parameters can be given as arguments to <code>plot</code> .
<code>lower.panel</code> , <code>upper.panel</code>	separate panel functions to be used below and above the diagonal respectively.
<code>diag.panel</code>	optional <code>function(x, ...)</code> to be applied on the diagonals.
<code>text.panel</code>	optional <code>function(x, y, labels, cex, font, ...)</code> to be applied on the diagonals.

`label.pos`      y position of labels in the text panel.  
`cex.labels, font.labels`  
                   graphics parameters for the text panel.  
`rowlaptop`      logical. Should the layout be matrix-like with row 1 at the top, or graph-like with row 1 at the bottom?  
`gap`              Distance between subplots, in margin lines.

### Details

The  $ij$ th scatterplot contains  $x[, i]$  plotted against  $x[, j]$ . The “scatterplot” can be customised by setting panel functions to appear as something completely different. The off-diagonal panel functions are passed the appropriate columns of  $x$  as  $x$  and  $y$ : the diagonal panel function (if any) is passed a single column, and the `text.panel` function is passed a single  $(x, y)$  location and the column name.

The graphical parameters `pch` and `col` can be used to specify a vector of plotting symbols and colors to be used in the plots.

The graphical parameter `oma` will be set by `pairs.default` unless supplied as an argument.

A panel function should not attempt to start a new plot, but just plot within a given coordinate system: thus `plot` and `boxplot` are not panel functions.

By default, missing values are passed to the panel functions and will often be ignored within a panel. However, for the formula method and `na.action = na.omit`, all cases which contain a missing values for any of the variables are omitted completely (including when the scales are selected). (The latter was the default behaviour prior to R 2.0.0.)

### Author(s)

Enhancements for R 1.0.0 contributed by Dr. Jens Oehlschlaegel-Akiyoshi and R-core members.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### Examples

```

pairs(iris[1:4], main = "Anderson's Iris Data -- 3 species",
      pch = 21, bg = c("red", "green3", "blue")[unclass(iris$Species)])

## formula method
pairs(~ Fertility + Education + Catholic, data = swiss,
      subset = Education < 20, main = "Swiss data, Education < 20")

pairs(USJudgeRatings)

## put histograms on the diagonal
panel.hist <- function(x, ...)
{
  usr <- par("usr"); on.exit(par(usr))

```

```

par(usr = c(usr[1:2], 0, 1.5) )
h <- hist(x, plot = FALSE)
breaks <- h$breaks; nB <- length(breaks)
y <- h$counts; y <- y/max(y)
rect(breaks[-nB], 0, breaks[-1], y, col="cyan", ...)
}
pairs(USJudgeRatings[1:5], panel=panel.smooth,
      cex = 1.5, pch = 24, bg="light blue",
      diag.panel=panel.hist, cex.labels = 2, font.labels=2)

## put (absolute) correlations on the upper panels,
## with size proportional to the correlations.
panel.cor <- function(x, y, digits=2, prefix="", cex.cor)
{
  usr <- par("usr"); on.exit(par(usr))
  par(usr = c(0, 1, 0, 1))
  r <- abs(cor(x, y))
  txt <- format(c(r, 0.123456789), digits=digits)[1]
  txt <- paste(prefix, txt, sep="")
  if(missing(cex.cor)) cex <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = cex * r)
}
pairs(USJudgeRatings, lower.panel=panel.smooth, upper.panel=panel.cor)

```

---

panel.smooth

*Simple Panel Plot*


---

## Description

An example of a simple useful panel function to be used as argument in e.g., [coplot](#) or [pairs](#).

## Usage

```

panel.smooth(x, y, col = par("col"), bg = NA, pch = par("pch"),
             cex = 1, col.smooth = "red", span = 2/3, iter = 3,
             ...)

```

## Arguments

<code>x, y</code>	numeric vectors of the same length
<code>col, bg, pch, cex</code>	numeric or character codes for the color(s), point type and size of <a href="#">points</a> ; see also <a href="#">par</a> .
<code>col.smooth</code>	color to be used by lines for drawing the smooths.
<code>span</code>	smoothing parameter <code>f</code> for <a href="#">lowess</a> , see there.
<code>iter</code>	number of robustness iterations for <a href="#">lowess</a> .
<code>...</code>	further arguments to <a href="#">lines</a> .

**See Also**

`coplot` and `pairs` where `panel.smooth` is typically used; `lowess`.

**Examples**

```
pairs(swiss, panel = panel.smooth, pch = ".")# emphasize the smooths
pairs(swiss, panel = panel.smooth, lwd = 2, cex= 1.5, col="blue")# hmm...
```

---

 par

*Set or Query Graphical Parameters*


---

**Description**

`par` can be used to set or query graphical parameters. Parameters can be set by specifying them as arguments to `par` in `tag = value` form, or by passing them as a list of tagged values.

**Usage**

```
par(..., no.readonly = FALSE)

<highlevel plot> (... , <tag> = <value>)
```

**Arguments**

`...` arguments in `tag = value` form, or a list of tagged values. The tags must come from the graphical parameters described below.

`no.readonly` logical; if TRUE and there are no other arguments, only parameters are returned which can be set by a subsequent `par()` call.

**Details**

Parameters are queried by giving one or more character vectors to `par`.

`par()` (no arguments) or `par(no.readonly=TRUE)` is used to get *all* the graphical parameters (as a named list). Their names are currently taken from the variable `.Pars`. `.Pars.readonly` contains the names of the `par` arguments which are *readonly*.

**R.O.** indicates *read-only arguments*: These may only be used in queries, i.e., they do *not* set anything.

All but these **R.O.** and the following *low-level arguments* can be set as well in high-level and mid-level plot functions, such as `plot`, `points`, `lines`, `axis`, `title`, `text`, `mtext`:

- "ask"
- "family", "fig", "fin"
- "lend", "lheight", "ljoin", "lmitre"
- "mai", "mar", "mex"
- "mfrow", "mfcoll", "mfg"

- "new"
- "oma", "omd", "omi"
- "pin", "plt", "ps", "pty"
- "usr"
- "xlog", "ylog"

### Value

When parameters are set, their former values are returned in an invisible named list. Such a list can be passed as an argument to `par` to restore the parameter values. Use `par(no.readonly = TRUE)` for the full list of parameters that can be restored.

When just one parameter is queried, the value is a character string. When two or more parameters are queried, the result is a list of character strings, with the list names giving the parameters.

Note the inconsistency: setting one parameter returns a list, but querying one parameter returns a vector.

### Graphical Parameters

**adj** The value of `adj` determines the way in which text strings are justified. A value of 0 produces left-justified text, 0.5 centered text and 1 right-justified text. (Any value in  $[0, 1]$  is allowed, and on most devices values outside that interval will also work.) Note that the `adj` argument of `text` also allows `adj = c(x, y)` for different adjustment in x- and y- direction.

**ann** If set to `FALSE`, high-level plotting functions do not annotate the plots they produce with axis and overall titles. The default is to do annotation.

**ask** logical. If `TRUE`, the user is asked for input, before a new figure is drawn.

**bg** The color to be used for the background of plots. A description of how colors are specified is given below.

**bty** A character string which determined the type of box which is drawn about plots. If `bty` is one of "o", "l", "7", "c", "u", or "]" the resulting box resembles the corresponding upper case letter. A value of "n" suppresses the box.

**cex** A numerical value giving the amount by which plotting text and symbols should be scaled relative to the default.

**cex.axis** The magnification to be used for axis annotation relative to the current.

**cex.lab** The magnification to be used for x and y labels relative to the current.

**cex.main** The magnification to be used for main titles relative to the current.

**cex.sub** The magnification to be used for sub-titles relative to the current.

**cin** *R.O.*; character size (`width,height`) in inches.

**col** A specification for the default plotting color. A description of how colors are specified is given below.

**col.axis** The color to be used for axis annotation.

**col.lab** The color to be used for x and y labels.

**col.main** The color to be used for plot main titles.

- col.sub** The color to be used for plot sub-titles.
- cra** *R.O.*; size of default character (`width,height`) in “rasters” (pixels).
- crt** A numerical value specifying (in degrees) how single characters should be rotated. It is unwise to expect values other than multiples of 90 to work. Compare with `srt` which does string rotation.
- csi** *R.O.*; height of (default sized) characters in inches.
- cxy** *R.O.*; size of default character (`width,height`) in user coordinate units. `par("cxy")` is `par("cin")/par("pin")` scaled to user coordinates. Note that `c(strwidth(ch), strwidth(ch))` for a given string `ch` is usually much more precise.
- din** *R.O.*; the device dimensions, (`width,height`), in inches.
- err** (*Unimplemented*; `R` is silent when points outside the plot region are *not* plotted.) The degree of error reporting desired.
- family** The name of a font family for drawing text. This name is device-independent and gets mapped by each graphics device to a device-specific font description. The default value is "" which means that the default device font will be used. Standard values are "serif", "sans", "mono", and "symbol". Different devices may define others. Some devices will ignore this setting completely.
- fg** The color to be used for the foreground of plots. This is the default color used for things like axes and boxes around plots. A description of how colors are specified is given below.
- fig** A numerical vector of the form `c(x1, x2, y1, y2)` which gives the (NDC) coordinates of the figure region in the display region of the device. If you set this, unlike `S`, you start a new plot, so to add to an existing plot use `new=TRUE` as well.
- fin** The figure region dimensions, (`width,height`), in inches. If you set this, unlike `S`, you start a new plot.
- font** An integer which specifies which font to use for text. If possible, device drivers arrange so that 1 corresponds to plain text, 2 to bold face, 3 to italic and 4 to bold italic.
- font.axis** The font to be used for axis annotation.
- font.lab** The font to be used for x and y labels.
- font.main** The font to be used for plot main titles.
- font.sub** The font to be used for plot sub-titles.
- gamma** the gamma correction, see argument `gamma` to `hsv`.
- lab** A numerical vector of the form `c(x, y, len)` which modifies the way that axes are annotated. The values of `x` and `y` give the (approximate) number of tickmarks on the x and y axes and `len` specifies the label size. The default is `c(5, 5, 7)`. *Currently, len is unimplemented.*
- las** numeric in {0,1,2,3}; the style of axis labels.
- 0:** always parallel to the axis [*default*],
  - 1:** always horizontal,
  - 2:** always perpendicular to the axis,
  - 3:** always vertical.
- Note that other string/character rotation (via argument `srt` to `par`) does *not* affect the axis labels.

- lend** The line end style. This can be specified as an integer or string: 0 and "round" mean rounded line caps; 1 and "butt" mean butt line caps; 2 and "square" mean square line caps.
- lheight** The line height multiplier. The height of a line of text (used to vertically space multi-line text) is found by multiplying the current font size both by the current character expansion and by the line height multiplier. Default value is 1.
- ljoin** The line join style. This can be specified as an integer or string: 0 and "round" mean rounded line joins; 1 and "mitre" mean mitred line joins; 2 and "bevel" mean bevelled line joins.
- lmitre** The line mitre limit. This controls when mitred line joins are automatically converted into bevelled line joins. The value must be larger than 1 and the default is 10. Not all devices will honour this setting.
- lty** The line type. Line types can either be specified as an integer (0=blank, 1=solid, 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash", where "blank" uses 'invisible lines' (i.e., doesn't draw them).  
Alternatively, a string of up to 8 characters (from c(1:9, "A":"F")) may be given, giving the length of line segments which are alternatively drawn and skipped. See section 'Line Type Specification' below.
- lwd** The line width, a *positive* number, defaulting to 1.
- mai** A numerical vector of the form c(bottom, left, top, right) which gives the margin size specified in inches.
- mar** A numerical vector of the form c(bottom, left, top, right) which gives the number of lines of margin to be specified on the four sides of the plot. The default is c(5, 4, 4, 2) + 0.1.
- mex** mex is a character size expansion factor which is used to describe coordinates in the margins of plots. Note that this does not change the font size, rather specifies the size of font used to convert between mar and mai, and between oma and omi.
- mfc col, mfrow** A vector of the form c(nr, nc). Subsequent figures will be drawn in an nr-by-nc array on the device by *columns* (mfc col), or *rows* (mfrow), respectively.  
In a layout with exactly two rows and columns the base value of "cex" is reduced by a factor of 0.83: if there are three or more of either rows or columns, the reduction factor is 0.66.  
Consider the alternatives, [layout](#) and [split.screen](#).
- mfg** A numerical vector of the form c(i, j) where i and j indicate which figure in an array of figures is to be drawn next (if setting) or is being drawn (if enquiring). The array must already have been set by mfc col or mfrow.  
For compatibility with S, the form c(i, j, nr, nc) is also accepted, when nr and nc should be the current number of rows and number of columns. Mismatches will be ignored, with a warning.
- mgp** The margin line (in mex units) for the axis title, axis labels and axis line. The default is c(3, 1, 0).
- mkh** The height in inches of symbols to be drawn when the value of pch is an integer. *Completely ignored currently.*
- new** logical, defaulting to FALSE. If set to TRUE, the next high-level plotting command (actually [plot.new](#)) should *not clean* the frame before drawing "as if it was on a *new* device".

- oma** A vector of the form `c(bottom, left, top, right)` giving the size of the outer margins in lines of text.
- omd** A vector of the form `c(x1, x2, y1, y2)` giving the outer margin region in NDC (= normalized device coordinates), i.e., as fraction (in  $[0, 1]$ ) of the device region.
- omi** A vector of the form `c(bottom, left, top, right)` giving the size of the outer margins in inches.
- pch** Either an integer specifying a symbol or a single character to be used as the default in plotting points.
- pin** The current plot dimensions, `(width,height)`, in inches.
- plt** A vector of the form `c(x1, x2, y1, y2)` giving the coordinates of the plot region as fractions of the current figure region.
- ps** integer; the pointsize of text and symbols.
- pty** A character specifying the type of plot region to be used; "s" generates a square plotting region and "m" generates the maximal plotting region.
- smo** (*Unimplemented*) a value which indicates how smooth circles and circular arcs should be.
- srt** The string rotation in degrees. See the comment about `crt`.
- tck** The length of tick marks as a fraction of the smaller of the width or height of the plotting region. If `tck >= 0.5` it is interpreted as a fraction of the relevant side, so if `tck=1` grid lines are drawn. The default setting (`tck = NA`) is to use `tcl = -0.5` (see below).
- tcl** The length of tick marks as a fraction of the height of a line of text. The default value is `-0.5`; setting `tcl = NA` sets `tck = -0.01` which is S' default.
- tmag** A number specifying the enlargement of text of the main title relative to the other annotating text of the plot.
- type** character; the default plot type desired, see `plot.default(type=...)`, defaulting to "p".
- usr** A vector of the form `c(x1, x2, y1, y2)` giving the extremes of the user coordinates of the plotting region. When a logarithmic scale is in use (i.e., `par("xlog")` is true, see below), then the x-limits will be  $10^{\text{par("usr")[1:2]}}$ . Similarly for the y-axis.
- xaxp** A vector of the form `c(x1, x2, n)` giving the coordinates of the extreme tick marks and the number of intervals between tick-marks when `par("xlog")` is false. Otherwise, when *log* coordinates are active, the three values have a different meaning: For a small range, `n` is *negative*, and the ticks are as in the linear case, otherwise, `n` is in `1:3`, specifying a case number, and `x1` and `x2` are the lowest and highest power of 10 inside the user coordinates,  $10^{\text{par("usr")[1:2]}}$ . (The "usr" coordinates are log10-transformed here!)
- n=1** will produce tick marks at  $10^j$  for integer  $j$ ,
- n=2** gives marks  $k10^j$  with  $k \in \{1, 5\}$ ,
- n=3** gives marks  $k10^j$  with  $k \in \{1, 2, 5\}$ .
- See `axTicks()` for a pure R implementation of this.
- xaxs** The style of axis interval calculation to be used for the x-axis. Possible values are "r", "i", "e", "s", "d". The styles are generally controlled by the range of data or `xlim`, if given. Style "r" (regular) first extends the data range by 4 percent and then finds an axis with pretty labels that fits within the range. Style "i" (internal) just finds an axis with pretty labels that

fits within the original data range. Style "s" (standard) finds an axis with pretty labels within which the original data range fits. Style "e" (extended) is like style "s", except that it is also ensured that there is room for plotting symbols within the bounding box. Style "d" (direct) specifies that the current axis should be used on subsequent plots. (*Only "x" and "i" styles are currently implemented*)

**xaxt** A character which specifies the axis type. Specifying "n" causes an axis to be set up, but not plotted. The standard value is "s": for compatibility with S values "l" and "e" are accepted but are equivalent to "s".

**xlog** logical value (see `log` in `plot.default`). If TRUE, a logarithmic scale is in use (e.g., after `plot(*, log = "x")`). For a new device, it defaults to FALSE, i.e., linear scale.

**xpd** A logical value or NA. If FALSE, all plotting is clipped to the plot region, if TRUE, all plotting is clipped to the figure region, and if NA, all plotting is clipped to the device region.

**yaxp** A vector of the form `c(y1, y2, n)` giving the coordinates of the extreme tick marks and the number of intervals between tick-marks unless for log coordinates, see `xaxp` above.

**yaxs** The style of axis interval calculation to be used for the y-axis. See `xaxs` above.

**yaxt** A character which specifies the axis type. Specifying "n" causes an axis to be set up, but not plotted.

**ylog** a logical value; see `xlog` above.

### Color Specification

Colors can be specified in several different ways. The simplest way is with a character string giving the color name (e.g., "red"). A list of the possible colors can be obtained with the function `colors`. Alternatively, colors can be specified directly in terms of their RGB components with a string of the form "#RRGGBB" where each of the pairs RR, GG, BB consist of two hexadecimal digits giving a value in the range 00 to FF. Colors can also be specified by giving an index into a small table of colors, the `palette`. This provides compatibility with S. Index 0 corresponds to the background color.

Additionally, "transparent" or (integer) NA is *transparent*, useful for filled areas (such as the background!), and just invisible for things like lines or text.

The functions `rgb`, `hsv`, `gray` and `rainbow` provide additional ways of generating colors.

### Line Type Specification

Line types can either be specified by giving an index into a small built in table of line types (1 = solid, 2 = dashed, etc, see `lty` above) or directly as the lengths of on/off stretches of line. This is done with a string of an even number (up to eight) of characters, namely non-zero (hexadecimal) digits which give the lengths in consecutive positions in the string. For example, the string "33" specifies three units on followed by three off and "3313" specifies three units on followed by three off followed by one on and finally three off. The 'units' here are (on most devices) proportional to `lwd`, and with `lwd = 1` are in pixels or points.

The five standard dash-dot line types (`lty = 2:6`) correspond to `c("44", "13", "1343", "73", "2262")`.

Note that NA is not a valid value for `lty`.

**Note**

The effect of restoring all the (settable) graphics parameters as in the examples is hard to predict if the device has been resized. Several of them are attempting to set the same things in different ways, and those last in the alphabet will win. In particular, the settings of `mai`, `mar`, `pin`, `plt` and `pty` interact, as do the outer margin settings, the figure layout and figure region size.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`plot.default` for some high-level plotting parameters; `colors`, `gray`, `rainbow`, `rgb`; `options` for other setup parameters; graphic devices `x11`, `postscript` and setting up device regions by `layout` and `split.screen`.

**Examples**

```
op <- par(mfrow = c(2, 2), # 2 x 2 pictures on one plot
         pty = "s")      # square plotting region,
                        # independent of device size

## At end of plotting, reset to previous settings:
par(op)

## Alternatively,
op <- par(no.readonly = TRUE) # the whole list of settable par's.
## do lots of plotting and par(.) calls, then reset:
par(op)

par("ylog") # FALSE
plot(1 : 12, log = "y")
par("ylog") # TRUE

plot(1:2, xaxs = "i") # 'inner axis' w/o extra space
stopifnot(par("xaxp")[1:2] == 1:2 &&
          par("usr") [1:2] == 1:2)

( nr.prof <-
  c(prof.pilots=16,lawyers=11,farmers=10,salesmen=9,physicians=9,
    mechanics=6,policemen=6,managers=6,engineers=5,teachers=4,
    housewives=3,students=3,armed.forces=1))
par(las = 3)
barplot(rbind(nr.prof)) # R 0.63.2: shows alignment problem
par(las = 0)# reset to default

## 'fg' use:
plot(1:12, type = "b", main="'fg' : axes, ticks and box in gray",
     fg = gray(0.7), bty="7" , sub=R.version.string)
```

```

ex <- function() {
  old.par <- par(no.readonly = TRUE) # all par settings which
                                     # could be changed.

  on.exit(par(old.par))
  ## ...
  ## ... do lots of par() settings and plots
  ## ...
  invisible() #-- now, par(old.par) will be executed
}
ex()

```

---

persp

*Perspective Plots*

---

### Description

This function draws perspective plots of surfaces over the x–y plane. `persp` is a generic function.

### Usage

```
persp(x, ...)
```

```
## Default S3 method:
```

```

persp(x = seq(0, 1, len = nrow(z)), y = seq(0, 1, len = ncol(z)), z,
      xlim = range(x), ylim = range(y), zlim = range(z, na.rm = TRUE),
      xlab = NULL, ylab = NULL, zlab = NULL, main = NULL, sub = NULL,
      theta = 0, phi = 15, r = sqrt(3), d = 1, scale = TRUE,
      expand = 1, col = "white", border = NULL, ltheta = -135, lphi = 0,
      shade = NA, box = TRUE, axes = TRUE, nticks = 5,
      ticktype = "simple", ...)

```

### Arguments

<code>x</code> , <code>y</code>	locations of grid lines at which the values in <code>z</code> are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If <code>x</code> is a list, its components <code>x\$x</code> and <code>x\$y</code> are used for <code>x</code> and <code>y</code> , respectively.
<code>z</code>	a matrix containing the values to be plotted (NAs are allowed). Note that <code>x</code> can be used instead of <code>z</code> for convenience.
<code>xlim</code> , <code>ylim</code> , <code>zlim</code>	x-, y- and z-limits. The plot is produced so that the rectangular volume defined by these limits is visible.
<code>xlab</code> , <code>ylab</code> , <code>zlab</code>	titles for the axes. N.B. These must be character strings; expressions are not accepted. Numbers will be coerced to character strings.
<code>main</code> , <code>sub</code>	main and sub title, as for <a href="#">title</a> .
<code>theta</code> , <code>phi</code>	angles defining the viewing direction. <code>theta</code> gives the azimuthal direction and <code>phi</code> the colatitude.

<code>r</code>	the distance of the eyepoint from the centre of the plotting box.
<code>d</code>	a value which can be used to vary the strength of the perspective transformation. Values of <code>d</code> greater than 1 will lessen the perspective effect and values less and 1 will exaggerate it.
<code>scale</code>	before viewing the <code>x</code> , <code>y</code> and <code>z</code> coordinates of the points defining the surface are transformed to the interval <code>[0,1]</code> . If <code>scale</code> is <code>TRUE</code> the <code>x</code> , <code>y</code> and <code>z</code> coordinates are transformed separately. If <code>scale</code> is <code>FALSE</code> the coordinates are scaled so that aspect ratios are retained. This is useful for rendering things like DEM information.
<code>expand</code>	a expansion factor applied to the <code>z</code> coordinates. Often used with <code>0 &lt; expand &lt; 1</code> to shrink the plotting box in the <code>z</code> direction.
<code>col</code>	the color(s) of the surface facets. Transparent colours are ignored. This is recycled to the $(nx - 1)(ny - 1)$ facets.
<code>border</code>	the color of the line drawn around the surface facets. A value of <code>NA</code> will disable the drawing of borders. This is sometimes useful when the surface is shaded.
<code>ltheta</code> , <code>lphi</code>	if finite values are specified for <code>ltheta</code> and <code>lphi</code> , the surface is shaded as though it was being illuminated from the direction specified by azimuth <code>ltheta</code> and colatitude <code>lphi</code> .
<code>shade</code>	the shade at a surface facet is computed as $((1+d)/2)^{\text{shade}}$ , where <code>d</code> is the dot product of a unit vector normal to the facet and a unit vector in the direction of a light source. Values of <code>shade</code> close to one yield shading similar to a point light source model and values close to zero produce no shading. Values in the range 0.5 to 0.75 provide an approximation to daylight illumination.
<code>box</code>	should the bounding box for the surface be displayed. The default is <code>TRUE</code> .
<code>axes</code>	should ticks and labels be added to the box. The default is <code>TRUE</code> . If <code>box</code> is <code>FALSE</code> then no ticks or labels are drawn.
<code>ticktype</code>	character: <code>"simple"</code> draws just an arrow parallel to the axis to indicate direction of increase; <code>"detailed"</code> draws normal ticks as per 2D plots.
<code>nticks</code>	the (approximate) number of tick marks to draw on the axes. Has no effect if <code>ticktype</code> is <code>"simple"</code> .
<code>...</code>	additional graphical parameters (see <a href="#">par</a> ).

## Details

The plots are produced by first transforming the coordinates to the interval `[0,1]`. The surface is then viewed by looking at the origin from a direction defined by `theta` and `phi`. If `theta` and `phi` are both zero the viewing direction is directly down the negative `y` axis. Changing `theta` will vary the azimuth and changing `phi` the colatitude.

There is a hook called `"persp"` (see [setHook](#)) called after the plot is completed, which is used in the testing code to annotate the plot page. The hook function(s) are called with no argument.

Notice that `persp` interprets the `z` matrix as a table of  $f(x[i], y[j])$  values, so that the `x` axis corresponds to row number and the `y` axis to column number, with column 1 at the bottom, so that with the standard rotation angles, the top left corner of the matrix is displayed at the left hand side, closest to the user.

**Value**

The *viewing transformation matrix*, say  $VT$ , a  $4 \times 4$  matrix suitable for projecting 3D coordinates  $(x, y, z)$  into the 2D plane using homogenous 4D coordinates  $(x, y, z, t)$ . It can be used to superimpose additional graphical elements on the 3D plot, by `lines()` or `points()`, e.g. using the function `trans3d` given in the last examples section below.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`contour` and `image`.

**Examples**

```
## More examples in demo(persp) !!
## -----

# (1) The Obligatory Mathematical surface.
#     Rotated sinc function.

x <- seq(-10, 10, length= 30)
y <- x
f <- function(x,y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
z <- outer(x, y, f)
z[is.na(z)] <- 1
op <- par(bg = "white")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue",
      ltheta = 120, shade = 0.75, ticktype = "detailed",
      xlab = "X", ylab = "Y", zlab = "Sinc( r )"
) -> res
round(res, 3)

# (2) Add to existing persp plot :

trans3d <- function(x,y,z, pmat) {
  tr <- cbind(x,y,z,1) %*% pmat
  list(x = tr[,1]/tr[,4], y= tr[,2]/tr[,4])
}
xE <- c(-10,10); xy <- expand.grid(xE, xE)
points(trans3d(xy[,1], xy[,2], 6, pm = res), col = 2, pch =16)
lines (trans3d(x, y=10, z= 6 + sin(x), pm = res), col = 3)

phi <- seq(0, 2*pi, len = 201)
r1 <- 7.725 # radius of 2nd maximum
xr <- r1 * cos(phi)
yr <- r1 * sin(phi)
lines(trans3d(xr,yr, f(xr,yr), res), col = "pink", lwd=2)## (no hidden lines)
```

```
# (3) Visualizing a simple DEM model

z <- 2 * volcano      # Exaggerate the relief
x <- 10 * (1:nrow(z)) # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z)) # 10 meter spacing (E to W)
## Don't draw the grid lines : border = NA
par(bg = "slategray")
persp(x, y, z, theta = 135, phi = 30, col = "green3", scale = FALSE,
      ltheta = -120, shade = 0.75, border = NA, box = FALSE)
par(op)
```

---

 pie

*Pie Charts*


---

### Description

Draw a pie chart.

### Usage

```
pie(x, labels = names(x), edges = 200, radius = 0.8,
    density = NULL, angle = 45, col = NULL, border = NULL,
    lty = NULL, main = NULL, ...)
```

### Arguments

<code>x</code>	a vector of positive quantities. The values in <code>x</code> are displayed as the areas of pie slices.
<code>labels</code>	a vector of character strings giving names for the slices. For empty or NA labels, no pointing line is drawn either.
<code>edges</code>	the circular outline of the pie is approximated by a polygon with this many edges.
<code>radius</code>	the pie is drawn centered in a square box whose sides range from $-1$ to $1$ . If the character strings labeling the slices are long it may be necessary to use a smaller radius.
<code>density</code>	the density of shading lines, in lines per inch. The default value of <code>NULL</code> means that no shading lines are drawn. Non-positive values of <code>density</code> also inhibit the drawing of shading lines.
<code>angle</code>	the slope of shading lines, given as an angle in degrees (counter-clockwise).
<code>col</code>	a vector of colors to be used in filling or shading the slices. If missing a set of 6 pastel colours is used, unless <code>density</code> is specified when <code>par("fg")</code> is used.
<code>border, lty</code>	(possibly vectors) arguments passed to <code>polygon</code> which draws each slice.
<code>main</code>	an overall title for the plot.
<code>...</code>	graphical parameters can be given as arguments to <code>pie</code> . They will affect the main title and labels only.

**Note**

Pie charts are a very bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas. A bar chart or dot chart is a preferable way of displaying this type of data.

Cleveland (1985), page 264: “Data that can be shown by pie charts always can be shown by a dot chart. This means that judgements of position along a common scale can be made instead of the less accurate angle judgements.” This statement is based on the empirical investigations of Cleveland and McGill as well as investigations by perceptual psychologists.

Prior to R 1.5.0 this was known as `piechart`, which is the name of a Trellis function, so the name was changed to be compatible with S.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Cleveland, W. S. (1985) *The elements of graphing data*. Wadsworth: Monterey, CA, USA.

**See Also**

[dotchart](#).

**Examples**

```
pie(rep(1, 24), col = rainbow(24), radius = 0.9)

pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
names(pie.sales) <- c("Blueberry", "Cherry",
  "Apple", "Boston Cream", "Other", "Vanilla Cream")
pie(pie.sales) # default colours
pie(pie.sales,
  col = c("purple", "violetred1", "green3", "cornsilk", "cyan", "white"))
pie(pie.sales, col = gray(seq(0.4,1.0,length=6)))
pie(pie.sales, density = 10, angle = 15 + 10 * 1:6)

n <- 200
pie(rep(1,n), labels="", col=rainbow(n), border=NA,
  main = "pie(*, labels=\"\", col=rainbow(n), border=NA,..")
```

---

 plot

*Generic X-Y Plotting*


---

**Description**

Generic function for plotting of R objects. For more details about the graphical parameter arguments, see [par](#).

**Usage**

```
plot(x, y, ...)
```

**Arguments**

x	the coordinates of points in the plot. Alternatively, a single plotting structure, function or <i>any R object with a plot method</i> can be provided.
y	the y coordinates of points in the plot, <i>optional</i> if x is an appropriate structure.
...	graphical parameters can be given as arguments to <code>plot</code> . Many methods will also accept the following arguments:
type	<p>what type of plot should be drawn. Possible types are</p> <ul style="list-style-type: none"> <li>• "p" for <b>p</b>oints,</li> <li>• "l" for <b>l</b>ines,</li> <li>• "b" for <b>b</b>oth,</li> <li>• "c" for the lines part alone of "b",</li> <li>• "o" for both "overplotted",</li> <li>• "h" for "histogram" like (or "high-density") vertical lines,</li> <li>• "s" for stair steps,</li> <li>• "S" for other steps, see <i>Details</i> below,</li> <li>• "n" for no plotting.</li> </ul> <p>All other types give a warning or an error; using, e.g., <code>type = "punkte"</code> being equivalent to <code>type = "p"</code> for S compatibility.</p>
main	an overall title for the plot: see <a href="#">title</a> .
sub	a sub title for the plot: see <a href="#">title</a> .
xlab	a title for the x axis: see <a href="#">title</a> .
ylab	a title for the y axis: see <a href="#">title</a> .

**Details**

For simple scatter plots, `plot.default` will be used. However, there are `plot` methods for many R objects, including [functions](#), [data.frames](#), [density](#) objects, etc. Use `methods(plot)` and the documentation for these.

The two step types differ in their x-y preference: Going from  $(x_1, y_1)$  to  $(x_2, y_2)$  with  $x_1 < x_2$ , `type = "s"` moves first horizontal, then vertical, whereas `type = "S"` moves the other way around.

**See Also**

[plot.default](#), [plot.formula](#) and other methods; [points](#), [lines](#), [par](#).

**Examples**

```

plot(cars)
lines(lowess(cars))

plot(sin, -pi, 2*pi)

## Discrete Distribution Plot:
plot(table(rpois(100,5)), type = "h", col = "red", lwd=10,
      main="rpois(100,lambda=5)")

## Simple quantiles/ECDF, see ecdf() {library(stats)} for a better one:
plot(x <- sort(rnorm(47)), type = "s", main = "plot(x, type = \"s\")")
points(x, cex = .5, col = "dark red")

```

---

plot.data.frame      *Plot Method for Data Frames*

---

**Description**

plot.data.frame, a method of the `plot` generic, uses `stripchart` for *one* variable, `plot.default` (scatterplot) for *two* variables, and `pairs` (scatterplot matrix) otherwise.

**Usage**

```

## S3 method for class 'data.frame':
plot(x, ...)

```

**Arguments**

`x`                    object of class `data.frame`.

`...`                further arguments to `stripchart`, `plot.default` or `pairs`.

**See Also**

[data.frame](#)

**Examples**

```

plot(OrchardSprays[1], method="jitter")
plot(OrchardSprays[c(4,1)])
plot(OrchardSprays)

```

---

plot.default                    *The Default Scatterplot Function*


---

**Description**

Draw a scatter plot with “decorations” such as axes and titles in the active graphics window.

**Usage**

```
## Default S3 method:
plot(x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
     ann = par("ann"), axes = TRUE, frame.plot = axes,
     panel.first = NULL, panel.last = NULL,
     col = par("col"), bg = NA, pch = par("pch"),
     cex = 1, lty = par("lty"), lab = par("lab"),
     lwd = par("lwd"), asp = NA, ...)
```

**Arguments**

<code>x,y</code>	the <code>x</code> and <code>y</code> arguments provide the <code>x</code> and <code>y</code> coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function <a href="#">xy.coords</a> for details.
<code>type</code>	1-character string giving the type of plot desired. The following values are possible, for details, see <a href="#">plot</a> : "p" for points, "l" for lines, "o" for overplotted points and lines, "b", "c" for (empty if "c") points joined by lines, "s" and "S" for stair steps and "h" for histogram-like vertical lines. Finally, "n" does not produce any points or lines.
<code>xlim</code>	the <code>x</code> limits (min,max) of the plot.
<code>ylim</code>	the <code>y</code> limits of the plot.
<code>log</code>	a character string which contains "x" if the <code>x</code> axis is to be logarithmic, "y" if the <code>y</code> axis is to be logarithmic and "xy" or "yx" if both axes are to be logarithmic.
<code>main</code>	a main title for the plot.
<code>sub</code>	a sub title for the plot.
<code>xlab</code>	a label for the <code>x</code> axis.
<code>ylab</code>	a label for the <code>y</code> axis.
<code>ann</code>	a logical value indicating whether the default annotation (title and <code>x</code> and <code>y</code> axis labels) should appear on the plot.
<code>axes</code>	a logical value indicating whether axes should be drawn on the plot.
<code>frame.plot</code>	a logical indicating whether a box should be drawn around the plot.
<code>panel.first</code>	an expression to be evaluated after the plot axes are set up but before any plotting takes place. This can be useful for drawing background grids or scatterplot smooths.

<code>panel.last</code>	an expression to be evaluated after plotting has taken place.
<code>col</code>	The colors for lines and points. Multiple colors can be specified so that each point can be given its own color. If there are fewer colors than points they are recycled in the standard fashion. Lines will all be plotted in the first colour specified.
<code>bg</code>	background color for open plot symbols, see <a href="#">points</a> .
<code>pch</code>	a vector of plotting characters or symbols: see <a href="#">points</a> .
<code>cex</code>	a numerical vector giving the amount by which plotting text and symbols should be scaled relative to the default.
<code>lty</code>	the line type, see <a href="#">par</a> .
<code>lab</code>	the specification for the (approximate) numbers of tick marks on the x and y axes.
<code>lwd</code>	the positive line width.
<code>asp</code>	the $y/x$ aspect ratio, see <a href="#">plot.window</a> .
<code>...</code>	graphical parameters as in <a href="#">par</a> may also be passed as arguments.

## References

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
- Cleveland, W. S. (1985) *The Elements of Graphing Data*. Monterey, CA: Wadsworth.

## See Also

[plot](#), [plot.window](#), [xy.coords](#).

## Examples

```
Speed <- cars$speed
Distance <- cars$dist
plot(Speed, Distance, panel.first = grid(8,8),
     pch = 0, cex = 1.2, col = "blue")
plot(Speed, Distance,
     panel.first = lines(lowess(Speed, Distance), lty = "dashed"),
     pch = 0, cex = 1.2, col = "blue")

## Show the different plot types
x <- 0:12
y <- sin(pi/5 * x)
op <- par(mfrow = c(3,3), mar = .1+ c(2,2,3,1))
for (tp in c("p","l","b", "c","o","h", "s","S","n")) {
  plot(y ~ x, type = tp,
       main = paste("plot(*, type = \"",tp,"\"",sep=""))
  if(tp == "S") {
    lines(x,y, type = "s", col = "red", lty = 2)
    mtext("lines(*, type = \"s\\", ...) ", col = "red", cex=.8)
  }
}
```

```

par(op)

##--- Log-Log Plot with custom axes
lx <- seq(1,5, length=41)
yl <- expression(e^{-frac(1,2) * {log[10](x)}^2})
y <- exp(-.5*lx^2)
op <- par(mfrow=c(2,1), mar=par("mar")+c(0,1,0,0))
plot(10^lx, y, log="xy", type="l", col="purple",
     main="Log-Log plot", ylab=yl, xlab="x")
plot(10^lx, y, log="xy", type="o", pch='.', col="forestgreen",
     main="Log-Log plot with custom axes", ylab=yl, xlab="x",
     axes = FALSE, frame.plot = TRUE)
axis(1, at = my.at <- 10^(1:5), labels = formatC(my.at, format="fg"))
at.y <- 10^(-5:-1)
axis(2, at = at.y, labels = formatC(at.y, format="fg"), col.axis="red")
par(op)

```

---

plot.design

*Plot Univariate Effects of a ‘Design’ or Model*


---

### Description

Plot univariate effects of one or more [factors](#), typically for a designed experiment as analyzed by [aov\(\)](#). Further, in S this is a method of the [plot](#) generic function for design objects.

### Usage

```

plot.design(x, y = NULL, fun = mean, data = NULL, ...,
           ylim = NULL, xlab = "Factors", ylab = NULL,
           main = NULL, ask = NULL, xaxt = par("xaxt"),
           axes = TRUE, xtick = FALSE)

```

### Arguments

<code>x</code>	either a data frame containing the design factors and optionally the response, or a <a href="#">formula</a> or <a href="#">terms</a> object.
<code>y</code>	the response, if not given in <code>x</code> .
<code>fun</code>	a function (or name of one) to be applied to each subset. It must return one number for a numeric (vector) input.
<code>data</code>	data frame containing the variables referenced by <code>x</code> when that is formula like.
<code>...</code>	graphical arguments such as <code>col</code> , see <a href="#">par</a> .
<code>ylim</code>	range of <code>y</code> values, as in <a href="#">plot.default</a> .
<code>xlab</code>	<code>x</code> axis label, see <a href="#">title</a> .
<code>ylab</code>	<code>y</code> axis label with a “smart” default.
<code>main</code>	main title, see <a href="#">title</a> .

ask	logical indicating if the user should be asked before a new page is started – in the case of multiple y's.
xaxt	character giving the type of x axis.
axes	logical indicating if axes should be drawn.
xtick	logical indicating if “ticks” (one per factor) should be drawn on the x axis.

### Details

The supplied function will be called once for each level of each factor in the design and the plot will show these summary values. The levels of a particular factor are shown along a vertical line, and the overall value of `fun()` for the response is drawn as a horizontal line.

This is a new R implementation which will not be completely compatible to the earlier S implementations. This is not a bug but might still change.

### Note

A big effort was taken to make this closely compatible to the S version. However, `col` (and `fg`) specification has different effects.

### Author(s)

Roberto Frisullo and Martin Maechler

### References

Chambers, J. M. and Hastie, T. J. eds (1992) *Statistical Models in S*. Chapman & Hall, London, **the white book**, pp. 546–7 (and 163–4).

Freeny, A. E. and Landwehr, J. M. (1990) Displays for data from large designed experiments; *Computer Science and Statistics: Proc. 22nd SympInterface*, 117–126, Springer Verlag.

### See Also

[interaction.plot](#) for a “standard graphic” of designed experiments.

### Examples

```
plot.design(warpbreaks)# automatic for data frame with one numeric var.

Form <- breaks ~ wool + tension
summary(fml <- aov(Form, data = warpbreaks))
plot.design(      Form, data = warpbreaks, col = 2)# same as above

## More than one y :
utils::str(esoph)
plot.design(esoph) ## two plots; if interactive you are "ask"ed

## or rather, compare mean and median:
op <- par(mfcol = 1:2)
plot.design(ncases/ncontrols ~ ., data = esoph, ylim = c(0, 0.8))
```

```
plot.design(ncases/ncontrols ~ ., data = esoph, ylim = c(0, 0.8),
            fun = median)
par(op)
```

---

plot.factor                      *Plotting Factor Variables*

---

### Description

This functions implements a “scatterplot” method for [factor](#) arguments of the *generic* [plot](#) function. Actually, [boxplot](#) or [barplot](#) are used when appropriate.

### Usage

```
## S3 method for class 'factor':
plot(x, y, legend.text = levels(y), ...)
```

### Arguments

<code>x, y</code>	numeric or factor. <code>y</code> may be missing.
<code>legend.text</code>	a vector of text used to construct a legend for the plot. Only used if <code>y</code> is present and a factor.
<code>...</code>	Further arguments to <a href="#">plot</a> , see also <a href="#">par</a> .

### See Also

[plot.default](#), [plot.formula](#), [barplot](#), [boxplot](#).

### Examples

```
plot(PlantGrowth) # -> plot.data.frame
plot(weight ~ group, data = PlantGrowth) # numeric vector ~ factor
plot(cut(weight, 2) ~ group, data = PlantGrowth) # factor ~ factor
## passing "..." to barplot() eventually:
plot(cut(weight, 3) ~ group, data = PlantGrowth, density = 16*(1:3), col=NULL)

plot(PlantGrowth$group, axes=FALSE, main="no axes")# extremely silly
```

---

plot.formula                      *Formula Notation for Scatterplots*

---

### Description

Specify a scatterplot or add points or lines via a formula.

### Usage

```
## S3 method for class 'formula':
plot(formula, data = parent.frame(), ..., subset,
      ylab = varnames[response], ask = TRUE)

## S3 method for class 'formula':
points(formula, data = parent.frame(), ..., subset)

## S3 method for class 'formula':
lines(formula, data = parent.frame(), ..., subset)
```

### Arguments

formula	a <a href="#">formula</a> , such as $y \sim x$ .
data	a data.frame (or list) from which the variables in formula should be taken.
...	Further graphical parameters may also be passed as arguments, see <a href="#">par</a> . <code>horizontal = TRUE</code> is also accepted.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
ylab	the y label of the plot(s).
ask	logical, see <a href="#">par</a> .

### Details

Both the terms in the formula and the ... arguments are evaluated in data enclosed in `parent.frame()` if data is a list or a data frame. The terms of the formula and those arguments in ... that are of the same length as data are subjected to the subsetting specified in subset. If the formula in `plot.formula` contains more than one non-response term, a series of plots of y against each term is given. A plot against the running index can be specified as `plot(y ~ 1)`.

Missing values are not considered in these methods, and in particular cases with missing values are not removed.

If y is an object (ie. has a `class` attribute) then `plot.formula` looks for a plot method for that class first. Otherwise, the class of x will determine the type of the plot. For factors this will be a parallel boxplot, and argument `horizontal = TRUE` can be used (see [boxplot](#)).

**Value**

These functions are invoked for their side effect of drawing in the active graphics device.

**See Also**

[plot.default](#), [plot.factor](#).

**Examples**

```
op <- par(mfrow=c(2,1))
plot(Ozone ~ Wind, data = airquality, pch=as.character(Month))
plot(Ozone ~ Wind, data = airquality, pch=as.character(Month),
      subset = Month != 7)
par(op)
```

---

plot.histogram      *Plot Histograms*

---

**Description**

These are methods for objects of class "histogram", typically produced by [hist](#).

**Usage**

```
## S3 method for class 'histogram':
plot(x, freq = equidist, density = NULL, angle = 45,
      col = NULL, border = par("fg"), lty = NULL,
      main = paste("Histogram of", x$name),
      sub = NULL, xlab = x$name, ylab,
      xlim = range(x$breaks), ylim = NULL,
      axes = TRUE, labels = FALSE, add = FALSE, ...)

## S3 method for class 'histogram':
lines(x, ...)
```

**Arguments**

x	a histogram object, or a list with components density, mid, etc, see <a href="#">hist</a> for information about the components of x.
freq	logical; if TRUE, the histogram graphic is to present a representation of frequencies, i.e. x\$counts; if FALSE, <i>relative</i> frequencies ("probabilities"), i.e., x\$density, are plotted. The default is true for equidistant breaks and false otherwise.
col	a colour to be used to fill the bars. The default of NULL yields unfilled bars.
border	the color of the border around the bars.
angle, density	select shading of bars by lines: see <a href="#">rect</a> .

`lty` the line type used for the bars, see also [lines](#).  
`main, sub, xlab, ylab` these arguments to `title` have useful defaults here.  
`xlim, ylim` the range of x and y values with sensible defaults.  
`axes` logical, indicating if axes should be drawn.  
`labels` logical or character. Additionally draw labels on top of bars, if not `FALSE`; if `TRUE`, draw the counts or rounded densities; if `labels` is a character, draw itself.  
`add` logical. If `TRUE`, only the bars are added to the current plot. This is what `lines.histogram(*)` does.  
`...` further graphical parameters to `title` and `axis`.

### Details

`lines.histogram(*)` is the same as `plot.histogram(*, add = TRUE)`.

### See Also

[hist](#), [stem](#), [density](#).

### Examples

```

(wwt <- hist(women$weight, nc= 7, plot = FALSE))
plot(wwt, labels = TRUE) # default main & xlab using wwt$xname
plot(wwt, border = "dark blue", col = "light blue",
      main = "Histogram of 15 women's weights", xlab = "weight [pounds]")

## Fake "lines" example, using non-default labels:
w2 <- wwt; w2$counts <- w2$counts - 1
lines(w2, col = "Midnight Blue", labels = ifelse(w2$counts, "> 1", "1"))
  
```

---

plot.table

*Plot Methods for 'table' Objects*

---

### Description

This is a method of the generic `plot` function for (contingency) [table](#) objects. Whereas for two- and more dimensional tables, a [mosaicplot](#) is drawn, one-dimensional ones are plotted “bar like”.

### Usage

```

## S3 method for class 'table':
plot(x, type = "h", ylim = c(0, max(x)), lwd = 2,
      xlab = NULL, ylab = NULL, frame.plot = is.num, ...)
  
```

**Arguments**

x	a <a href="#">table</a> (like) object.
type	plotting type.
ylim	range of y-axis.
lwd	line width for bars when type = "h" is used in the 1D case.
xlab, ylab	x- and y-axis labels.
frame.plot	logical indicating if a frame ( <a href="#">box</a> ) should be drawn in the 1D case. Defaults to true when x has <a href="#">dimnames</a> coerceable to numbers.
...	further graphical arguments, see <a href="#">plot.default</a> .

**Details**

The current implementation (R 1.2) is somewhat experimental and will be improved and extended.

**See Also**

[plot.factor](#), the [plot](#) method for factors.

**Examples**

```
## 1-d tables
(Poiss.tab <- table(N = rpois(200, lam= 5)))
plot(Poiss.tab, main = "plot(table(rpois(200, lam=5)))")

plot(table(state.division))

## 4-D :
plot(Titanic, main = "plot(Titanic, main= *)")
```

---

plot.window	<i>Set up World Coordinates for Graphics Window</i>
-------------	---

---

**Description**

This function sets up the world coordinate system for a graphics window. It is called by higher level functions such as [plot.default](#) (after [plot.new](#)).

**Usage**

```
plot.window(xlim, ylim, log = "", asp = NA, ...)
```

**Arguments**

`xlim`, `ylim` numeric of length 2, giving the x and y coordinates ranges.  
`log` character; indicating which axes should be in log scale.  
`asp` numeric, giving the **aspect** ratio y/x.  
`...` further graphical parameters as in [par](#).

**Details**

Note that if `asp` is a finite positive value then the window is set up so that one data unit in the x direction is equal in length to `asp` × one data unit in the y direction.

The special case `asp == 1` produces plots where distances between points are represented accurately on screen. Values with `asp > 1` can be used to produce more accurate maps when using latitude and longitude.

The function attempts to produce a plausible set of scales if one or both of `xlim` and `ylim` is of length one or the two values given are identical, but it is better to avoid that case.

Usually, one should rather use the higher level functions such as [plot](#), [hist](#), [image](#), ..., instead and refer to their help pages for explanation of the arguments.

**See Also**

[xy.coords](#), [plot.xy](#), [plot.default](#).

**Examples**

```
##--- An example for the use of 'asp' :
require(stats) # normally loaded
loc <- cmdscale(eurodist)
rx <- range(x <- loc[,1])
ry <- range(y <- -loc[,2])
plot(x, y, type="n", asp=1, xlab="", ylab="")
abline(h = pretty(rx, 10), v = pretty(ry, 10), col = "lightgray")
text(x, y, names(eurodist), cex=0.8)
```

---

plot.xy

*Basic Internal Plot Function*

---

**Description**

This is *the* internal function that does the basic plotting of points and lines. Usually, one should rather use the higher level functions instead and refer to their help pages for explanation of the arguments.

**Usage**

```
plot.xy(xy, type, pch = 1, lty = "solid", col = par("fg"), bg = NA,
        cex = 1, lwd = par("lwd"), ...)
```

**Arguments**

<code>xy</code>	A four-element list as results from <code>xy.coords</code> .
<code>type</code>	1 character code.
<code>pch</code>	character or integer code for kind of points/lines, see <code>points.default</code> .
<code>lty</code>	line type code, see <code>lines</code> .
<code>col</code>	color code or name, see <code>colors</code> , <code>palette</code> .
<code>bg</code>	background (“fill”) color for open plot symbols.
<code>cex</code>	character expansion.
<code>lwd</code>	line width, also used for (non-filled) plot symbols, see <code>lines</code> and <code>points</code> .
<code>...</code>	further graphical parameters.

**Details**

The arguments `pch`, `col`, `bg`, `cex`, `lwd` are may be vectors and recycled, depending on `type`, see `points` and `lines` for specifics.

**See Also**

`plot`, `plot.default`, `points`, `lines`.

**Examples**

```
points.default # to see how it calls "plot.xy(xy.coords(x, y), ...)"
```

---

`points`

*Add Points to a Plot*

---

**Description**

`points` is a generic function to draw a sequence of points at the specified coordinates. The specified character(s) are plotted, centered at the coordinates.

**Usage**

```
points(x, ...)

## Default S3 method:
points(x, y = NULL, type = "p", pch = par("pch"),
       col = par("col"), bg = NA, cex = 1, ...)
```

**Arguments**

<code>x, y</code>	coordinate vectors of points to plot.
<code>type</code>	character indicating the type of plotting; actually any of the types as in <a href="#">plot</a> .
<code>pch</code>	plotting “character”, i.e., symbol to use. <code>pch</code> can either be a <a href="#">character</a> or an integer code for a set of graphics symbols. The full set of S symbols is available with <code>pch=0:18</code> , see the last picture from <code>example(points)</code> , i.e., the examples below.  In addition, there is a special set of R plotting symbols which can be obtained with <code>pch=19:25</code> and <code>21:25</code> can be colored and filled with different colors: <ul style="list-style-type: none"> <li>• <code>pch=19</code>: solid circle,</li> <li>• <code>pch=20</code>: bullet (smaller circle),</li> <li>• <code>pch=21</code>: circle,</li> <li>• <code>pch=22</code>: square,</li> <li>• <code>pch=23</code>: diamond,</li> <li>• <code>pch=24</code>: triangle point-up,</li> <li>• <code>pch=25</code>: triangle point down.</li> </ul> Values <code>pch=26:32</code> are currently unused, and <code>pch=32:255</code> give the text symbol in the encoding in use (see <a href="#">postscript</a> ).
<code>col</code>	color code or name, see <a href="#">par</a> .
<code>bg</code>	background (“fill”) color for open plot symbols
<code>cex</code>	character expansion: a numerical vector.
<code>...</code>	Further graphical parameters (see <a href="#">plot.xy</a> and <a href="#">par</a> ) may also be supplied as arguments.

**Details**

The coordinates can be passed in a plotting structure (a list with `x` and `y` components), a two-column matrix, a time series, .... See [xy.coords](#).

Arguments `pch`, `col`, `bg`, `cex` and `lwd` can be vectors (which will be recycled as needed) giving a value for each point plotted. Points whose `x`, `y`, `pch`, `col` or `cex` value is `NA` are omitted from the plot.

Graphical parameters are permitted as arguments to this function.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[plot](#), [lines](#), and the underlying “primitive” [plot.xy](#).

**Examples**

```

plot(-4:4, -4:4, type = "n")# setting up coord. system
points(rnorm(200), rnorm(200), col = "red")
points(rnorm(100)/2, rnorm(100)/2, col = "blue", cex = 1.5)

op <- par(bg = "light blue")
x <- seq(0,2*pi, len=51)
## something "between type='b' and type='o'":
plot(x, sin(x), type="o", pch=21, bg=par("bg"), col = "blue", cex=.6,
     main='plot(..., type="o", pch=21, bg=par("bg"))')
par(op)

##----- Showing all the extra & some char graphics symbols -----
Pex <- 3 ## good for both .Device=="postscript" and "x11"
ipch <- 1:(np <- 25+11); k <- floor(sqrt(np)); dd <- c(-1,1)/2
rx <- dd + range(ix <- (ipch-1) %/% k)
ry <- dd + range(iy <- 3 + (k-1)-(ipch-1) %% k)
pch <- as.list(ipch)
pch[25+ 1:11] <- as.list(c("*", ".", "o", "O", "0", "+", "-", ":", "|", "%", "#"))
plot(rx, ry, type="n", axes = FALSE, xlab = "", ylab = "",
     main = paste("plot symbols : points (... pch = *, cex =", Pex, ")"))
abline(v = ix, h = iy, col = "lightgray", lty = "dotted")
for(i in 1:np) {
  pc <- pch[[i]]
  points(ix[i], iy[i], pch = pc, col = "red", bg = "yellow", cex = Pex)
  ## red symbols with a yellow interior (where available)
  text(ix[i] - .3, iy[i], pc, col = "brown", cex = 1.2)
}

```

---

polygon

*Polygon Drawing*

---

**Description**

polygon draws the polygons whose vertices are given in x and y.

**Usage**

```

polygon(x, y = NULL, density = NULL, angle = 45,
       border = NULL, col = NA, lty = NULL, xpd = NULL, ...)

```

**Arguments**

**x, y** vectors containing the coordinates of the vertices of the polygon.

**density** the density of shading lines, in lines per inch. The default value of NULL means that no shading lines are drawn. A zero value of density means no shading lines whereas negative values (and NA) suppress shading (and so allow color filling).

angle	the slope of shading lines, given as an angle in degrees (counter-clockwise).
col	the color for filling the polygon. The default, NA, is to leave polygons unfilled.
border	the color to draw the border. The default, NULL, uses <code>par("fg")</code> . Use <code>border = NA</code> to omit borders. For compatibility with S, <code>border</code> can also be logical, in which case FALSE is equivalent to NA (borders omitted) and TRUE is equivalent to NULL (use the foreground colour),
lty	the line type to be used, as in <code>par</code> .
xpd	(where) should clipping take place? Defaults to <code>par("xpd")</code> .
...	graphical parameters can be given as arguments to <code>polygon</code> .

### Details

The coordinates can be passed in a plotting structure (a list with x and y components), a two-column matrix, .... See `xy.coords`.

It is assumed that the polygon is closed by joining the last point to the first point.

The coordinates can contain missing values. The behaviour is similar to that of `lines`, except that instead of breaking a line into several lines, NA values break the polygon into several complete polygons (including closing the last point to the first point). See the examples below.

When multiple polygons are produced, the values of `density`, `angle`, `col`, `border`, and `lty` are recycled in the usual manner.

### Bugs

The present shading algorithm can produce incorrect results for self-intersecting polygons.

### Author(s)

The code implementing polygon shading was donated by Kevin Buhr (`buhr@stat.wisc.edu`).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`segments` for even more flexibility, `lines`, `rect`, `box`, `abline`.

`par` for how to specify colors.

### Examples

```
x <- c(1:9,8:1)
y <- c(1,2*(5:3),2,-1,17,9,8,2:9)
op <- par(mfcol=c(3,1))
for(xpd in c(FALSE,TRUE,NA)) {
  plot(1:10, main=paste("xpd =", xpd)) ; box("figure", col = "pink", lwd=3)
```

```

    polygon(x,y, xpd=xpd, col = "orange", lty=2, lwd=2, border = "red")
  }
  par(op)

  n <- 100
  xx <- c(0:n, n:0)
  yy <- c(c(0,cumsum(rnorm(n))), rev(c(0,cumsum(rnorm(n)))))
  plot (xx, yy, type="n", xlab="Time", ylab="Distance")
  polygon(xx, yy, col="gray", border = "red")
  title("Distance Between Brownian Motions")

  # Multiple polygons from NA values
  # and recycling of col, border, and lty
  op <- par(mfrow=c(2,1))
  plot(c(1,9), 1:2, type="n")
  polygon(1:9, c(2,1,2,1,1,2,1,2,1),
         col=c("red", "blue"),
         border=c("green", "yellow"),
         lwd=3, lty=c("dashed", "solid"))
  plot(c(1,9), 1:2, type="n")
  polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
         col=c("red", "blue"),
         border=c("green", "yellow"),
         lwd=3, lty=c("dashed", "solid"))
  par(op)

  # Line-shaded polygons
  plot(c(1,9), 1:2, type="n")
  polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
         density=c(10, 20), angle=c(-45, 45))

```

---

rect

*Draw One or More Rectangles*


---

### Description

rect draws a rectangle (or sequence of rectangles) with the given coordinates, fill and border colors.

### Usage

```

rect(xleft, ybottom, xright, ytop, density = NULL, angle = 45,
     col = NULL, border = NULL, lty = NULL, lwd = par("lwd"),
     xpd = NULL, ...)

```

### Arguments

xleft            a vector (or scalar) of left x positions.  
ybottom         a vector (or scalar) of bottom y positions.

<code>xright</code>	a vector (or scalar) of right x positions.
<code>ytopy</code>	a vector (or scalar) of top y positions.
<code>density</code>	the density of shading lines, in lines per inch. The default value of <code>NULL</code> means that no shading lines are drawn. A zero value of <code>density</code> means no shading lines whereas negative values (and <code>NA</code> ) suppress shading (and so allow color filling).
<code>angle</code>	angle (in degrees) of the shading lines.
<code>col</code>	color(s) to fill or shade the rectangle(s) with. The default <code>NULL</code> , or also <code>NA</code> do not fill, i.e., draw transparent rectangles, unless <code>density</code> is specified.
<code>border</code>	color for rectangle border(s). Can also be <code>FALSE</code> to suppress the border, or <code>TRUE</code> in which case <code>col</code> is used.
<code>lty</code>	line type for borders and shading; defaults to <code>"solid"</code> .
<code>lwd</code>	line width for borders and shading.
<code>xpd</code>	logical ( <b>"expand"</b> ); defaults to <code>par("xpd")</code> . See <code>par(xpd=)</code> .
<code>...</code>	other graphical parameters can be given as arguments.

### Details

The positions supplied, i.e., `xleft`, `...`, are relative to the current plotting region. If the x-axis goes from 100 to 200 then `xleft` must be larger than 100 and `xright` must be less than 200.

It is a primitive function used in `hist`, `barplot`, `legend`, etc.

### See Also

`box` for the "standard" box around the plot; `polygon` and `segments` for flexible line drawing.

`par` for how to specify colors.

### Examples

```
## set up the plot region:
op <- par(bg = "thistle")
plot(c(100, 250), c(300, 450), type = "n", xlab="", ylab="",
      main = "2 x 11 rectangles; 'rect(100+i,300+i, 150+i,380+i)'")
i <- 4*(0:10)
## draw rectangles with bottom left (100, 300)+i and top right (150, 380)+i
rect(100+i, 300+i, 150+i, 380+i, col=rainbow(11, start=.7,end=.1))
rect(240-i, 320+i, 250-i, 410+i, col=heat.colors(11), lwd=i/5)
## Background alternating ( transparent / "bg" ) :
j <- 10*(0:5)
rect(125+j, 360+j, 141+j, 405+j/2, col = c(NA,0), border = "gold", lwd = 2)
rect(125+j, 296+j/2, 141+j, 331+j/5, col = c(NA,"midnightblue"))
mtext("+ 2 x 6 rect(*, col = c(NA,0)) and col = c(NA,\"m..blue\")")

## an example showing colouring and shading
plot(c(100, 200), c(300, 450), type= "n", xlab="", ylab="")
rect(100, 300, 125, 350) # transparent
rect(100, 400, 125, 450, col="green", border="blue") # coloured
```

```

rect(115, 375, 150, 425, col=par("bg"), border="transparent")
rect(150, 300, 175, 350, density=10, border="red")
rect(150, 400, 175, 450, density=30, col="blue",
      angle=-30, border="transparent")

legend(180, 450, legend=1:4, fill=c(NA, "green", par("fg"), "blue"),
       density=c(NA, NA, 10, 30), angle=c(NA, NA, 30, -30))

par(op)

```

---

 rug

*Add a Rug to a Plot*


---

### Description

Adds a *rug* representation (1-d plot) of the data to the plot.

### Usage

```

rug(x, ticksize=0.03, side=1, lwd=0.5, col,
    quiet = getOption("warn") < 0, ...)

```

### Arguments

<code>x</code>	A numeric vector
<code>ticksize</code>	The length of the ticks making up the ‘rug’. Positive lengths give inwards ticks.
<code>side</code>	On which side of the plot box the rug will be plotted. Normally 1 (bottom) or 3 (top).
<code>lwd</code>	The line width of the ticks.
<code>col</code>	The colour the ticks are plotted in, default is black.
<code>quiet</code>	logical indicating if there should be a warning about clipped values.
<code>...</code>	further arguments, passed to <code>axis(...)</code> , such as <code>line</code> or <code>pos</code> for specifying the location of the rug.

### Details

Because of the way `rug` is implemented, only values of `x` that fall within the plot region are included. There will be a warning if any finite values are omitted, but non-finite values are omitted silently.

Because of the way colours are done the axis itself is coloured the same as the ticks. You can always replot the box in black if you don’t like this feature.

### References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

[jitter](#) which you may want for ties in x.

**Examples**

```
with(faithful, {
  plot(stats::density(eruptions, bw=0.15))
  rug(eruptions)
  rug(jitter(eruptions, amount = .01), side = 3, col = "light blue")
})
```

---

screen

*Creating and Controlling Multiple Screens on a Single Device*

---

**Description**

`split.screen` defines a number of regions within the current device which can, to some extent, be treated as separate graphics devices. It is useful for generating multiple plots on a single device. Screens can themselves be split, allowing for quite complex arrangements of plots.

`screen` is used to select which screen to draw in.

`erase.screen` is used to clear a single screen, which it does by filling with the background colour.

`close.screen` removes the specified screen definition(s).

**Usage**

```
split.screen(figs, screen, erase = TRUE)
screen(n = , new = TRUE)
erase.screen(n = )
close.screen(n, all.screens = FALSE)
```

**Arguments**

<code>figs</code>	A two-element vector describing the number of rows and the number of columns in a screen matrix <i>or</i> a matrix with 4 columns. If a matrix, then each row describes a screen with values for the left, right, bottom, and top of the screen (in that order) in NDC units, that is 0 at the lower left coner of the device surface, and 1 at the upper right corner.
<code>screen</code>	A number giving the screen to be split. It defaults to the current screen if there is one, otherwise the whole device region.
<code>erase</code>	logical: should be selected screen be cleared?
<code>n</code>	A number indicating which screen to prepare for drawing ( <code>screen</code> ), erase ( <code>erase.screen</code> ), or close ( <code>close.screen</code> ). ( <code>close.screen</code> will accept a vector of screen numbers.)
<code>new</code>	A logical value indicating whether the screen should be erased as part of the preparation for drawing in the screen.
<code>all.screens</code>	A logical value indicating whether all of the screens should be closed.

## Details

The first call to `split.screen` places R into split-screen mode. The other split-screen functions only work within this mode. While in this mode, certain other commands should be avoided (see the Warnings section below). Split-screen mode is exited by the command `close.screen(all = TRUE)`.

If the current screen is closed, `close.screen` sets the current screen to be the next larger screen number if there is one, otherwise to the first available screen.

## Value

`split.screen` returns a vector of screen numbers for the newly-created screens. With no arguments, `split.screen` returns a vector of valid screen numbers.

`screen invisibly` returns the number of the selected screen. With no arguments, `screen` returns the number of the current screen.

`close.screen` returns a vector of valid screen numbers.

`screen`, `erase.screen`, and `close.screen` all return `FALSE` if R is not in split-screen mode.

## Warnings

The recommended way to use these functions is to completely draw a plot and all additions (ie. points and lines) to the base plot, prior to selecting and plotting on another screen. The behavior associated with returning to a screen to add to an existing plot is unpredictable and may result in problems that are not readily visible.

These functions are totally incompatible with the other mechanisms for arranging plots on a device: `par(mfrow)`, `par(mfcol)` and `layout()`.

The functions are also incompatible with some plotting functions, such as `coplot`, which make use of these other mechanisms.

`erase.screen` will appear not to work if the background colour is transparent (as it is by default on most devices).

## References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

## See Also

`par`, `layout`, `Devices`, `dev.*`

## Examples

```
if (interactive()) {
  par(bg = "white")           # default is likely to be transparent
  split.screen(c(2,1))       # split display into two screens
  split.screen(c(1,3), screen = 2) # now split the bottom half into 3
  screen(1) # prepare screen 1 for output
  plot(10:1)
```

```

screen(4) # prepare screen 4 for output
plot(10:1)
close.screen(all = TRUE) # exit split-screen mode

split.screen(c(2,1)) # split display into two screens
split.screen(c(1,2),2) # split bottom half in two
plot(1:10) # screen 3 is active, draw plot
erase.screen() # forgot label, erase and redraw
plot(1:10, ylab= "ylab 3")
screen(1) # prepare screen 1 for output
plot(1:10)
screen(4) # prepare screen 4 for output
plot(1:10, ylab="ylab 4")
screen(1, FALSE) # return to screen 1, but do not clear
plot(10:1, axes=FALSE, lty=2, ylab="") # overlay second plot
axis(4) # add tic marks to right-hand axis
title("Plot 1")
close.screen(all = TRUE) # exit split-screen mode
}

```

---

segments

---

*Add Line Segments to a Plot*


---

## Description

Draw line segments between pairs of points.

## Usage

```

segments(x0, y0, x1, y1,
         col = par("fg"), lty = par("lty"), lwd = par("lwd"), ...)

```

## Arguments

`x0, y0` coordinates of points **from** which to draw.  
`x1, y1` coordinates of points **to** which to draw.  
`col, lty, lwd` usual graphical parameters as in [par](#).  
`...` further graphical parameters (from [par](#)).

## Details

For each `i`, a line segment is drawn between the point `(x0[i], y0[i])` and the point `(x1[i], y1[i])`.

The graphical parameters `col` and `lty` can be used to specify a color and line texture for the line segments (`col` may be a vector).

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[arrows](#), [polygon](#) for slightly easier and less flexible line drawing, and [lines](#) for the usual polygons.

## Examples

```
x <- runif(12); y <- rnorm(12)
i <- order(x,y); x <- x[i]; y <- y[i]
plot(x,y, main="arrows(.) and segments(.)")
## draw arrows from point to point :
s <- seq(length(x)-1)# one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3)
s <- s[-length(s)]
segments(x[s], y[s], x[s+2], y[s+2], col= 'pink')
```

---

stars

*Star (Spider/Radar) Plots and Segment Diagrams*


---

## Description

Draw star plots or segment diagrams of a multivariate data set. With one single location, also draws “spider” (or “radar”) plots.

## Usage

```
stars(x, full = TRUE, scale = TRUE, radius = TRUE,
      labels = dimnames(x)[[1]], locations = NULL,
      nrow = NULL, ncol = NULL, len = 1,
      key.loc = NULL, key.labels = dimnames(x)[[2]], key.xpd = TRUE,
      xlim = NULL, ylim = NULL, flip.labels = NULL,
      draw.segments = FALSE, col.segments = 1:n.seg, col.stars = NA,
      axes = FALSE, frame.plot = axes,
      main = NULL, sub = NULL, xlab = "", ylab = "",
      cex = 0.8, lwd = 0.25, lty = par("lty"), xpd = FALSE,
      mar = pmin(par("mar"),
                 1.1+ c(2*axes+ (xlab != ""),
                       2*axes+ (ylab != ""), 1,0)),
      add = FALSE, plot = TRUE, ...)
```

**Arguments**

<code>x</code>	matrix or data frame of data. One star or segment plot will be produced for each row of <code>x</code> . Missing values (NA) are allowed, but they are treated as if they were 0 (after scaling, if relevant).
<code>full</code>	logical flag: if TRUE, the segment plots will occupy a full circle. Otherwise, they occupy the (upper) semicircle only.
<code>scale</code>	logical flag: if TRUE, the columns of the data matrix are scaled independently so that the maximum value in each column is 1 and the minimum is 0. If FALSE, the presumption is that the data have been scaled by some other algorithm to the range [0, 1].
<code>radius</code>	logical flag: in TRUE, the radii corresponding to each variable in the data will be drawn.
<code>labels</code>	vector of character strings for labeling the plots. Unlike the S function <code>stars</code> , no attempt is made to construct labels if <code>labels = NULL</code> .
<code>locations</code>	Either two column matrix with the x and y coordinates used to place each of the segment plots; or numeric of length 2 when all plots should be superimposed (for a “spider plot”). By default, <code>locations = NULL</code> , the segment plots will be placed in a rectangular grid.
<code>nrow, ncol</code>	integers giving the number of rows and columns to use when <code>locations</code> is NULL. By default, <code>nrow == ncol</code> , a square layout will be used.
<code>len</code>	scale factor for the length of radii or segments.
<code>key.loc</code>	vector with x and y coordinates of the unit key.
<code>key.labels</code>	vector of character strings for labeling the segments of the unit key. If omitted, the second component of <code>dimnames(x)</code> is used, if available.
<code>key.xpd</code>	clipping switch for the unit key (drawing and labeling), see <code>par("xpd")</code> .
<code>xlim</code>	vector with the range of x coordinates to plot.
<code>ylim</code>	vector with the range of y coordinates to plot.
<code>flip.labels</code>	logical indicating if the label locations should flip up and down from diagram to diagram. Defaults to a somewhat smart heuristic.
<code>draw.segments</code>	logical. If TRUE draw a segment diagram.
<code>col.segments</code>	color vector (integer or character, see <code>par</code> ), each specifying a color for one of the segments (variables). Ignored if <code>draw.segments = FALSE</code> .
<code>col.stars</code>	color vector (integer or character, see <code>par</code> ), each specifying a color for one of the stars (cases). Ignored if <code>draw.segments = TRUE</code> .
<code>axes</code>	logical flag: if TRUE axes are added to the plot.
<code>frame.plot</code>	logical flag: if TRUE, the plot region is framed.
<code>main</code>	a main title for the plot.
<code>sub</code>	a sub title for the plot.
<code>xlab</code>	a label for the x axis.
<code>ylab</code>	a label for the y axis.

<code>cex</code>	character expansion factor for the labels.
<code>lwd</code>	line width used for drawing.
<code>lty</code>	line type used for drawing.
<code>xpd</code>	logical or NA indicating if clipping should be done, see <code>par(xpd = .)</code> .
<code>mar</code>	argument to <code>par(mar = *)</code> , typically choosing smaller margins than by default.
<code>...</code>	further arguments, passed to the first call of <code>plot()</code> , see <code>plot.default</code> and to <code>box()</code> if <code>frame.plot</code> is true.
<code>add</code>	logical, if TRUE <i>add</i> stars to current plot.
<code>plot</code>	logical, if FALSE, nothing is plotted.

### Details

Missing values are treated as 0.

Each star plot or segment diagram represents one row of the input `x`. Variables (columns) start on the right and wind counterclockwise around the circle. The size of the (scaled) column is shown by the distance from the center to the point on the star or the radius of the segment representing the variable.

Only one page of output is produced.

### Note

This code started life as spatial star plots by David A. Andrews. See <http://www.udallas.edu:8080/~andrews/software/software.html>.

Prior to 1.4.1, scaling only shifted the maximum to 1, although documented as here.

### Author(s)

Thomas S. Dye

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### Examples

```
stars(mtcars[, 1:7], key.loc = c(14, 2),
      main = "Motor Trend Cars : stars(*, full = F)", full = FALSE)
stars(mtcars[, 1:7], key.loc = c(14, 1.5),
      main = "Motor Trend Cars : full stars()", flip.labels=FALSE)

## 'Spider' or 'Radar' plot:
stars(mtcars[, 1:7], locations = c(0,0), radius = FALSE,
      key.loc=c(0,0), main="Motor Trend Cars", lty = 2)

## Segment Diagrams:
```

```

palette(rainbow(12, s = 0.6, v = 0.75))
stars(mtcars[, 1:7], len = 0.8, key.loc = c(12, 1.5),
      main = "Motor Trend Cars", draw.segments = TRUE)
stars(mtcars[, 1:7], len = 0.6, key.loc = c(1.5, 0),
      main = "Motor Trend Cars", draw.segments = TRUE,
      frame.plot=TRUE, nrow = 4, cex = .7)

## scale linearly (not affinely) to [0, 1]
USJudge <- apply(USJudgeRatings, 2, function(x) x/max(x))
Jnam <- row.names(USJudgeRatings)
Snam <- abbreviate(substring(Jnam,1,regexpr("[,\\.]",Jnam) - 1), 7)
stars(USJudge, labels = Jnam, scale = FALSE,
      key.loc = c(13, 1.5), main = "Judge not ...", len = 0.8)
stars(USJudge, labels = Snam, scale = FALSE,
      key.loc = c(13, 1.5), radius = FALSE)

loc <- stars(USJudge, labels = NULL, scale = FALSE,
            radius = FALSE, frame.plot = TRUE,
            key.loc = c(13, 1.5), main = "Judge not ...", len = 1.2)
text(loc, Snam, col = "blue", cex = 0.8, xpd = TRUE)

## 'Segments':
stars(USJudge, draw.segments = TRUE, scale = FALSE, key.loc = c(13,1.5))

## 'Spider':
stars(USJudgeRatings, locations=c(0,0), scale=FALSE,radius = FALSE,
      col.stars=1:10, key.loc = c(0,0), main="US Judges rated")
## 'Radar-Segments'
stars(USJudgeRatings[1:10,], locations = 0:1, scale=FALSE,
      draw.segments = TRUE, col.segments=0, col.stars=1:10,key.loc= 0:1,
      main="US Judges 1-10 ")
palette("default")
stars(cbind(1:16,10*(16:1)),draw.segments=TRUE,
      main = "A Joke -- do *not* use symbols on 2D data!")

```

---

stem

*Stem-and-Leaf Plots*


---

## Description

stem produces a stem-and-leaf plot of the values in x. The parameter scale can be used to expand the scale of the plot. A value of scale=2 will cause the plot to be roughly twice as long as the default.

## Usage

```
stem(x, scale = 1, width = 80, atom = 1e-08)
```

**Arguments**

<code>x</code>	a numeric vector.
<code>scale</code>	This controls the plot length.
<code>width</code>	The desired width of plot.
<code>atom</code>	a tolerance.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```
stem(islands)
stem(log10(islands))
```

---

stripchart

*1-D Scatter Plots*


---

**Description**

`stripchart` produces one dimensional scatter plots (or dot plots) of the given data. These plots are a good alternative to [boxplots](#) when sample sizes are small.

**Usage**

```
stripchart(x, method = "overplot", jitter = 0.1, offset = 1/3,
           vertical = FALSE, group.names, add = FALSE,
           at = NULL, xlim = NULL, ylim = NULL,
           main = "", ylab = "", xlab = "",
           log = "", pch = 0, col = par("fg"), cex = par("cex"))
```

**Arguments**

<code>x</code>	the data from which the plots are to be produced. The data can be specified as a single vector, or as list of vectors, each corresponding to a component plot. Alternatively a symbolic specification of the form <code>x ~ g</code> can be given, indicating the the observations in the vector <code>x</code> are to be grouped according to the levels of the factor <code>g</code> . NAs are allowed in the data.
<code>method</code>	the method to be used to separate coincident points. The default method "overplot" causes such points to be overplotted, but it is also possible to specify "jitter" to jitter the points, or "stack" have coincident points stacked. The last method only makes sense for very granular data.
<code>jitter</code>	when jittering is used, <code>jitter</code> gives the amount of jittering applied.
<code>offset</code>	when stacking is used, points are stacked this many line-heights (symbol widths) apart.

vertical when vertical is TRUE the plots are drawn vertically rather than the default horizontal.

group.names group labels which will be printed alongside (or underneath) each plot.

add logical, if true *add* boxplot to current plot.

at numeric vector giving the locations where the boxplots should be drawn, particularly when add = TRUE; defaults to 1:n where n is the number of boxes.

xlim, ylim, main, ylab, xlab, log, pch, col, cex  
Graphical parameters.

### Details

Extensive examples of the use of this kind of plot can be found in Box, Hunter and Hunter or Seber and Wild.

### Examples

```
x <- rnorm(50)
xr <- round(x, 1)
stripchart(x) ; m <- mean(par("usr")[1:2])
text(m, 1.04, "stripchart(x, \"overplot\")")
stripchart(xr, method = "stack", add = TRUE, at = 1.2)
text(m, 1.35, "stripchart(round(x,1), \"stack\")")
stripchart(xr, method = "jitter", add = TRUE, at = 0.7)
text(m, 0.85, "stripchart(round(x,1), \"jitter\")")

with(OrchardSprays,
     stripchart(decrease ~ treatment,
               main = "stripchart(Orchardsprays)", ylab = "decrease",
               vertical = TRUE, log = "y"))

with(OrchardSprays,
     stripchart(decrease ~ treatment, at = c(1:8)^2,
               main = "stripchart(Orchardsprays)", ylab = "decrease",
               vertical = TRUE, log = "y"))
```

---

strwidth

*Plotting Dimensions of Character Strings and Math Expressions*

---

### Description

These functions compute the width or height, respectively, of the given strings or mathematical expressions  $s[i]$  on the current plotting device in *user* coordinates, *inches* or as fraction of the figure width `par("fin")`.

### Usage

```
strwidth(s, units = "user", cex = NULL)
strheight(s, units = "user", cex = NULL)
```

**Arguments**

<code>s</code>	character vector or <a href="#">expressions</a> whose string widths in plotting units are to be determined. An attempt is made to coerce other vectors to character, and other language objects to expressions.
<code>units</code>	character indicating in which units <code>s</code> is measured; should be one of "user", "inches", "figure"; partial matching is performed.
<code>cex</code>	numeric character <b>exp</b> ansion factor; multiplied by <code>par("cex")</code> yields the final character size; the default NULL is equivalent to 1.

**Value**

Numeric vector with the same length as `s`, giving the width or height for each `s[i]`. NA strings are given width and height 0 (as they are not plotted).

**See Also**

[text](#), [nchar](#)

**Examples**

```
str.ex <- c("W","w","I",".", "WwI.")
op <- par(pty='s'); plot(1:100,1:100, type="n")
sw <- strwidth(str.ex); sw
all.equal(sum(sw[1:4]), sw[5])#- since the last string contains the others

sw.i <- strwidth(str.ex, "inches"); 25.4 * sw.i # width in [mm]
unique(sw / sw.i)
# constant factor: 1 value
mean(sw.i / strwidth(str.ex, "fig")) / par('fin')[1] # = 1: are the same

## See how letters fall in classes -- depending on graphics device and font!
all.lett <- c(letters, LETTERS)
shL <- strheight(all.lett, units = "inches") * 72 # 'big points'
table(shL) # all have same heights ...
mean(shL)/par("cin")[2] # around 0.6

(swL <- strwidth(all.lett, units="inches") * 72) # 'big points'
split(all.lett, factor(round(swL, 2)))

sumex <- expression(sum(x[i], i=1,n), e^{i * pi} == -1)
strwidth(sumex)
strheight(sumex)

par(op)#- reset to previous setting
```

---

sunflowerplot      *Produce a Sunflower Scatter Plot*

---

### Description

Multiple points are plotted as “sunflowers” with multiple leaves (“petals”) such that overplotting is visualized instead of accidental and invisible.

### Usage

```
sunflowerplot(x, y = NULL, number, log = "", digits = 6,
              xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
              add = FALSE, rotate = FALSE,
              pch = 16, cex = 0.8, cex.fact = 1.5,
              size = 1/8, seg.col = 2, seg.lwd = 1.5, ...)
```

### Arguments

x	numeric vector of x-coordinates of length n, say, or another valid plotting structure, as for <code>plot.default</code> , see also <code>xy.coords</code> .
y	numeric vector of y-coordinates of length n.
number	integer vector of length n. <code>number[i]</code> = number of replicates for $(x[i], y[i])$ , may be 0. Default: compute the exact multiplicity of the points $x[ ]$ , $y[ ]$ .
log	character indicating log coordinate scale, see <code>plot.default</code> .
digits	when <code>number</code> is computed (i.e., not specified), x and y are rounded to <code>digits</code> significant digits before multiplicities are computed.
xlab, ylab	character label for x-, or y-axis, respectively.
xlim, ylim	<code>numeric(2)</code> limiting the extents of the x-, or y-axis.
add	logical; should the plot be added on a previous one? Default is FALSE.
rotate	logical; if TRUE, randomly rotate the sunflowers (preventing artefacts).
pch	plotting character to be used for points ( <code>number[i]=1</code> ) and center of sunflowers.
cex	numeric; character size expansion of center points (s. <code>pch</code> ).
cex.fact	numeric <i>shrinking</i> factor to be used for the center points <i>when there are flower leaves</i> , i.e., <code>cex / cex.fact</code> is used for these.
size	of sunflower leaves in inches, <code>1[in] := 2.54[cm]</code> . Default: 1/8; approximately 3.2mm.
seg.col	color to be used for the <b>segments</b> which make the sunflowers leaves, see <code>par(col=)</code> ; <code>col = "gold"</code> reminds of real sunflowers.
seg.lwd	numeric; the line width for the leaves' segments.
...	further arguments to <code>plot</code> [if <code>add=FALSE</code> ].

**Details**

For `number[i]=1`, a (slightly enlarged) usual plotting symbol (`pch`) is drawn. For `number[i] > 1`, a small plotting symbol is drawn and `number[i]` equi-angular “rays” emanate from it.

If `rotate=TRUE` and `number[i] >= 2`, a random direction is chosen (instead of the y-axis) for the first ray. The goal is to `jitter` the orientations of the sunflowers in order to prevent artefactual visual impressions.

**Value**

A list with three components of same length,

<code>x</code>	x coordinates
<code>y</code>	y coordinates
<code>number</code>	number

**Side Effects**

A scatter plot is drawn with “sunflowers” as symbols.

**Author(s)**

Andreas Ruckstuhl, Werner Stahel, Martin Maechler, Tim Hesterberg, 1989–1993. Port to R by Martin Maechler <maechler@stat.math.ethz.ch>.

**References**

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Wadsworth.

Schilling, M. F. and Watkins, A. E. (1994) A suggestion for sunflower plots. *The American Statistician*, **48**, 303–305.

**See Also**

[density](#)

**Examples**

```
## 'number' is computed automatically:
sunflowerplot(iris[, 3:4])
## Imitating Chambers et al., p.109, closely:
sunflowerplot(iris[, 3:4],cex=.2, cex.f=1, size=.035, seg.lwd=.8)

sunflowerplot(x=sort(2*round(rnorm(100))), y= round(rnorm(100),0),
              main = "Sunflower Plot of Rounded N(0,1)")

## A 'point process' {explicit 'number' argument}:
sunflowerplot(rnorm(100),rnorm(100), number=rpois(n=100,lambda=2),
              rotate=TRUE, main="Sunflower plot")
```

---

symbols	<i>Draw Symbols (Circles, Squares, Stars, Thermometers, Boxplots) on a Plot</i>
---------	---

---

### Description

This function draws symbols on a plot. One of six symbols; *circles*, *squares*, *rectangles*, *stars*, *thermometers*, and *boxplots*, can be plotted at a specified set of x and y coordinates. Specific aspects of the symbols, such as relative size, can be customized by additional parameters.

### Usage

```
symbols(x, y = NULL, circles, squares, rectangles, stars,
        thermometers, boxplots, inches = TRUE, add = FALSE,
        fg = 1, bg = NA, xlab = NULL, ylab = NULL, main = NULL,
        xlim = NULL, ylim = NULL, ...)
```

### Arguments

x, y	the x and y co-ordinates for the symbols. They can be specified in any way which is accepted by <a href="#">xy.coords</a> .
circles	a vector giving the radii of the circles.
squares	a vector giving the length of the sides of the squares.
rectangles	a matrix with two columns. The first column gives widths and the second the heights of rectangle symbols.
stars	a matrix with three or more columns giving the lengths of the rays from the center of the stars. NA values are replaced by zeroes.
thermometers	a matrix with three or four columns. The first two columns give the width and height of the thermometer symbols. If there are three columns, the third is taken as a proportion. The thermometers are filled from their base to this proportion of their height. If there are four columns, the third and fourth columns are taken as proportions. The thermometers are filled between these two proportions of their heights.
boxplots	a matrix with five columns. The first two columns give the width and height of the boxes, the next two columns give the lengths of the lower and upper whiskers and the fifth the proportion (with a warning if not in [0,1]) of the way up the box that the median line is drawn.
inches	If <code>inches</code> is <code>FALSE</code> , the units are taken to be those of the x axis. If <code>inches</code> is <code>TRUE</code> , the symbols are scaled so that the largest symbol is one inch in height. If a number is given the symbols are scaled to make largest symbol this height in inches.
add	if <code>add</code> is <code>TRUE</code> , the symbols are added to an existing plot, otherwise a new plot is created.
fg	colors the symbols are to be drawn in (the default is the value of the <code>col</code> graphics parameter).

<code>bg</code>	if specified, the symbols are filled with this color. The default is to leave the symbols unfilled.
<code>xlab</code>	the x label of the plot if <code>add</code> is not true; this applies to the following arguments as well. Defaults to the <a href="#">deparsed</a> expression used for <code>x</code> .
<code>ylab</code>	the y label of the plot.
<code>main</code>	a main title for the plot.
<code>xlim</code>	numeric of length 2 giving the x limits for the plot.
<code>ylim</code>	numeric of length 2 giving the y limits for the plot.
<code>...</code>	graphics parameters can also be passed to this function, as can the plot aspect ratio <code>asp</code> (see <a href="#">plot.window</a> ).

### Details

Observations which have missing coordinates or missing size parameters are not plotted. The exception to this is *stars*. In that case, the length of any rays which are NA is reset to zero.

Circles of radius zero are plotted at radius one pixel (which is device-dependent).

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

W. S. Cleveland (1985) *The Elements of Graphing Data*. Monterey, California: Wadsworth.

### See Also

[stars](#) for drawing *stars* with a bit more flexibility; [sunflowerplot](#).

### Examples

```
x <- 1:10
y <- sort(10*runif(10))
z <- runif(10)
z3 <- cbind(z, 2*runif(10), runif(10))
symbols(x, y, thermometers=cbind(.5, 1, z), inches=.5, fg = 1:10)
symbols(x, y, thermometers = z3, inches=FALSE)
text(x,y, apply(format(round(z3, dig=2)), 1, paste, collapse = ","),
      adj = c(-.2,0), cex = .75, col = "purple", xpd=NA)

## Note that example(trees) shows more sensible plots!
N <- nrow(trees)
attach(trees)
## Girth is diameter in inches
symbols(Height, Volume, circles=Girth/24, inches=FALSE,
        main="Trees' Girth")# xlab and ylab automatically
## Colors too:
palette(rainbow(N, end = 0.9))
symbols(Height, Volume, circles=Girth/16, inches=FALSE, bg = 1:N,
        fg="gray30", main="symbols(*, circles=Girth/16, bg = 1:N)")
palette("default"); detach()
```

---

 text

*Add Text to a Plot*


---

### Description

`text` draws the strings given in the vector `labels` at the coordinates given by `x` and `y`. `y` may be missing since `xy.coords(x, y)` is used for construction of the coordinates.

### Usage

```
text(x, ...)
```

```
## Default S3 method:
text(x, y = NULL, labels = seq(along = x), adj = NULL,
     pos = NULL, offset = 0.5, vfont = NULL,
     cex = 1, col = NULL, font = NULL, xpd = NULL, ...)
```

### Arguments

<code>x</code> , <code>y</code>	numeric vectors of coordinates where the text <code>labels</code> should be written. If the length of <code>x</code> and <code>y</code> differs, the shorter one is recycled.
<code>labels</code>	one or more character strings or expressions specifying the <i>text</i> to be written. An attempt is made to coerce other vectors (and factors) to character, and other language objects to expressions. If <code>labels</code> is longer than <code>x</code> and <code>y</code> , the coordinates are recycled to the length of <code>labels</code> .
<code>adj</code>	one or two values in $[0, 1]$ which specify the <code>x</code> (and optionally <code>y</code> ) adjustment of the labels. On most devices values outside that interval will also work.
<code>pos</code>	a position specifier for the text. If specified this overrides any <code>adj</code> value given. Values of 1, 2, 3 and 4, respectively indicate positions below, to the left of, above and to the right of the specified coordinates.
<code>offset</code>	when <code>pos</code> is specified, this value gives the offset of the label from the specified coordinate in fractions of a character width.
<code>vfont</code>	if a character vector of length 2 is specified, then Hershey vector fonts are used. The first element of the vector selects a typeface and the second element selects a style.
<code>cex</code>	numeric character <b>exp</b> ansion factor; multiplied by <code>par("cex")</code> yields the final character size.
<code>col</code> , <code>font</code>	the color and font to be used; these default to the values of the global graphical parameters in <code>par()</code> .
<code>xpd</code>	(where) should clipping take place? Defaults to <code>par("xpd")</code> .
<code>...</code>	further graphical parameters (from <code>par</code> ).

## Details

labels must be of type [character](#) or [expression](#) (or be coercible to such a type). In the latter case, quite a bit of mathematical notation is available such as sub- and superscripts, greek letters, fractions, etc.

`adj` allows *adjustment* of the text with respect to  $(x, y)$ . Values of 0, 0.5, and 1 specify left/bottom, middle and right/top, respectively. The default is for centered text, i.e., `adj = c(0.5, 0.5)`. Accurate vertical centering needs character metric information on individual characters, which is only available on some devices.

The `pos` and `offset` arguments can be used in conjunction with values returned by `identify` to recreate an interactively labelled plot.

Text can be rotated by using graphical parameters `srt` (see [par](#)); this rotates about the centre set by `adj`.

Graphical parameters `col`, `cex` and `font` can be vectors and will then be applied cyclically to the labels (and extra values will be ignored).

Labels whose `x`, `y`, `labels`, `cex` or `col` value is `NA` are omitted from the plot.

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[mtext](#), [title](#), [Hershey](#) for details on Hershey vector fonts, [plotmath](#) for details and more examples on mathematical annotation.

## Examples

```
plot(-1:1,-1:1, type = "n", xlab = "Re", ylab = "Im")
K <- 16; text(exp(1i * 2 * pi * (1:K) / K), col = 2)

## The following two examples use latin1 characters: these may not
## appear correctly (or be omitted entirely).
plot(1:10, 1:10, main = "text(...) examples\n~~~~~",
     sub = "R is GNU 1', but not ó ...")
mtext("ñISO-accentsz: ́ ́ é è ø å Å æ &AElig", side=3)
points(c(6,2), c(2,1), pch = 3, cex = 4, col = "red")
text(6, 2, "the text is CENTERED around (x,y) = (6,2) by default",
     cex = .8)
text(2, 1, "or Left/Bottom - JUSTIFIED at (2,1) by 'adj = c(0,0)'",
     adj = c(0,0))
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)", cex = .75)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i=1, n)))

## Two more latin1 examples
text(5,10.2,
     "Le français, c'est facile: Règles, Liberté, Egalité, Fraternité...")
text(5,9.8, "Jetzt no chli züritüütsch: (noch ein biSSchen Zürcher deutsch)")
```

---

title	<i>Plot Annotation</i>
-------	------------------------

---

### Description

This function can be used to add labels to a plot. Its first four principal arguments can also be used as arguments in most high-level plotting functions. They must be of type `character` or `expression`. In the latter case, quite a bit of mathematical notation is available such as sub- and superscripts, greek letters, fractions, etc.

### Usage

```
title(main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
      line = NA, outer = FALSE, ...)
```

### Arguments

main	The main title (on top) using font and size (character expansion) <code>par("font.main")</code> and color <code>par("col.main")</code> .
sub	Sub-title (at bottom) using font and size <code>par("font.sub")</code> and color <code>par("col.sub")</code> .
xlab	X axis label using font and character expansion <code>par("font.axis")</code> and color <code>par("col.axis")</code> .
ylab	Y axis label, same font attributes as <code>xlab</code> .
line	specifying a value for <code>line</code> overrides the default placement of labels, and places them this many lines from the plot.
outer	a logical value. If <code>TRUE</code> , the titles are placed in the outer margins of the plot.
...	further graphical parameters from <code>par</code> . Use e.g., <code>col.main</code> or <code>cex.sub</code> instead of just <code>col</code> or <code>cex</code> .

### Details

The labels passed to `title` can be simple strings or expressions, or they can be a list containing the string to be plotted, and a selection of the optional modifying graphical parameters `cex=`, `col=`, `font=`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`mtext`, `text`; `plotmath` for details on mathematical annotation.

**Examples**

```

plot(cars, main = "") # here, could use main directly
title(main = "Stopping Distance versus Speed")

plot(cars, main = "")
title(main = list("Stopping Distance versus Speed", cex=1.5,
                 col="red", font=3))

## Specifying "...":
plot(1, col.axis = "sky blue", col.lab = "thistle")
title("Main Title", sub = "sub title",
      cex.main = 2, font.main = 4, col.main = "blue",
      cex.sub = 0.75, font.sub = 3, col.sub = "red")

x <- seq(-4, 4, len = 101)
y <- cbind(sin(x), cos(x))
matplot(x, y, type = "l", xaxt = "n",
        main = expression(paste(plain(sin) * phi, " and ",
                                plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
        col.main = "blue")
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     lab = expression(-pi, -pi/2, 0, pi/2, pi))
abline(h = 0, v = pi/2 * c(-1,1), lty = 2, lwd = .1, col = "gray70")

```

xy.coords

*Extracting Plotting Structures***Description**

`xy.coords` is used by many functions to obtain x and y coordinates for plotting. The use of this common mechanism across all R functions produces a measure of consistency.

**Usage**

```
xy.coords(x, y, xlab = NULL, ylab = NULL, log = NULL, recycle = FALSE)
```

**Arguments**

<code>x, y</code>	the x and y coordinates of a set of points. Alternatively, a single argument <code>x</code> can be provided.
<code>xlab, ylab</code>	names for the x and y variables to be extracted.
<code>log</code>	character, "x", "y" or both, as for <code>plot</code> . Sets negative values to <code>NA</code> and gives a warning.
<code>recycle</code>	logical; if TRUE, recycle ( <code>rep</code> ) the shorter of <code>x</code> or <code>y</code> if their lengths differ.

## Details

An attempt is made to interpret the arguments `x` and `y` in a way suitable for plotting.

If `y` is missing and `x` is a

**formula:** of the form `yvar ~ xvar`. `xvar` and `yvar` are used as `x` and `y` variables.

**list:** containing components `x` and `y`, these are used to define plotting coordinates.

**time series:** the `x` values are taken to be `time(x)` and the `y` values to be the time series.

**matrix with two columns:** the first is assumed to contain the `x` values and the second the `y` values.

In any other case, the `x` argument is coerced to a vector and returned as `y` component where the resulting `x` is just the index vector `1:n`. In this case, the resulting `xlab` component is set to "Index".

If `x` (after transformation as above) inherits from class "POSIXt" it is coerced to class "POSIXct".

## Value

A list with the components

<code>x</code>	numeric (i.e., "double") vector of abscissa values.
<code>y</code>	numeric vector of the same length as <code>x</code> .
<code>xlab</code>	<code>character(1)</code> or <code>NULL</code> , the 'label' of <code>x</code> .
<code>ylab</code>	<code>character(1)</code> or <code>NULL</code> , the 'label' of <code>y</code> .

## See Also

[plot.default](#), [lines](#), [points](#) and [lowess](#) are examples of functions which use this mechanism.

## Examples

```
xy.coords(stats::fft(c(1:10)), NULL)

with(cars, xy.coords(dist ~ speed, NULL)$xlab ) # = "speed"

xy.coords(1:3, 1:2, recycle=TRUE)
xy.coords(-2:10, NULL, log="y")
##> warning: 3 y values <=0 omitted ..
```

**Description**

Utility for obtaining consistent x, y and z coordinates and labels for three dimensional (3D) plots.

**Usage**

```
xyz.coords(x, y, z, xlab = NULL, ylab = NULL, zlab = NULL,
          log = NULL, recycle = FALSE)
```

**Arguments**

<code>x, y, z</code>	the x, y and z coordinates of a set of points. Alternatively, a single argument <code>x</code> can be provided. In this case, an attempt is made to interpret the argument in a way suitable for plotting. If the argument is a formula <code>zvar ~ xvar + yvar</code> , <code>xvar</code> , <code>yvar</code> and <code>zvar</code> are used as x, y and z variables; if the argument is a list containing components <code>x</code> , <code>y</code> and <code>z</code> , these are assumed to define plotting coordinates; if the argument is a matrix with three columns, the first is assumed to contain the x values, etc. Alternatively, two arguments <code>x</code> and <code>y</code> can be provided. One may be real, the other complex; in any other case, the arguments are coerced to vectors and the values plotted against their indices.
<code>xlab, ylab, zlab</code>	names for the x, y and z variables to be extracted.
<code>log</code>	character, "x", "y", "z" or combinations. Sets negative values to <code>NA</code> and gives a warning.
<code>recycle</code>	logical; if TRUE, recycle ( <a href="#">rep</a> ) the shorter ones of x, y or z if their lengths differ.

**Value**

A list with the components

<code>x</code>	numeric (i.e., <a href="#">double</a> ) vector of abscissa values.
<code>y</code>	numeric vector of the same length as <code>x</code> .
<code>z</code>	numeric vector of the same length as <code>x</code> .
<code>xlab</code>	<code>character(1)</code> or NULL, the axis label of x.
<code>ylab</code>	<code>character(1)</code> or NULL, the axis label of y.
<code>zlab</code>	<code>character(1)</code> or NULL, the axis label of z.

**Author(s)**

Uwe Ligges and Martin Maechler

**See Also**

[xy.coords](#) for 2D.

**Examples**

```
xyz.coords(data.frame(10*1:9, -4), y = NULL, z = NULL)

xyz.coords(1:6, stats::fft(1:6), z = NULL, xlab = "X", ylab = "Y")

y <- 2 * (x2 <- 10 + (x1 <- 1:10))
xyz.coords(y ~ x1 + x2, y = NULL, z = NULL)

xyz.coords(data.frame(x = -1:9, y = 2:12, z = 3:13), y = NULL, z = NULL,
               log = "xy")
##> Warning message: 2 x values <= 0 omitted ...
```

## Chapter 5

# The grid package

---

<code>absolute.size</code>	<i>Absolute Size of a Grob</i>
----------------------------	--------------------------------

---

### Description

This function converts a unit object into absolute units. Absolute units are unaffected, but non-absolute units are converted into "null" units.

### Usage

```
absolute.size(unit)
```

### Arguments

<code>unit</code>	An object of class "unit".
-------------------	----------------------------

### Details

Absolute units are things like "inches", "cm", and "lines". Non-absolute units are "npc" and "native".

This function is designed to be used in `widthDetails` and `heightDetails` methods.

### Value

An object of class "unit".

### Author(s)

Paul Murrell

### See Also

[widthDetails](#) and [heightDetails](#) methods.

---

convertNative	<i>Convert a Unit Object to Native units</i>
---------------	--

---

### Description

**This function is deprecated in grid version 0.8 and will be made defunct in grid version 1.9**

You should use the `convertUnit()` function or one of its close allies instead.

This function returns a numeric vector containing the specified x or y locations or dimensions, converted to "user" or "data" units, relative to the current viewport.

### Usage

```
convertNative(unit, dimension="x", type="location")
```

### Arguments

<code>unit</code>	A unit object.
<code>dimension</code>	Either "x" or "y".
<code>type</code>	Either "location" or "dimension".

### Value

A numeric vector.

### WARNING

If you draw objects based on output from these conversion functions, then resize your device, the objects will be drawn incorrectly – the base R display list will not recalculate these conversions. This means that you can only rely on the results of these calculations if the size of your device is fixed.

### Author(s)

Paul Murrell

### See Also

[grid.convert, unit](#)

### Examples

```
grid.newpage()
pushViewport(viewport(width=unit(.5, "npc"),
                      height=unit(.5, "npc")))
grid.rect()
w <- convertNative(unit(1, "inches"))
h <- convertNative(unit(1, "inches"), "y")
```

```
# This rectangle starts off life as 1in square, but if you
# resize the device it will no longer be 1in square
grid.rect(width=unit(w, "native"), height=unit(h, "native"),
          gp=gpar(col="red"))
popViewport(1)

# How to use grid.convert(), etc instead
convertNative(unit(1, "inches")) ==
  convertX(unit(1, "inches"), "native", valueOnly=TRUE)
convertNative(unit(1, "inches"), "y", "dimension") ==
  convertHeight(unit(1, "inches"), "native", valueOnly=TRUE)
```

---

dataViewport

*Create a Viewport with Scales based on Data*


---

### Description

This is a convenience function for producing a viewport with x- and/or y-scales based on numeric values passed to the function.

### Usage

```
dataViewport(xData = NULL, yData = NULL, xscale = NULL,
            yscale = NULL, extension = 0.05, ...)
```

### Arguments

xData	A numeric vector of data.
yData	A numeric vector of data.
xscale	A numeric vector (length 2).
yscale	A numeric vector (length 2).
extension	A numeric.
...	All other arguments will be passed to a call to the <code>viewport()</code> function.

### Details

If `xscale` is not specified then the values in `x` are used to generate an x-scale based on the range of `x`, extended by the proportion specified in `extension`. Similarly for the y-scale.

### Value

A grid viewport object.

### Author(s)

Paul Murrell

**See Also**

[viewport](#) and [plotViewport](#).

---

drawDetails

*Customising grid Drawing*

---

**Description**

These generic hook functions are called whenever a grid grob is drawn. They provide an opportunity for customising the drawing of a new class derived from grob (or gTree).

**Usage**

```
drawDetails(x, recording)
draw.details(x, recording)
preDrawDetails(x)
postDrawDetails(x)
```

**Arguments**

<code>x</code>	A grid grob.
<code>recording</code>	A logical value indicating whether a grob is being added to the display list or redrawn from the display list.

**Details**

These functions are called by the `grid.draw` methods for grobs and gTrees.

`preDrawDetails` is called first during the drawing of a grob. This is where any additional viewports should be pushed (see, for example, `grid::preDrawDetails.frame`). Note that the default behaviour for grobs is to push any viewports in the `vp` slot, and for gTrees is to also push and up any viewports in the `childrenvp` slot so there is typically nothing to do here.

`drawDetails` is called next and is where any additional calculations and graphical output should occur (see, for example, `grid::drawDetails.xaxis`). Note that the default behaviour for gTrees is to draw all grobs in the `children` slot so there is typically nothing to do here.

`postDrawDetails` is called last and should reverse anything done in `preDrawDetails` (i.e., `pop` or `up` any viewports that were pushed; again, see, for example, `grid::postDrawDetails.frame`). Note that the default behaviour for grobs is to `pop` any viewports that were pushed so there is typically nothing to do here.

Note that `preDrawDetails` and `postDrawDetails` are also called in the calculation of "grobwidth" and "grobheight" units.

**Value**

None of these functions are expected to return a value.

**Author(s)**

Paul Murrell

**See Also**[grid.draw](#)

---

`editDetails`*Customising grid Editing*

---

**Description**

This generic hook function is called whenever a grid grob is edited via `grid.edit` or `editGrob`. This provides an opportunity for customising the editing of a new class derived from `grob` (or `gTree`).

**Usage**

```
editDetails(x, specs)
```

**Arguments**

<code>x</code>	A grid grob.
<code>specs</code>	A list of named elements. The names indicate the grob slots to modify and the values are the new values for the slots.

**Details**

This function is called by `grid.edit` and `editGrob`. A method should be written for classes derived from `grob` or `gTree` if a change in a slot has an effect on other slots in the grob or children of a `gTree` (e.g., see `grid:::editDetails.xaxis`).

Note that the slot already has the new value.

**Value**

The function **MUST** return the modified grob.

**Author(s)**

Paul Murrell

**See Also**[grid.edit](#)

gEdit

*Create and Apply Edit Objects***Description**

The functions `gEdit` and `gEditList` create objects representing an edit operation (essentially a list of arguments to `editGrob`).

The functions `applyEdit` and `applyEdits` apply one or more edit operations to a graphical object.

These functions are most useful for developers creating new graphical functions and objects.

**Usage**

```
gEdit(...)
gEditList(...)
applyEdit(x, edit)
applyEdits(x, edits)
```

**Arguments**

<code>...</code>	one or more arguments to the <code>editGrob</code> function (for <code>gEdit</code> ) or one or more "gEdit" objects (for <code>gEditList</code> ).
<code>x</code>	a grob (grid graphical object).
<code>edit</code>	a "gEdit" object.
<code>edits</code>	either a "gEdit" object or a "gEditList" object.

**Value**

`gEdit` returns an object of class "gEdit".

`gEditList` returns an object of class "gEditList".

`applyEdit` and `applyEditList` return the modified grob.

**Author(s)**

Paul Murrell

**See Also**

[grob editGrob](#)

**Examples**

```
grid.rect(gp=gpar(col="red"))
# same thing, but more verbose
grid.draw(applyEdit(rectGrob(), gEdit(gp=gpar(col="red"))))
```

**Description**

`gpar ( )` should be used to create a set of graphical parameter settings. It returns an object of class "gpar". This is basically a list of name-value pairs.

`get.gpar ( )` can be used to query the current graphical parameter settings.

**Usage**

```
gpar ( . . . )
get.gpar ( names = NULL )
```

**Arguments**

`. . .` Any number of named arguments.  
`names` A character vector of valid graphical parameter names.

**Details**

All grid viewports and (predefined) graphical objects have a slot called `gp`, which contains a "gpar" object. When a viewport is pushed onto the viewport stack and when a graphical object is drawn, the settings in the "gpar" object are enforced. In this way, the graphical output is modified by the `gp` settings until the graphical object has finished drawing, or until the viewport is popped off the viewport stack, or until some other viewport or graphical object is pushed or begins drawing.

Valid parameter names are:

<code>col</code>	Colour for lines and borders.
<code>fill</code>	Colour for filling rectangles, polygons, ...
<code>alpha</code>	Alpha channel for transparency
<code>lty</code>	Line type
<code>lwd</code>	Line width
<code>lineend</code>	Line end style (round, butt, square)
<code>linejoin</code>	Line join style (round, mitre, bevel)
<code>linemitre</code>	Line mitre limit (number greater than 1)
<code>fontsize</code>	The size of text (in points)
<code>cex</code>	Multiplier applied to fontsize
<code>fontfamily</code>	The font family
<code>fontface</code>	The font face (bold, italic, ...)
<code>lineheight</code>	The height of a line as a multiple of the size of text
<code>font</code>	Font face (alias for fontface; for backward compatibility)

The `alpha` setting is combined with the alpha channel for individual colours by multiplying (with both alpha settings normalised to the range 0 to 1).

The size of text is `fontsize*cex`. The size of a line is `fontsize*cex*lineheight`.

The `cex` setting is cumulative; if a viewport is pushed with a `cex` of 0.5 then another viewport is pushed with a `cex` of 0.5, the effective `cex` is 0.25.

Changes to the `fontfamily` may be ignored by some devices, but is supported by PostScript, PDF, X11, Windows, and Quartz. The `fontfamily` may be used to specify one of the Hershey Font families (e.g., `HersheySerif`) and this specification will be honoured on all devices.

The specification of `fontface` can be an integer or a string. If an integer, then it follows the R base graphics standard: 1 = plain, 2 = bold, 3 = italic, 4 = bold italic. If a string, then valid values are: "plain", "bold", "italic", "oblique", and "bold.italic". For the special case of the `HersheySerif` font family, "cyrillic", "cyrillic.oblique", and "EUC" are also available.

Specifying the value `NULL` for a parameter is the same as not specifying any value for that parameter, except for `col` and `fill`, where `NULL` indicates not to draw a border or not to fill an area (respectively).

All parameter values can be vectors of multiple values. (This will not always make sense – for example, viewports will only take notice of the first parameter value.)

`get.gpar()` returns all current graphical parameter settings.

### Value

An object of class "gpar".

### Author(s)

Paul Murrell

### See Also

[Hershey](#).

### Examples

```
get.gpar()
gpar(col = "red")
gpar(col = "blue", lty = "solid", lwd = 3, fontsize = 16)
get.gpar(c("col", "lty"))
grid.newpage()
vp <- viewport(w = .8, h = .8, gp = gpar(col="blue"))
grid.draw(gTree(children=gList(rectGrob(gp = gpar(col="red")),
                             textGrob(paste("The rect is its own colour (red)",
                                             "but this text is the colour",
                                             "set by the gTree (green)",
                                             sep = "\n"))),
          gp = gpar(col="green"), vp = vp))
grid.text("This text is the colour set by the viewport (blue)",
          y = 1, just = c("center", "bottom"),
          gp = gpar(fontsize=20), vp = vp)
grid.newpage()
## example with multiple values for a parameter
```

```
pushViewport(viewport())
grid.points(1:10/11, 1:10/11, gp = gpar(col=1:10))
popViewport()
```

---

gPath

*Concatenate Grob Names*

---

## Description

This function can be used to generate a grob path for use in `grid.edit` and friends.

A grob path is a list of nested grob names.

## Usage

```
gPath(...)
```

## Arguments

... Character values which are grob names.

## Details

Grob names must only be unique amongst grobs which share the same parent in a `gTree`.

This function can be used to generate a specification for a grob that includes the grob's parent's name (and the name of its parent and so on).

For interactive use, it is possible to directly specify a path, but it is strongly recommended that this function is used otherwise in case the path separator is changed in future versions of `grid`.

## Value

A `gPath` object.

## See Also

[grob](#), [editGrob](#), [addGrob](#), [removeGrob](#), [getGrob](#), [setGrob](#)

## Examples

```
gPath("g1", "g2")
```

---

 Grid

*Grid Graphics*


---

### Description

General information about the grid graphics package.

### Details

Grid graphics provides an alternative to the standard R graphics. The user is able to define arbitrary rectangular regions (called *viewports*) on the graphics device and define a number of coordinate systems for each region. Drawing can be specified to occur in any viewport using any of the available coordinate systems.

Grid graphics and standard R graphics do not mix!

Type `library(help = grid)` to see a list of (public) Grid graphics functions.

### Author(s)

Paul Murrell

### See Also

[viewport](#), [grid.layout](#), and [unit](#).

### Examples

```
## Diagram of a simple layout
grid.show.layout(grid.layout(4,2,
  heights=unit(rep(1, 4),
    c("lines", "lines", "lines", "null")),
  widths=unit(c(1, 1), "inches")))
## Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
  w=unit(1, "inches"), h=unit(1, "inches")))
## A flash plotting example
grid.multipanel(vp=viewport(0.5, 0.5, 0.8, 0.8))
```

---

 Grid Viewports

*Create a Grid Viewport*


---

### Description

These functions create viewports, which describe a rectangular regions on a graphics device and define a number of coordinate systems within those regions.

**Usage**

```
viewport(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
        width = unit(1, "npc"), height = unit(1, "npc"),
        default.units = "npc", just = "centre",
        gp = gpar(), clip = "inherit",
        xscale = c(0, 1), yscale = c(0, 1),
        angle = 0,
        layout = NULL,
        layout.pos.row = NULL, layout.pos.col = NULL,
        name = NULL)
vpList(...)
vpStack(...)
vpTree(parent, children)
```

**Arguments**

<code>x</code>	A numeric vector or unit object specifying x-location.
<code>y</code>	A numeric vector or unit object specifying y-location.
<code>width</code>	A numeric vector or unit object specifying width.
<code>height</code>	A numeric vector or unit object specifying height.
<code>default.units</code>	A string indicating the default units to use if <code>x</code> , <code>y</code> , <code>width</code> , or <code>height</code> are only given as numeric vectors.
<code>just</code>	A string vector specifying the justification of the viewport relative to its (x, y) location. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible values are: "left", "right", "centre", "center", "bottom", and "top".
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>clip</code>	One of "on", "inherit", or "off", indicating whether to clip to the extent of this viewport, inherit the clipping region from the parent viewport, or turn clipping off altogether. For back-compatibility, a logical value of <code>TRUE</code> corresponds to "on" and <code>FALSE</code> corresponds to "inherit".
<code>xscale</code>	A numeric vector of length two indicating the minimum and maximum on the x-scale.
<code>yscale</code>	A numeric vector of length two indicating the minimum and maximum on the y-scale.
<code>angle</code>	A numeric value indicating the angle of rotation of the viewport. Positive values indicate the amount of rotation, in degrees, anticlockwise from the positive x-axis.
<code>layout</code>	A Grid layout object which splits the viewport into subregions.
<code>layout.pos.row</code>	A numeric vector giving the rows occupied by this viewport in its parent's layout.

<code>layout.pos.col</code>	A numeric vector giving the columns occupied by this viewport in its parent's layout.
<code>name</code>	A character value to uniquely identify the viewport once it has been pushed onto the viewport tree.
<code>...</code>	Any number of grid viewport objects.
<code>parent</code>	A grid viewport object.
<code>children</code>	A <code>vpList</code> object.

### Details

The location and size of a viewport are relative to the coordinate systems defined by the viewport's parent (either a graphical device or another viewport). The location and size can be specified in a very flexible way by specifying them with unit objects. When specifying the location of a viewport, specifying both `layout.pos.row` and `layout.pos.col` as `NULL` indicates that the viewport ignores its parent's layout and specifies its own location and size (via its `locn`). If only one of `layout.pos.row` and `layout.pos.col` is `NULL`, this means occupy ALL of the appropriate row(s)/column(s). For example, `layout.pos.row = 1` and `layout.pos.col = NULL` means occupy all of row 1. Specifying non-`NULL` values for both `layout.pos.row` and `layout.pos.col` means occupy the intersection of the appropriate rows and columns. If a vector of length two is specified for `layout.pos.row` or `layout.pos.col`, this indicates a range of rows or columns to occupy. For example, `layout.pos.row = c(1, 3)` and `layout.pos.col = c(2, 4)` means occupy cells in the intersection of rows 1, 2, and 3, and columns, 2, 3, and 4.

Clipping obeys only the most recent viewport clip setting. For example, if you clip to viewport1, then clip to viewport2, the clipping region is determined wholly by viewport2, the size and shape of viewport1 is irrelevant (until viewport2 is popped of course).

If a viewport is rotated (because of its own `angle` setting or because it is within another viewport which is rotated) then the `clip` flag is ignored.

Viewport names need not be unique. When pushed, viewports sharing the same parent must have unique names, which means that if you push a viewport with the same name as an existing viewport, the existing viewport will be replaced in the viewport tree. A viewport name can be any valid R symbol name, but grid uses the reserved name "ROOT" for the top-level viewport. Also, when specifying a viewport name in `downViewport` and `seekViewport`, it is possible to provide a viewport path, which consists of several names concatenated using the separator (currently `::`). Consequently, it is not advisable to use this separator in viewport names.

The viewports in a `vpList` are pushed in parallel. The viewports in a `vpStack` are pushed in series. When a `vpTree` is pushed, the parent is pushed first, then the children are pushed in parallel.

### Value

An R object of class `viewport`.

### Author(s)

Paul Murrell

**See Also**

[Grid](#), [pushViewport](#), [popViewport](#), [downViewport](#), [seekViewport](#), [upViewport](#), [unit](#), [grid.layout](#), [grid.show.layout](#).

**Examples**

```
# Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                           w=unit(1, "inches"), h=unit(1, "inches")))
# Demonstrate viewport clipping
clip.demo <- function(i, j, clip1, clip2) {
  pushViewport(viewport(layout.pos.col=i,
                        layout.pos.row=j))
  pushViewport(viewport(width=0.6, height=0.6, clip=clip1))
  grid.rect(gp=gpar(fill="white"))
  grid.circle(r=0.55, gp=gpar(col="red", fill="pink"))
  popViewport()
  pushViewport(viewport(width=0.6, height=0.6, clip=clip2))
  grid.polygon(x=c(0.5, 1.1, 0.6, 1.1, 0.5, -0.1, 0.4, -0.1),
              y=c(0.6, 1.1, 0.5, -0.1, 0.4, -0.1, 0.5, 1.1),
              gp=gpar(col="blue", fill="light blue"))
  popViewport(2)
}

grid.newpage()
grid.rect(gp=gpar(fill="grey"))
pushViewport(viewport(layout=grid.layout(2, 2)))
clip.demo(1, 1, FALSE, FALSE)
clip.demo(1, 2, TRUE, FALSE)
clip.demo(2, 1, FALSE, TRUE)
clip.demo(2, 2, TRUE, TRUE)
popViewport()
# Demonstrate turning clipping off
grid.newpage()
pushViewport(viewport(w=.5, h=.5, clip="on"))
grid.rect()
grid.circle(r=.6, gp=gpar(lwd=10))
pushViewport(viewport(clip="inherit"))
grid.circle(r=.6, gp=gpar(lwd=5, col="grey"))
pushViewport(viewport(clip="off"))
grid.circle(r=.6)
popViewport(3)
# Demonstrate vpList, vpStack, and vpTree
grid.newpage()
tree <- vpTree(viewport(w=0.8, h=0.8, name="A"),
               vpList(vpStack(viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                     just=c("left", "bottom"), name="B"),
                       viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                     just=c("left", "bottom"), name="C"),
                       viewport(x=0.1, y=0.1, w=0.5, h=0.5,
                                     just=c("left", "bottom"), name="D")),
                     viewport(x=0.5, w=0.4, h=0.9,
```

```

                                just="left", name="E"))
pushViewport(tree)
for (i in LETTERS[1:5]) {
  seekViewport(i)
  grid.rect()
  grid.text(current.vpTree(FALSE),
            x=unit(1, "mm"), y=unit(1, "npc") - unit(1, "mm"),
            just=c("left", "top"),
            gp=gpar(fontsize=8))
}

```

---

grid.add

*Add a Grid Graphical Object*


---

### Description

Add a grob to a gTree or a descendant of a gTree.

### Usage

```
grid.add(gPath, child, strict = FALSE, grep = FALSE, global = FALSE,
        allDevices = FALSE, redraw = TRUE)
```

```
addGrob(gTree, child, gPath = NULL, strict = FALSE, grep = FALSE,
        global = FALSE)
setChildren(x, children)
```

### Arguments

gTree, x	A gTree object.
gPath	A gPath object. For <code>grid.add</code> this specifies a gTree on the display list. For <code>addGrob</code> this specifies a descendant of the specified gTree.
child	A grob object.
children	A gList object.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., <code>c(TRUE, FALSE)</code> means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
redraw	A logical value to indicate whether to redraw the grob.

**Details**

addGrob copies the specified grob and returns a modified grob.

grid.add destructively modifies a grob on the display list. If redraw is TRUE it then redraws everything to reflect the change.

setChildren is a basic function for setting all children of a gTree at once (instead of repeated calls to addGrob).

**Value**

addGrob returns a grob object; grid.add returns NULL.

**Author(s)**

Paul Murrell

**See Also**

[grob](#), [getGrob](#), [addGrob](#), [removeGrob](#).

**Examples**

---

grid.arrows	<i>Draw Arrows</i>
-------------	--------------------

---

**Description**

Functions to create and draw arrows at either end of a line, or at either end of a line.to, lines, or segments grob.

**Usage**

```
grid.arrows(x = c(0.25, 0.75), y = 0.5, default.units = "npc",
            grob = NULL,
            angle = 30, length = unit(0.25, "inches"),
            ends = "last", type = "open", name = NULL,
            gp = gpar(), draw = TRUE, vp = NULL)
arrowsGrob(x = c(0.25, 0.75), y = 0.5, default.units = "npc",
           grob = NULL,
           angle = 30, length = unit(0.25, "inches"),
           ends = "last", type = "open", name = NULL,
           gp = gpar(), vp = NULL)
```

**Arguments**

<code>x</code>	A numeric vector or unit object specifying x-values.
<code>y</code>	A numeric vector or unit object specifying y-values.
<code>default.units</code>	A string indicating the default units to use if <code>x</code> or <code>y</code> are only given as numeric vectors.
<code>grob</code>	A grob to add arrows to; currently can only be a <code>line.to</code> , <code>lines</code> , or <code>segments</code> grob.
<code>angle</code>	A numeric specifying (half) the width of the arrow head (in degrees).
<code>length</code>	A unit object specifying the length of the arrow head.
<code>ends</code>	One of "first", "last", or "both", indicating which end of the line to add arrow heads.
<code>type</code>	Either "open" or "closed" to indicate the type of arrow head.
<code>name</code>	A character identifier.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>vp</code>	A Grid viewport object (or NULL).

**Details**

Both functions create an arrows grob (a graphical object describing arrows), but only `grid.arrows()` draws the arrows (and then only if `draw` is TRUE).

If the `grob` argument is specified, this overrides any `x` and/or `y` arguments.

**Value**

An arrows grob. `grid.arrows()` returns the value invisibly.

**Author(s)**

Paul Murrell

**See Also**

[Grid](#), [viewport](#), [grid.line.to](#), [grid.lines](#), [grid.segments](#)

**Examples**

```
pushViewport(viewport(layout=grid.layout(2, 4)))
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows()
popViewport()
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=1))
```

```

grid.rect(gp=gpar(col="grey"))
grid.arrows(angle=15, type="closed")
popViewport()
pushViewport(viewport(layout.pos.col=3,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows(angle=5, length=unit(0.1, "npc"),
            type="closed", gp=gpar(fill="white"))
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=1))
grid.rect(gp=gpar(col="grey"))
grid.arrows(x=unit(0:80/100, "npc"),
            y=unit(1 - (0:80/100)^2, "npc"))
popViewport()
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
grid.arrows(ends="both")
popViewport()
pushViewport(viewport(layout.pos.col=2,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Recycling arguments
grid.arrows(x=unit(1:10/11, "npc"), y=unit(1:3/4, "npc"))
popViewport()
pushViewport(viewport(layout.pos.col=3,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Drawing arrows on a segments grob
gs <- segmentsGrob(x0=unit(1:4/5, "npc"),
                  xl=unit(1:4/5, "npc"))
grid.arrows(grob=gs, length=unit(0.1, "npc"),
            type="closed", gp=gpar(fill="white"))
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=2))
grid.rect(gp=gpar(col="grey"))
# Arrows on a lines grob
# Name these because going to grid.edit them later
gl <- linesGrob(name="curve", x=unit(0:80/100, "npc"),
               y=unit((0:80/100)^2, "npc"))
grid.arrows(name="arrowOnLine", grob=gl, angle=15, type="closed",
            gp=gpar(fill="black"))
popViewport()
pushViewport(viewport(layout.pos.col=1,
                      layout.pos.row=2))
grid.move.to(x=0.5, y=0.8)
popViewport()
pushViewport(viewport(layout.pos.col=4,
                      layout.pos.row=1))
# Arrows on a line.to grob
glt <- lineToGrob(x=0.5, y=0.2, gp=gpar(lwd=3))

```

```
grid.arrows(grob=glt, ends="first", gp=gpar(lwd=3))
popViewport(2)
grid.edit(gPath("arrowOnLine", "curve"), y=unit((0:80/100)^3, "npc"))
```

---

grid.circle	<i>Draw a Circle</i>
-------------	----------------------

---

## Description

Functions to create and draw a circle.

## Usage

```
grid.circle(x=0.5, y=0.5, r=0.5, default.units="npc", name=NULL,
            gp=gpar(), draw=TRUE, vp=NULL)
circleGrob(x=0.5, y=0.5, r=0.5, default.units="npc", name=NULL,
           gp=gpar(), vp=NULL)
```

## Arguments

<code>x</code>	A numeric vector or unit object specifying x-locations.
<code>y</code>	A numeric vector or unit object specifying y-locations.
<code>r</code>	A numeric vector or unit object specifying radii.
<code>default.units</code>	A string indicating the default units to use if <code>x</code> , <code>y</code> , <code>width</code> , or <code>height</code> are only given as numeric vectors.
<code>name</code>	A character identifier.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>vp</code>	A Grid viewport object (or <code>NULL</code> ).

## Details

Both functions create a circle grob (a graphical object describing a circle), but only `grid.circle()` draws the circle (and then only if `draw` is `TRUE`).

The radius may be given in any units; if the units are *relative* (e.g., "npc" or "native") then the radius will be different depending on whether it is interpreted as a width or as a height. In such cases, the smaller of these two values will be the result. To see the effect, type `grid.circle()` and adjust the size of the window.

## Value

A circle grob. `grid.circle()` returns the value invisibly.

**Author(s)**

Paul Murrell

**See Also**[Grid](#), [viewport](#)

---

grid.collection      *Create a Coherent Group of Grid Graphical Objects*

---

**Description**

This function is deprecated; please use `gTree`.

This function creates a graphical object which contains several other graphical objects. When it is drawn, it draws all of its children.

It may be convenient to name the elements of the collection.

**Usage**

```
grid.collection(..., gp=gpar(), draw=TRUE, vp=NULL)
```

**Arguments**

...	Zero or more objects of class "grob".
gp	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
draw	A logical value to indicate whether to produce graphical output.
vp	A Grid viewport object (or <code>NULL</code> ).

**Value**

A collection grob.

**Author(s)**

Paul Murrell

**See Also**[grid.grob](#).**Examples**

---

grid.convert	<i>Convert Between Different grid Coordinate Systems</i>
--------------	--

---

### Description

These functions take a unit object and convert it to an equivalent unit object in a different coordinate system.

### Usage

```

convertX(x, unitTo, valueOnly = FALSE)
convertY(x, unitTo, valueOnly = FALSE)
convertWidth(x, unitTo, valueOnly = FALSE)
convertHeight(x, unitTo, valueOnly = FALSE)
convertUnit(x, unitTo,
            axisFrom = "x", typeFrom = "location",
            axisTo = axisFrom, typeTo = typeFrom,
            valueOnly = FALSE)
grid.convertX(x, unitTo, valueOnly = FALSE)
grid.convertY(x, unitTo, valueOnly = FALSE)
grid.convertWidth(x, unitTo, valueOnly = FALSE)
grid.convertHeight(x, unitTo, valueOnly = FALSE)
grid.convert(x, unitTo,
            axisFrom = "x", typeFrom = "location",
            axisTo = axisFrom, typeTo = typeFrom,
            valueOnly = FALSE)

```

### Arguments

x	A unit object.
unitTo	The coordinate system to convert the unit to. See the <a href="#">unit</a> function for valid coordinate systems.
axisFrom	Either "x" or "y" to indicate whether the unit object represents a value in the x- or y-direction.
typeFrom	Either "location" or "dimension" to indicate whether the unit object represents a location or a length.
axisTo	Same as axisFrom, but applies to the unit object that is to be created.
typeTo	Same as typeFrom, but applies to the unit object that is to be created.
valueOnly	A logical indicating. If TRUE then the function does not return a unit object, but rather only the converted numeric values.

### Details

The convertUnit function allows for general-purpose conversions. The other four functions are just more convenient front-ends to it for the most common conversions.

The conversions occur within the current viewport.

It is not currently possible to convert to all valid coordinate systems (e.g., "strwidth" or "grob-width"). I'm not sure if all of these are impossible, they just seem implausible at this stage.

In normal usage of grid, these functions should not be necessary. If you want to express a location or dimension in inches rather than user coordinates then you should simply do something like `unit(1, "inches")` rather than something like `unit(0.134, "native")`.

In some cases, however, it is necessary for the user to perform calculations on a unit value and this function becomes necessary. In such cases, please take note of the warning below.

The `grid.*` versions are just previous incarnations which have been deprecated.

### Value

A unit object in the specified coordinate system (unless `valueOnly` is TRUE in which case the returned value is a numeric).

### Warning

The conversion is only valid for the current device size. If the device is resized then at least some conversions will become invalid. For example, suppose that I create a unit object as follows: `oneinch <- convertUnit(unit(1, "inches"), "native")`. Now if I resize the device, the unit object in `oneinch` no longer corresponds to a physical length of 1 inch.

### Author(s)

Paul Murrell

### See Also

[unit](#)

### Examples

```
## A tautology
convertX(unit(1, "inches"), "inches")
## The physical units
convertX(unit(2.54, "cm"), "inches")
convertX(unit(25.4, "mm"), "inches")
convertX(unit(72.27, "points"), "inches")
convertX(unit(1/12*72.27, "picas"), "inches")
convertX(unit(72, "bigpts"), "inches")
convertX(unit(1157/1238*72.27, "dida"), "inches")
convertX(unit(1/12*1157/1238*72.27, "cicero"), "inches")
convertX(unit(65536*72.27, "scaledpts"), "inches")
convertX(unit(1/2.54, "inches"), "cm")
convertX(unit(1/25.4, "inches"), "mm")
convertX(unit(1/72.27, "inches"), "points")
convertX(unit(1/(1/12*72.27), "inches"), "picas")
convertX(unit(1/72, "inches"), "bigpts")
convertX(unit(1/(1157/1238*72.27), "inches"), "dida")
convertX(unit(1/(1/12*1157/1238*72.27), "inches"), "cicero")
```

```
convertX(unit(1/(65536*72.27), "inches"), "scaledpts")

pushViewport(viewport(width=unit(1, "inches"),
                      height=unit(2, "inches"),
                      xscale=c(0, 1),
                      yscale=c(1, 3)))

## Location versus dimension
convertY(unit(2, "native"), "inches")
convertHeight(unit(2, "native"), "inches")
## From "x" to "y" (the conversion is via "inches")
convertUnit(unit(1, "native"), "native",
            axisFrom="x", axisTo="y")
## Convert several values at once
convertX(unit(c(0.5, 2.54), c("npc", "cm")),
         c("inches", "native"))
popViewport()
## Convert a complex unit
convertX(unit(1, "strwidth", "Hello"), "native")
```

---

grid.copy

*Make a Copy of a Grid Graphical Object*

---

### Description

This function is redundant and will disappear in future versions.

### Usage

```
grid.copy(grob)
```

### Arguments

grob            A grob object.

### Value

A copy of the grob object.

### Author(s)

Paul Murrell

### See Also

[grid.grob](#).

### Examples

---

grid.display.list *Control the Grid Display List*

---

### Description

Turn the Grid display list on or off.

### Usage

```
grid.display.list(on=TRUE)
engine.display.list(on=TRUE)
```

### Arguments

`on` A logical value to indicate whether the display list should be on or off.

### Details

All drawing and viewport-setting operations are (by default) recorded in the Grid display list. This allows redrawing to occur following an editing operation.

This display list could get very large so it may be useful to turn it off in some cases; this will of course disable redrawing.

All graphics output is also recorded on the main display list of the R graphics engine (by default). This supports redrawing following a device resize and allows copying between devices.

Turning off this display list means that grid will redraw from its own display list for device resizes and copies. This will be slower than using the graphics engine display list.

### Value

None.

### WARNING

Turning the display list on causes the display list to be erased!

Turning off both the grid display list and the graphics engine display list will result in no redrawing whatsoever.

### Author(s)

Paul Murrell

---

grid.draw	<i>Draw a grid grob</i>
-----------	-------------------------

---

**Description**

Produces graphical output from a graphical object.

**Usage**

```
grid.draw(x, recording=TRUE)
```

**Arguments**

<code>x</code>	An object of class "grob" or NULL.
<code>recording</code>	A logical value to indicate whether the drawing operation should be recorded on the Grid display list.

**Details**

This is a generic function with methods for grob and gTree objects.

The grob and gTree methods automatically push any viewports in a vp slot and automatically apply any gpar settings in a gp slot. In addition, the gTree method pushes and ups any viewports in a childrenvp slot and automatically calls `grid.draw` for any grobs in a children slot.

The methods for grob and gTree call the generic hook functions `preDrawDetails`, `drawDetails`, and `postDrawDetails` to allow classes derived from grob or gTree to perform additional viewport pushing/popping and produce additional output beyond the default behaviour for grobs and gTrees.

**Value**

None.

**Author(s)**

Paul Murrell

**See Also**

[grob](#).

**Examples**

```
grid.newpage()  
## Create a graphical object, but don't draw it  
l <- linesGrob()  
## Draw it  
grid.draw(l)
```

---

grid.edit

---

*Edit the Description of a Grid Graphical Object*


---

## Description

Changes the value of one of the slots of a grob and redraws the grob.

## Usage

```
grid.edit(gPath, ..., strict = FALSE, grep = FALSE, global = FALSE,
          allDevices = FALSE, redraw = TRUE)
editGrob(grob, gPath = NULL, ..., strict = FALSE, grep = FALSE,
         global = FALSE)
```

## Arguments

grob	A grob object.
...	Zero or more named arguments specifying new slot values.
gPath	A gPath object. For <code>grid.edit</code> this specifies a grob on the display list. For <code>editGrob</code> this specifies a descendant of the specified grob.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., <code>c(TRUE, FALSE)</code> means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
redraw	A logical value to indicate whether to redraw the grob.

## Details

`editGrob` copies the specified grob and returns a modified grob.

`grid.edit` destructively modifies a grob on the display list. If `redraw` is `TRUE` it then redraws everything to reflect the change.

Both functions call `editDetails` to allow a grob to perform custom actions and `validDetails` to check that the modified grob is still coherent.

## Value

`editGrob` returns a grob object; `grid.edit` returns `NULL`.

**Author(s)**

Paul Murrell

**See Also**[grob](#), [getGrob](#), [addGrob](#), [removeGrob](#).**Examples**

```

grid.newpage()
grid.xaxis(name = "xa", vp = viewport(width=.5, height=.5))
grid.edit("xa", gp = gpar(col="red"))
# won't work because no ticks (at is NULL)
try(grid.edit(gPath("xa", "ticks"), gp = gpar(col="green")))
grid.edit("xa", at = 1:4/5)
# Now it should work
try(grid.edit(gPath("xa", "ticks"), gp = gpar(col="green")))

```

grid.frame

*Create a Frame for Packing Objects***Description**

These functions, together with `grid.pack`, `grid.place`, `packGrob`, and `placeGrob` are part of a GUI-builder-like interface to constructing graphical images. The idea is that you create a frame with this function then use `grid.pack` or whatever to pack/place objects into the frame.

**Usage**

```

grid.frame(layout=NULL, name=NULL, gp=gpar(), vp=NULL, draw=TRUE)
frameGrob(layout=NULL, name=NULL, gp=gpar(), vp=NULL)

```

**Arguments**

layout	A Grid layout, or NULL. This can be used to initialise the frame with a number of rows and columns, with initial widths and heights, etc.
name	A character identifier.
vp	An object of class <code>viewport</code> , or NULL.
gp	An object of class <code>gpar</code> ; typically the output from a call to the function <code>gpar</code> .
draw	Should the frame be drawn.

**Details**

Both functions create a frame `grob` (a graphical object describing a frame), but only `grid.frame()` draws the frame (and then only if `draw` is `TRUE`). Nothing will actually be drawn, but it will put the frame on the display list, which means that the output will be dynamically updated as objects are packed into the frame. Possibly useful for debugging.

**Value**

A frame grob. `grid.frame()` returns the value invisibly.

**Author(s)**

Paul Murrell

**See Also**

[grid.pack](#)

**Examples**

```
grid.newpage()
grid.frame(name="gf", draw=TRUE)
grid.pack("gf", rectGrob(gp=gpar(fill="grey")))
grid.pack("gf", textGrob("hi there"), side="right")
```

---

grid.get

*Get a Grid Graphical Object*

---

**Description**

Retrieve a grob or a descendant of a grob.

**Usage**

```
grid.get(gPath, strict = FALSE, grep = FALSE, global = FALSE,
         allDevices = FALSE)

getGrob(gTree, gPath, strict = FALSE, grep = FALSE, global = FALSE)
```

**Arguments**

<code>gTree</code>	A <code>gTree</code> object.
<code>gPath</code>	A <code>gPath</code> object. For <code>grid.get</code> this specifies a grob on the display list. For <code>getGrob</code> this specifies a descendant of the specified <code>gTree</code> .
<code>strict</code>	A boolean indicating whether the <code>gPath</code> must be matched exactly.
<code>grep</code>	A boolean indicating whether the <code>gPath</code> should be treated as a regular expression. Values are recycled across elements of the <code>gPath</code> (e.g., <code>c(TRUE, FALSE)</code> means that every odd element of the <code>gPath</code> will be treated as a regular expression).
<code>global</code>	A boolean indicating whether the function should affect just the first match of the <code>gPath</code> , or whether all matches should be affected.
<code>allDevices</code>	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.

**Value**

A grob object.

**Author(s)**

Paul Murrell

**See Also**

[grob](#), [getGrob](#), [addGrob](#), [removeGrob](#).

**Examples**

```
grid.xaxis(name="xa")
grid.get("xa")
grid.get(gPath("xa", "ticks"))

grid.draw(gTree(name="gt", children=gList(xaxisGrob(name="axis"))))
grid.get(gPath("gt", "axis", "ticks"))
```

---

grid.grab

*Grab the current grid output*

---

**Description**

Creates a gTree object from the current grid display list.

**Usage**

```
grid.grab(...)
```

**Arguments**

... arguments passed to gTree, for example, a name and/or class for the gTree that is created.

**Details**

The gTree has a childrenvp slot containing all viewports on the display list (including those that are popped). Every grob on the display list is made a child of the new gTree, with a vpPath in the vp slot so that the grob is drawn in the appropriate viewport.

In other words, the gTree contains all elements on the display list, but in a slightly altered form.

The gTree should produce exactly the same output as the display list.

**Value**

A gTree object.

**See Also**[gTree](#)**Examples**

```
pushViewport(viewport(w=.5, h=.5))
grid.rect()
grid.points(runif(10), runif(10))
popViewport()
grab <- grid.grab()
grid.newpage()
grid.draw(grab)
```

---

`grid.grill`*Draw a Grill*

---

**Description**

This function draws a grill within a Grid viewport.

**Usage**

```
grid.grill(h = unit(seq(0.25, 0.75, 0.25), "npc"),
           v = unit(seq(0.25, 0.75, 0.25), "npc"),
           default.units = "npc", gp=gpar(col = "grey"), vp = NULL)
```

**Arguments**

<code>h</code>	A numeric vector or unit object indicating the horizontal location of the vertical grill lines.
<code>v</code>	A numeric vector or unit object indicating the vertical location of the horizontal grill lines.
<code>default.units</code>	A string indicating the default units to use if <code>h</code> or <code>v</code> are only given as numeric vectors.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>vp</code>	A Grid viewport object.

**Value**

None.

**Author(s)**

Paul Murrell

**See Also**

[Grid](#), [viewport](#).

---

grid.grob

*Create a Grid Graphical Object*

---

**Description**

These functions create grid graphical objects.

**Usage**

```
grid.grob(list.struct, cl = NULL, draw = TRUE)
grob(..., name = NULL, gp = NULL, vp = NULL, cl = NULL)
gTree(..., name = NULL, gp = NULL, vp = NULL, children = NULL,
       childrenvp = NULL, cl = NULL)
childNames(gTree)
gList(...)
```

**Arguments**

<code>...</code>	For <code>grob</code> and <code>gTree</code> , the named slots describing important features of the graphical object. For <code>gList</code> , a series of <code>grob</code> objects.
<code>list.struct</code>	A list (preferably with each element named).
<code>name</code>	A character identifier for the <code>grob</code> . Used to find the <code>grob</code> on the display list and/or as a child of another <code>grob</code> .
<code>children</code>	A <code>gList</code> object.
<code>childrenvp</code>	A viewport object (or <code>NULL</code> ).
<code>gp</code>	A <code>gpar</code> object, typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>vp</code>	A viewport object (or <code>NULL</code> ).
<code>cl</code>	A string giving the class attribute for the <code>list.struct</code>
<code>draw</code>	A logical value to indicate whether to produce graphical output.
<code>gTree</code>	A <code>gTree</code> object.

**Details**

These functions can be used to create a basic `grob`, `gTree`, or `gList` object, or a new class derived from one of these.

A grid graphical object (`grob`) is a description of a graphical item. These basic classes provide default behaviour for validating, drawing, and modifying graphical objects. Both call the function `validDetails` to check that the object returned is coherent.

A gTree can have other grobs as children; when a gTree is drawn, it draws all of its children. Before drawing its children, a gTree pushes its childrenvp slot and then navigates back up (calls upViewport) so that the children can specify their location within the childrenvp via a vpPath.

Grob names need not be unique in general, but all children of a gTree must have different names. A grob name can be any valid R symbol name. Also, it is not advisable to use the gPath separator (currently : :) in grob names.

The function childNames returns the names of the grobs which are children of a gTree.

All grid primitives (grid.lines, grid.rect, ...) and some higher-level grid components (e.g., grid.xaxis and grid.yaxis) are derived from these classes.

grid.grob is deprecated.

### Value

A grob object.

### Author(s)

Paul Murrell

### See Also

[grid.draw](#), [grid.edit](#), [grid.get](#).

### Examples

---

grid.layout                      *Create a Grid Layout*

---

### Description

This function returns a Grid layout, which describes a subdivision of a rectangular region.

### Usage

```
grid.layout(nrow = 1, ncol = 1,
            widths = unit(rep(1, ncol), "null"),
            heights = unit(rep(1, nrow), "null"),
            default.units = "null", respect = FALSE,
            just="centre")
```

**Arguments**

nrow	An integer describing the number of rows in the layout.
ncol	An integer describing the number of columns in the layout.
widths	A numeric vector or unit object describing the widths of the columns in the layout.
heights	A numeric vector or unit object describing the heights of the rows in the layout.
default.units	A string indicating the default units to use if widths or heights are only given as numeric vectors.
respect	A logical value or a numeric matrix. If a logical, this indicates whether row heights and column widths should respect each other. If a matrix, non-zero values indicate that the corresponding row and column should be respected (see examples below).
just	A string vector indicating how the layout should be justified if it is not the same size as its parent viewport. If there are two values, the first value specifies horizontal justification and the second value specifies vertical justification. Possible values are: "left", "right", "centre", "center", "bottom", and "top". NOTE that in this context, "left", for example, means align the left edge of the left-most layout column with the left edge of the parent viewport.

**Details**

The unit objects given for the widths and heights of a layout may use a special units that only has meaning for layouts. This is the "null" unit, which indicates what relative fraction of the available width/height the column/row occupies. See the reference for a better description of relative widths and heights in layouts.

**Value**

A Grid layout object.

**WARNING**

This function must NOT be confused with the base R graphics function layout. In particular, do not use layout in combination with Grid graphics. The documentation for layout may provide some useful information and this function should behave identically in comparable situations. The grid.layout function has added the ability to specify a broader range of units for row heights and column widths, and allows for nested layouts (see viewport).

**Author(s)**

Paul Murrell

**References**

Murrell, P. R. (1999), Layouts: A Mechanism for Arranging Plots on a Page, *Journal of Computational and Graphical Statistics*, **8**, 121–134.

**See Also**

[Grid](#), [grid.show.layout](#), [viewport](#), [layout](#)

**Examples**

```
## A variety of layouts (some a bit mid-bending ...)
layout.torture()
## Demonstration of layout justification
grid.newpage()
testlay <- function(just="centre") {
  pushViewport(viewport(layout=grid.layout(1, 1, widths=unit(1, "inches"),
    height=unit(0.25, "npc"),
    just=just)))
  pushViewport(viewport(layout.pos.col=1, layout.pos.row=1))
  grid.rect()
  grid.text(paste(just, collapse="-"))
  popViewport(2)
}
testlay()
testlay(c("left", "top"))
testlay(c("right", "top"))
testlay(c("right", "bottom"))
testlay(c("left", "bottom"))
testlay(c("left"))
testlay(c("right"))
testlay(c("bottom"))
testlay(c("top"))
```

---

grid.lines

*Draw Lines in a Grid Viewport*

---

**Description**

These functions create and draw a series of lines.

**Usage**

```
grid.lines(x = unit(c(0, 1), "npc", units.per.obs),
  y = unit(c(0, 1), "npc", units.per.obs),
  default.units = "npc", units.per.obs = FALSE, name = NULL,
  gp=gpar(), draw = TRUE, vp = NULL)
linesGrob(x = unit(c(0, 1), "npc", units.per.obs),
  y = unit(c(0, 1), "npc", units.per.obs),
  default.units = "npc", units.per.obs = FALSE, name = NULL,
  gp=gpar(), vp = NULL)
```

**Arguments**

<code>x</code>	A numeric vector or unit object specifying x-values.
<code>y</code>	A numeric vector or unit object specifying y-values.
<code>default.units</code>	A string indicating the default units to use if <code>x</code> or <code>y</code> are only given as numeric vectors.
<code>units.per.obs</code>	A logical value to indicate whether each individual (x, y) location has its own unit(s) specified.
<code>name</code>	A character identifier.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>vp</code>	A Grid viewport object (or NULL).

**Details**

Both functions create a lines `grob` (a graphical object describing lines), but only `grid.lines` draws the lines (and then only if `draw` is `TRUE`).

**Value**

A lines `grob`. `grid.lines` returns the value invisibly.

**Author(s)**

Paul Murrell

**See Also**

[Grid, viewport](#)

---

`grid.locator`

*Capture a Mouse Click*

---

**Description**

Allows the user to click the mouse once within the current graphics device and returns the location of the mouse click within the current viewport, in the specified coordinate system.

**Usage**

```
grid.locator(unit = "native")
```

**Arguments**

`unit`                   The coordinate system in which to return the location of the mouse click. See the [unit](#) function for valid coordinate systems.

**Details**

This function is modal (like the base function `locator`) so the command line and graphics drawing is blocked until the use has clicked the mouse in the current device.

**Value**

A unit object representing the location of the mouse click within the current viewport, in the specified coordinate system.

If the user did not click mouse button 1, the function (invisibly) returns `NULL`.

**Author(s)**

Paul Murrell

**See Also**

[viewport](#), [unit](#), [locator](#)

**Examples**

```
if (interactive()) {
  ## Need to write a more sophisticated unit as.character method
  unittrim <- function(unit) {
    sub("^[0-9]+|[0-9]+[.][0-9]*", "\\1", as.character(unit))
  }
  do.click <- function(unit) {
    click.locn <- grid.locator(unit)
    grid.segments(unit.c(click.locn$x, unit(0, "npc")),
                 unit.c(unit(0, "npc"), click.locn$y),
                 click.locn$x, click.locn$y,
                 gp=gpar(lty="dashed", col="grey"))
    grid.points(click.locn$x, click.locn$y, pch=16, size=unit(1, "mm"))
    clickx <- unittrim(click.locn$x)
    clicky <- unittrim(click.locn$y)
    grid.text(paste("(", clickx, ", ", clicky, ")", sep=""),
             click.locn$x + unit(2, "mm"), click.locn$y,
             just="left")
  }
  do.click("inches")
  pushViewport(viewport(width=0.5, height=0.5,
                       xscale=c(0, 100), yscale=c(0, 10)))

  grid.rect()
  grid.xaxis()
  grid.yaxis()
  do.click("native")
  popViewport()
}
```

```
}

```

---

grid.move.to	<i>Move or Draw to a Specified Position</i>
--------------	---

---

### Description

Grid has the notion of a current location. These functions sets that location.

### Usage

```
grid.move.to(x = 0, y = 0, default.units = "npc", name = NULL,
            draw = TRUE, vp = NULL)

moveToGrob(x = 0, y = 0, default.units = "npc", name = NULL, vp = NULL)

grid.line.to(x = 1, y = 1, default.units = "npc", name = NULL,
            gp = gpar(), draw = TRUE, vp = NULL)

lineToGrob(x = 1, y = 1, default.units = "npc", name = NULL,
            gp = gpar(), vp = NULL)
```

### Arguments

<code>x</code>	A numeric value or a unit object specifying an x-value.
<code>y</code>	A numeric value or a unit object specifying a y-value.
<code>default.units</code>	A string indicating the default units to use if <code>x</code> or <code>y</code> are only given as numeric values.
<code>name</code>	A character identifier.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>vp</code>	A Grid viewport object (or <code>NULL</code> ).

### Details

Both functions create a `move.to/line.to` grob (a graphical object describing a `move-to/line-to`), but only `grid.move.to/line.to()` draws the `move.to/line.to` (and then only if `draw` is `TRUE`).

### Value

A `move.to/line.to` grob. `grid.move.to/line.to()` returns the value invisibly.

### Author(s)

Paul Murrell

**See Also**

[Grid](#), [viewport](#)

**Examples**

```
grid.newpage()  
grid.move.to(0.5, 0.5)  
grid.line.to(1, 1)  
grid.line.to(0.5, 0)  
pushViewport(viewport(x=0, y=0, w=0.25, h=0.25, just=c("left", "bottom")))  
grid.rect()  
grid.grill()  
grid.line.to(0.5, 0.5)  
popViewport()
```

---

grid.newpage

*Move to a New Page on a Grid Device*

---

**Description**

This function erases the current device or moves to a new page.

**Usage**

```
grid.newpage(recording = TRUE)
```

**Arguments**

`recording` A logical value to indicate whether the new-page operation should be saved onto the Grid display list.

**Details**

There is a hook called "grid.newpage" (see [setHook](#)) which is used in the testing code to annotate the new page. The hook function(s) are called with no argument. (If the value is a character string, `get` is called on it from within the **grid** namespace.)

**Value**

None.

**Author(s)**

Paul Murrell

**See Also**

[Grid](#)

grid.pack

*Pack an Object within a Frame***Description**

This functions, together with `grid.frame` and `frameGrob` are part of a GUI-builder-like interface to constructing graphical images. The idea is that you create a frame with `grid.frame` or `frameGrob` then use this functions to pack objects into the frame.

**Usage**

```
grid.pack(gPath, grob, redraw = TRUE, side = NULL,
         row = NULL, row.before = NULL, row.after = NULL,
         col = NULL, col.before = NULL, col.after = NULL,
         width = NULL, height = NULL,
         force.width = FALSE, force.height = FALSE, border = NULL,
         dynamic = FALSE)
```

```
packGrob(frame, grob, side = NULL,
         row = NULL, row.before = NULL, row.after = NULL,
         col = NULL, col.before = NULL, col.after = NULL,
         width = NULL, height = NULL,
         force.width = FALSE, force.height = FALSE, border = NULL,
         dynamic = FALSE)
```

**Arguments**

<code>gPath</code>	A <code>gPath</code> object, which specifies a frame on the display list.
<code>frame</code>	An object of class <code>frame</code> , typically the output from a call to <code>grid.frame</code> .
<code>grob</code>	An object of class <code>grob</code> . The object to be packed.
<code>redraw</code>	A boolean indicating whether the output should be updated.
<code>side</code>	One of "left", "top", "right", "bottom" to indicate which side to pack the object on.
<code>row</code>	Which row to add the object to. Must be between 1 and the-number-of-rows-currently-in-the-frame + 1, or <code>NULL</code> in which case the object occupies all rows.
<code>row.before</code>	Add the object to a new row just before this row.
<code>row.after</code>	Add the object to a new row just after this row.
<code>col</code>	Which col to add the object to. Must be between 1 and the-number-of-cols-currently-in-the-frame + 1, or <code>NULL</code> in which case the object occupies all cols.
<code>col.before</code>	Add the object to a new col just before this col.
<code>col.after</code>	Add the object to a new col just after this col.
<code>width</code>	Specifies the width of the column that the object is added to (rather than allowing the width to be taken from the object).

<code>height</code>	Specifies the height of the row that the object is added to (rather than allowing the height to be taken from the object).
<code>force.width</code>	A logical value indicating whether the width of the column that the grob is being packed into should be EITHER the width specified in the call to <code>grid.pack</code> OR the maximum of that width and the pre-existing width.
<code>force.height</code>	A logical value indicating whether the height of the column that the grob is being packed into should be EITHER the height specified in the call to <code>grid.pack</code> OR the maximum of that height and the pre-existing height.
<code>border</code>	A <code>unit</code> object of length 4 indicating the borders around the object.
<code>dynamic</code>	If the width/height is taken from the grob being packed, this boolean flag indicates whether the <code>grobwidth/height</code> unit refers directly to the grob, or uses a <code>gPath</code> to the grob. In the latter case, changes to the grob will trigger a recalculation of the width/height.

### Details

`packGrob` modifies the given frame grob and returns the modified frame grob.

`grid.pack` destructively modifies a frame grob on the display list (and redraws the display list if `redraw` is `TRUE`).

These are (meant to be) very flexible functions. There are many different ways to specify where the new object is to be added relative to the objects already in the frame. The function checks that the specification is not self-contradictory.

NOTE that the width/height of the row/col that the object is added to is taken from the object itself unless the `width/height` is specified.

### Value

`packGrob` returns a frame grob, but `grid.pack` returns `NULL`.

### Author(s)

Paul Murrell

### See Also

[grid.frame](#), [grid.place](#), [grid.edit](#), and [gPath](#).

---

`grid.place`

*Place an Object within a Frame*

---

### Description

These functions provide a simpler (and faster) alternative to the `grid.pack()` and `packGrob` functions. They can be used to place objects within the existing rows and columns of a frame layout. They do not provide the ability to add new rows and columns nor do they affect the heights and widths of the rows and columns.

**Usage**

```
grid.place(gPath, grob, row = 1, col = 1, redraw = TRUE)
placeGrob(frame, grob, row = NULL, col = NULL)
```

**Arguments**

<code>gPath</code>	A <code>gPath</code> object, which specifies a frame on the display list.
<code>frame</code>	An object of class <code>frame</code> , typically the output from a call to <code>grid.frame</code> .
<code>grob</code>	An object of class <code>grob</code> . The object to be placed.
<code>row</code>	Which row to add the object to. Must be between 1 and the-number-of-rows-currently-in-the-frame.
<code>col</code>	Which col to add the object to. Must be between 1 and the-number-of-cols-currently-in-the-frame.
<code>redraw</code>	A boolean indicating whether the output should be updated.

**Details**

`placeGrob` modifies the given frame `grob` and returns the modified frame `grob`.

`grid.place` destructively modifies a frame `grob` on the display list (and redraws the display list if `redraw` is `TRUE`).

**Value**

`placeGrob` returns a frame `grob`, but `grid.place` returns `NULL`.

**Author(s)**

Paul Murrell

**See Also**

[grid.frame](#), [grid.pack](#), [grid.edit](#), and [gPath](#).

---

`grid.plot.and.legend`

*A Simple Plot and Legend Demo*

---

**Description**

This function is just a wrapper for a simple demonstration of how a basic plot and legend can be drawn from scratch using `grid`.

**Usage**

```
grid.plot.and.legend()
```

**Author(s)**

Paul Murrell

**Examples**

```
grid.plot.and.legend()
```

---

```
grid.points
```

---

```
Draw Data Symbols
```

---

**Description**

These functions create and draw data symbols.

**Usage**

```
grid.points(x = runif(10),
            y = runif(10),
            pch = 1, size = unit(1, "char"),
            default.units = "native", name = NULL,
            gp=gpar(), draw = TRUE, vp = NULL)
pointsGrob(x = runif(10),
           y = runif(10),
           pch = 1, size = unit(1, "char"),
           default.units = "native", name = NULL,
           gp=gpar(), vp = NULL)
```

**Arguments**

<code>x</code>	A numeric vector or unit object specifying x-values.
<code>y</code>	A numeric vector or unit object specifying y-values.
<code>pch</code>	A numeric or character vector indicating what sort of plotting symbol to use.
<code>size</code>	A unit object specifying the size of the plotting symbols.
<code>default.units</code>	A string indicating the default units to use if x or y are only given as numeric vectors.
<code>name</code>	A character identifier.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>vp</code>	A Grid viewport object (or <code>NULL</code> ).

**Details**

Both functions create a points grob (a graphical object describing points), but only `grid.points` draws the points (and then only if `draw` is `TRUE`).

**Value**

A points grob. `grid.points` returns the value invisibly.

**Author(s)**

Paul Murrell

**See Also**

[Grid](#), [viewport](#)

---

<code>grid.polygon</code>	<i>Draw a Polygon</i>
---------------------------	-----------------------

---

**Description**

These functions create and draw a polygon. The final point will automatically be connected to the initial point.

**Usage**

```
grid.polygon(x=c(0, 0.5, 1, 0.5), y=c(0.5, 1, 0.5, 0),
             id=NULL, id.lengths=NULL,
             default.units="npc", name=NULL,
             gp=gpar(), draw=TRUE, vp=NULL)
polygonGrob(x=c(0, 0.5, 1, 0.5), y=c(0.5, 1, 0.5, 0),
            id=NULL, id.lengths=NULL,
            default.units="npc", name=NULL,
            gp=gpar(), vp=NULL)
```

**Arguments**

<code>x</code>	A numeric vector or unit object specifying x-locations.
<code>y</code>	A numeric vector or unit object specifying y-locations.
<code>id</code>	A numeric vector used to separate locations in <code>x</code> and <code>y</code> into multiple polygons. All locations with the same <code>id</code> belong to the same polygon.
<code>id.lengths</code>	A numeric vector used to separate locations in <code>x</code> and <code>y</code> into multiple polygons. Specifies consecutive blocks of locations which make up separate polygons.
<code>default.units</code>	A string indicating the default units to use if <code>x</code> , <code>y</code> , <code>width</code> , or <code>height</code> are only given as numeric vectors.
<code>name</code>	A character identifier.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>vp</code>	A Grid viewport object (or <code>NULL</code> ).

**Details**

Both functions create a polygon grob (a graphical object describing a polygon), but only `grid.polygon` draws the polygon (and then only if `draw` is `TRUE`).

**Value**

A grob object.

**Author(s)**

Paul Murrell

**See Also**

[Grid](#), [viewport](#)

**Examples**

```
grid.polygon()
# Using id (NOTE: locations are not in consecutive blocks)
grid.newpage()
grid.polygon(x=c((0:4)/10, rep(.5, 5), (10:6)/10, rep(.5, 5)),
             y=c(rep(.5, 5), (10:6/10), rep(.5, 5), (0:4)/10),
             id=rep(1:5, 4),
             gp=gpar(fill=1:5))
# Using id.lengths
grid.newpage()
grid.polygon(x=outer(c(0, .5, 1, .5), 5:1/5),
             y=outer(c(.5, 1, .5, 0), 5:1/5),
             id.lengths=rep(4, 5),
             gp=gpar(fill=1:5))
```

---

```
grid.pretty
```

*Generate a Sensible Set of Breakpoints*

---

**Description**

Produces a pretty set of breakpoints within the range given.

**Usage**

```
grid.pretty(range)
```

**Arguments**

`range`            A numeric vector

**Value**

A numeric vector of breakpoints.

**Author(s)**

Paul Murrell

---

grid.prompt	<i>Prompt before new page</i>
-------------	-------------------------------

---

**Description**

This function can be used to control whether the user is prompted before starting a new page of output.

**Usage**

```
grid.prompt(ask)
```

**Arguments**

ask	a logical value. If TRUE, the user is prompted before a new page of output is started.
-----	--

**Value**

The current prompt setting *before* any new setting is applied.

**Author(s)**

Paul Murrell

**See Also**

[grid.newpage](#)

---

grid.rect	<i>Draw rectangles</i>
-----------	------------------------

---

### Description

These functions create and draw rectangles.

### Usage

```
grid.rect(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          width = unit(1, "npc"), height = unit(1, "npc"),
          just = "centre", default.units = "npc", name = NULL,
          gp=gpar(), draw = TRUE, vp = NULL)
rectGrob(x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          width = unit(1, "npc"), height = unit(1, "npc"),
          just = "centre", default.units = "npc", name = NULL,
          gp=gpar(), vp = NULL)
```

### Arguments

<code>x</code>	A numeric vector or unit object specifying x-location.
<code>y</code>	A numeric vector or unit object specifying y-location.
<code>width</code>	A numeric vector or unit object specifying width.
<code>height</code>	A numeric vector or unit object specifying height.
<code>just</code>	The justification of the rectangle about its (x, y) location. If two values are given, the first specifies horizontal justification and the second specifies vertical justification.
<code>default.units</code>	A string indicating the default units to use if x, y, width, or height are only given as numeric vectors.
<code>name</code>	A character identifier.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>vp</code>	A Grid viewport object (or <code>NULL</code> ).

### Details

Both functions create a `rect grob` (a graphical object describing rectangles), but only `grid.rect` draws the rectangles (and then only if `draw` is `TRUE`).

### Value

A `rect grob`. `grid.rect` returns the value invisibly.

**Author(s)**

Paul Murrell

**See Also**[Grid](#), [viewport](#)


---

grid.remove	<i>Remove a Grid Graphical Object</i>
-------------	---------------------------------------

---

**Description**

Remove a grob from a gTree or a descendant of a gTree.

**Usage**

```
grid.remove(gPath, warn = TRUE, strict = FALSE, grep = FALSE,
            global = FALSE, allDevices = FALSE, redraw = TRUE)

removeGrob(gTree, gPath, strict = FALSE, grep = FALSE, global = FALSE,
           warn = TRUE)
```

**Arguments**

gTree	A gTree object.
gPath	A gPath object. For <code>grid.remove</code> this specifies a gTree on the display list. For <code>removeGrob</code> this specifies a descendant of the specified gTree.
strict	A boolean indicating whether the gPath must be matched exactly.
grep	A boolean indicating whether the gPath should be treated as a regular expression. Values are recycled across elements of the gPath (e.g., <code>c(TRUE, FALSE)</code> means that every odd element of the gPath will be treated as a regular expression).
global	A boolean indicating whether the function should affect just the first match of the gPath, or whether all matches should be affected.
allDevices	A boolean indicating whether all open devices should be searched for matches, or just the current device. NOT YET IMPLEMENTED.
warn	A logical to indicate whether failing to find the specified grob should trigger an error.
redraw	A logical value to indicate whether to redraw the grob.

**Details**

`removeGrob` copies the specified grob and returns a modified grob.

`grid.remove` destructively modifies a grob on the display list. If `redraw` is `TRUE` it then redraws everything to reflect the change.

**Value**

removeGrob returns a grob object; grid.remove returns NULL.

**Author(s)**

Paul Murrell

**See Also**

[grob](#), [getGrob](#), [removeGrob](#), [removeGrob](#).

**Examples**


---

```
grid.segments          Draw Line Segments
```

---

**Description**

These functions create and draw line segments.

**Usage**

```
grid.segments(x0 = unit(0, "npc"), y0 = unit(0, "npc"),
              x1 = unit(1, "npc"), y1 = unit(1, "npc"),
              default.units = "npc", units.per.obs = FALSE,
              name = NULL, gp = gpar(), draw = TRUE, vp = NULL)
segmentsGrob(x0 = unit(0, "npc"), y0 = unit(0, "npc"),
             x1 = unit(1, "npc"), y1 = unit(1, "npc"),
             default.units = "npc", units.per.obs = FALSE,
             name = NULL, gp = gpar(), vp = NULL)
```

**Arguments**

x0	Numeric indicating the starting x-values of the line segments.
y0	Numeric indicating the starting y-values of the line segments.
x1	Numeric indicating the stopping x-values of the line segments.
y1	Numeric indicating the stopping y-values of the line segments.
default.units	A string.
units.per.obs	A boolean indicating whether distinct units are given for each x/y-value.
name	A character identifier.
gp	An object of class gpar.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or NULL)

**Details**

Both functions create a segments grob (a graphical object describing segments), but only `grid.segments` draws the segments (and then only if `draw` is `TRUE`).

**Value**

A segments grob. `grid.segments` returns the value invisibly.

**Author(s)**

Paul Murrell

**See Also**

[Grid](#), [viewport](#)

---

`grid.set`

*Set a Grid Graphical Object*

---

**Description**

Replace a grob or a descendant of a grob.

**Usage**

```
grid.set(gPath, newGrob, strict = FALSE, grep = FALSE, redraw = TRUE)
setGrob(gTree, gPath, newGrob, strict = FALSE, grep = FALSE)
```

**Arguments**

<code>gTree</code>	A <code>gTree</code> object.
<code>gPath</code>	A <code>gPath</code> object. For <code>grid.set</code> this specifies a grob on the display list. For <code>setGrob</code> this specifies a descendant of the specified <code>gTree</code> .
<code>newGrob</code>	A grob object.
<code>strict</code>	A boolean indicating whether the <code>gPath</code> must be matched exactly.
<code>grep</code>	A boolean indicating whether the <code>gPath</code> should be treated as a regular expression. Values are recycled across elements of the <code>gPath</code> (e.g., <code>c(TRUE, FALSE)</code> means that every odd element of the <code>gPath</code> will be treated as a regular expression).
<code>redraw</code>	A logical value to indicate whether to redraw the grob.

**Details**

`setGrob` copies the specified grob and returns a modified grob.

`grid.set` destructively replaces a grob on the display list. If `redraw` is `TRUE` it then redraws everything to reflect the change.

These functions should not normally be called by the user.

**Value**

setGrob returns a grob object; grid.set returns NULL.

**Author(s)**

Paul Murrell

**See Also**

[grid.grob](#).

---

grid.show.layout     *Draw a Diagram of a Grid Layout*

---

**Description**

This function uses Grid graphics to draw a diagram of a Grid layout.

**Usage**

```
grid.show.layout(l, newpage=TRUE, bg = "light grey",
  cell.border = "blue", cell.fill = "light blue",
  cell.label = TRUE, label.col = "blue",
  unit.col = "red", vp = NULL)
```

**Arguments**

l	A Grid layout object.
newpage	A logical value indicating whether to move on to a new page before drawing the diagram.
bg	The colour used for the background.
cell.border	The colour used to draw the borders of the cells in the layout.
cell.fill	The colour used to fill the cells in the layout.
cell.label	A logical indicating whether the layout cells should be labelled.
label.col	The colour used for layout cell labels.
unit.col	The colour used for labelling the widths/heights of columns/rows.
vp	A Grid viewport object (or NULL).

**Details**

A viewport is created within `vp` to provide a margin for annotation, and the layout is drawn within that new viewport. The margin is filled with light grey, the new viewport is filled with white and framed with a black border, and the layout regions are filled with light blue and framed with a blue border. The diagram is annotated with the widths and heights (including units) of the columns and rows of the layout using red text. (All colours are defaults and may be customised via function arguments.)

**Value**

None.

**Author(s)**

Paul Murrell

**See Also**[Grid](#), [viewport](#), [grid.layout](#)**Examples**

```
## Diagram of a simple layout
grid.show.layout(grid.layout(4,2,
    heights=unit(rep(1, 4),
        c("lines", "lines", "lines", "null")),
    widths=unit(c(1, 1), "inches")))
```

---

`grid.show.viewport` *Draw a Diagram of a Grid Viewport*

---

**Description**

This function uses Grid graphics to draw a diagram of a Grid viewport.

**Usage**

```
grid.show.viewport(v, parent.layout = NULL, newpage = TRUE,
    border.fill="light grey",
    vp.col="blue", vp.fill="light blue",
    scale.col="red",
    vp = NULL)
```

**Arguments**

<code>v</code>	A Grid viewport object.
<code>parent.layout</code>	A grid layout object. If this is not NULL and the viewport given in <code>v</code> has its location specified relative to the layout, then the diagram shows the layout and which cells <code>v</code> occupies within the layout.
<code>newpage</code>	A logical value to indicate whether to move to a new page before drawing the diagram.
<code>border.fill</code>	Colour to fill the border margin.
<code>vp.col</code>	Colour for the border of the viewport region.
<code>vp.fill</code>	Colour to fill the viewport region.
<code>scale.col</code>	Colour to draw the viewport axes.
<code>vp</code>	A Grid viewport object (or NULL).

**Details**

A viewport is created within `vp` to provide a margin for annotation, and the diagram is drawn within that new viewport. By default, the margin is filled with light grey, the new viewport is filled with white and framed with a black border, and the viewport region is filled with light blue and framed with a blue border. The diagram is annotated with the width and height (including units) of the viewport, the (x, y) location of the viewport, and the x- and y-scales of the viewport, using red lines and text.

**Value**

None.

**Author(s)**

Paul Murrell

**See Also**

[Grid](#), [viewport](#)

**Examples**

```
## Diagram of a sample viewport
grid.show.viewport(viewport(x=0.6, y=0.6,
                             w=unit(1, "inches"), h=unit(1, "inches")))
grid.show.viewport(viewport(layout.pos.row=2, layout.pos.col=2:3),
                    grid.layout(3, 4))
```

---

grid.text

*Draw Text*

---

**Description**

These functions create and draw text.

**Usage**

```
grid.text(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          just = "centre", rot = 0,
          check.overlap = FALSE, default.units = "npc",
          name = NULL, gp = gpar(), draw = TRUE, vp = NULL)

textGrob(label, x = unit(0.5, "npc"), y = unit(0.5, "npc"),
          just = "centre", rot = 0,
          check.overlap = FALSE, default.units = "npc",
          name = NULL, gp = gpar(), vp = NULL)
```

**Arguments**

<code>label</code>	A vector of strings or expressions to draw.
<code>x</code>	A numeric vector or unit object specifying x-values.
<code>y</code>	A numeric vector or unit object specifying y-values.
<code>just</code>	The justification of the text about its (x, y) location. If two values are given, the first specifies horizontal justification and the second specifies vertical justification.
<code>rot</code>	The angle to rotate the text.
<code>check.overlap</code>	A logical value to indicate whether to check for and omit overlapping text.
<code>default.units</code>	A string indicating the default units to use if <code>x</code> or <code>y</code> are only given as numeric vectors.
<code>name</code>	A character identifier.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>vp</code>	A Grid viewport object (or NULL).

**Details**

Both functions create a text grob (a graphical object describing text), but only `grid.text` draws the text (and then only if `draw` is TRUE).

If the `label` argument is an expression, the output is formatted as a mathematical annotation, as for base graphics text.

**Value**

A text grob. `grid.text` returns the value invisibly.

**Author(s)**

Paul Murrell

**See Also**

[Grid, viewport](#)

**Examples**

```
grid.newpage()
x <- stats::runif(20)
y <- stats::runif(20)
rot <- stats::runif(20, 0, 360)
grid.text("SOMETHING NICE AND BIG", x=x, y=y, rot=rot,
         gp=gpar(fontsize=20, col="grey"))
```

```

grid.text("SOMETHING NICE AND BIG", x=x, y=y, rot=rot,
         gp=gpar(fontsize=20), check=TRUE)
grid.newpage()
draw.text <- function(just, i, j) {
  grid.text("ABCD", x=x[j], y=y[i], just=just)
  grid.text(deparse(substitute(just)), x=x[j], y=y[i] + unit(2, "lines"),
           gp=gpar(col="grey", fontsize=8))
}
x <- unit(1:4/5, "npc")
y <- unit(1:4/5, "npc")
grid.grill(h=y, v=x, gp=gpar(col="grey"))
draw.text(c("bottom"), 1, 1)
draw.text(c("left", "bottom"), 2, 1)
draw.text(c("right", "bottom"), 3, 1)
draw.text(c("centre", "bottom"), 4, 1)
draw.text(c("centre"), 1, 2)
draw.text(c("left", "centre"), 2, 2)
draw.text(c("right", "centre"), 3, 2)
draw.text(c("centre", "centre"), 4, 2)
draw.text(c("top"), 1, 3)
draw.text(c("left", "top"), 2, 3)
draw.text(c("right", "top"), 3, 3)
draw.text(c("centre", "top"), 4, 3)
draw.text(c(), 1, 4)
draw.text(c("left"), 2, 4)
draw.text(c("right"), 3, 4)
draw.text(c("centre"), 4, 4)

```

---

grid.xaxis

*Draw an X-Axis*


---

### Description

These functions create and draw an x-axis.

### Usage

```

grid.xaxis(at = NULL, label = TRUE, main = TRUE, name = NULL,
          gp = gpar(), draw = TRUE, vp = NULL)

```

```

xaxisGrob(at = NULL, label = TRUE, main = TRUE, name = NULL,
          gp = gpar(), vp = NULL)

```

### Arguments

at	A numeric vector of x-value locations for the tick marks.
label	A logical value indicating whether to draw the labels on the tick marks.
main	A logical value indicating whether to draw the axis at the bottom (TRUE) or at the top (FALSE) of the viewport.

name	A character identifier.
gp	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
draw	A logical value indicating whether graphics output should be produced.
vp	A Grid viewport object (or <code>NULL</code> ).

### Details

Both functions create an `xaxis grob` (a graphical object describing an axis), but only `grid.xaxis` draws the axis (and then only if `draw` is `TRUE`).

### Value

An `xaxis grob`. `grid.xaxis` returns the value invisibly.

### Children

If the `at` slot of an `xaxis grob` is not `NULL` then the axis will have the following children:

**major** representing the line at the base of the tick marks.

**ticks** representing the tick marks.

**labels** representing the tick labels.

If the `at` slot is `NULL` then there are no children and ticks are drawn based on the current viewport scale.

### Author(s)

Paul Murrell

### See Also

[Grid](#), [viewport](#), [grid.yaxis](#)

---

`grid.yaxis`

*Draw a Y-Axis*

---

### Description

These functions create and draw a y-axis.

### Usage

```
grid.yaxis(at = NULL, label = TRUE, main = TRUE, name = NULL,
           gp = gpar(), draw = TRUE, vp = NULL)
```

```
yaxisGrob(at = NULL, label = TRUE, main = TRUE, name = NULL,
           gp = gpar(), vp = NULL)
```

**Arguments**

<code>at</code>	A numeric vector of y-value locations for the tick marks.
<code>label</code>	A logical value indicating whether to draw the labels on the tick marks.
<code>main</code>	A logical value indicating whether to draw the axis at the left (TRUE) or at the right (FALSE) of the viewport.
<code>name</code>	A character identifier.
<code>gp</code>	An object of class <code>gpar</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	A logical value indicating whether graphics output should be produced.
<code>vp</code>	A Grid viewport object (or NULL).

**Details**

Both functions create a `yaxis grob` (a graphical object describing a yaxis), but only `grid.yaxis` draws the yaxis (and then only if `draw` is TRUE).

**Value**

A `yaxis grob`. `grid.yaxis` returns the value invisibly.

**Children**

If the `at` slot of an `xaxis grob` is not NULL then the `xaxis` will have the following children:

**major** representing the line at the base of the tick marks.

**ticks** representing the tick marks.

**labels** representing the tick labels.

If the `at` slot is NULL then there are no children and ticks are drawn based on the current viewport scale.

**Author(s)**

Paul Murrell

**See Also**

[Grid](#), [viewport](#), [grid.xaxis](#)

---

`grobWidth`                      *Create a Unit Describing the Width of a Grob*

---

**Description**

These functions create a unit object describing the width or height of a grob. They are generic.

**Usage**

```
grobWidth(x)
grobHeight(x)
```

**Arguments**

`x`                      A grob object.

**Value**

A unit object.

**Author(s)**

Paul Murrell

**See Also**

[unit](#) and [stringWidth](#)

---

`plotViewport`                      *Create a Viewport with a Standard Plot Layout*

---

**Description**

This is a convenience function for producing a viewport with the common S-style plot layout – i.e., a central plot region surrounded by margins given in terms of a number of lines of text.

**Usage**

```
plotViewport(margins, ...)
```

**Arguments**

`margins`                      A numeric vector interpreted in the same way as `par(mar)` in base graphics.  
`...`                      All other arguments will be passed to a call to the `viewport()` function.

**Value**

A grid viewport object.

**Author(s)**

Paul Murrell

**See Also**

[viewport](#) and [dataViewport](#).

---

pop.viewport      *Pop a Viewport off the Grid Viewport Stack*

---

**Description**

Grid maintains a viewport stack — a list of nested drawing contexts.

This function makes the parent of the specified viewport the new default viewport.

**Usage**

```
pop.viewport(n=1, recording=TRUE)
```

**Arguments**

n	An integer giving the number of viewports to pop. Defaults to 1.
recording	A logical value to indicate whether the set-viewport operation should be recorded on the Grid display list.

**Value**

None.

**Warning**

This function has been deprecated. Please use `popViewport` instead.

**Author(s)**

Paul Murrell

**See Also**

[push.viewport](#).

---

`push.viewport`      *Push a Viewport onto the Grid Viewport Stack*

---

### Description

Grid maintains a viewport stack — a list of nested drawing contexts.

This function makes the specified viewport the default viewport and makes its parent the previous default viewport (i.e., nests the specified context within the previous default context).

### Usage

```
push.viewport(..., recording=TRUE)
```

### Arguments

<code>...</code>	One or more objects of class "viewport", or NULL.
<code>recording</code>	A logical value to indicate whether the set-viewport operation should be recorded on the Grid display list.

### Value

None.

### Warning

This function has been deprecated. Please use `pushViewport` instead.

### Author(s)

Paul Murrell

### See Also

[pop.viewport.](#)

---

Querying the Viewport Tree  
*Get the Current Grid Viewport (Tree)*

---

### Description

`current.viewport()` returns the viewport that Grid is going to draw into.

`current.vpTree` returns the entire Grid viewport tree.

`current.transform` returns the transformation matrix for the current viewport.

**Usage**

```
current.viewport(vp=NULL)
current.vpTree(all=TRUE)
current.transform()
```

**Arguments**

<code>vp</code>	A Grid viewport object. Use of this argument has been deprecated.
<code>all</code>	A logical value indicating whether the entire viewport tree should be returned.

**Details**

If `all` is `FALSE` then `current.vpTree` only returns the subtree below the current viewport.

**Value**

A Grid viewport object.

**Author(s)**

Paul Murrell

**See Also**

[viewport](#)

**Examples**

```
grid.newpage()
pushViewport(viewport(width=0.8, height=0.8, name="A"))
pushViewport(viewport(x=0.1, width=0.3, height=0.6,
  just="left", name="B"))
upViewport(1)
pushViewport(viewport(x=0.5, width=0.4, height=0.8,
  just="left", name="C"))
pushViewport(viewport(width=0.8, height=0.8, name="D"))
upViewport(1)
current.vpTree()
current.viewport()
current.vpTree(all=FALSE)
```

---

<code>stringWidth</code>	<i>Create a Unit Describing the Width of a String</i>
--------------------------	---

---

**Description**

These functions create a unit object describing the width or height of a string.

**Usage**

```
stringWidth(string)
stringHeight(string)
```

**Arguments**

<code>string</code>	A character vector.
---------------------	---------------------

**Value**

A unit object.

**Author(s)**

Paul Murrell

**See Also**

[unit](#) and [grobWidth](#)

---

<code>unit</code>	<i>Function to Create a Unit Object</i>
-------------------	---

---

**Description**

This function creates a unit object — a vector of unit values. A unit value is typically just a single numeric value with an associated unit.

**Usage**

```
unit(x, units, data=NULL)
```

**Arguments**

<code>x</code>	A numeric vector.
<code>units</code>	A character vector specifying the units for the corresponding numeric values.
<code>data</code>	This argument is used to supply extra information for special <code>unit</code> types.

## Details

Unit objects allow the user to specify locations and dimensions in a large number of different coordinate systems. All drawing occurs relative to a viewport and the `units` specifies what coordinate system to use within that viewport.

Possible `units` (coordinate systems) are:

**"npc"** Normalised Parent Coordinates (the default). The origin of the viewport is (0, 0) and the viewport has a width and height of 1 unit. For example, (0.5, 0.5) is the centre of the viewport.

**"cm"** Centimetres.

**"inches"** Inches. 1 in = 2.54 cm.

**"mm"** Millimetres. 10 mm = 1 cm.

**"points"** Points. 72.27 pt = 1 in.

**"picas"** Picas. 1 pc = 12 pt.

**"bigpts"** Big Points. 72 bp = 1 in.

**"dida"** Dida. 1157 dd = 1238 pt.

**"cicero"** Cicero. 1 cc = 12 dd.

**"scaledpts"** Scaled Points. 65536 sp = 1 pt.

**"lines"** Lines of text. Locations and dimensions are in terms of multiples of the default text size of the viewport (as specified by the viewport's `fontsize` and `lineheight`).

**"char"** Multiples of nominal font height of the viewport (as specified by the viewport's `fontsize`).

**"native"** Locations and dimensions are relative to the viewport's `xscale` and `yscale`.

**"snpc"** Square Normalised Parent Coordinates. Same as Normalised Parent Coordinates, except gives the same answer for horizontal and vertical locations/dimensions. It uses the *lesser* of `npc-width` and `npc-height`. This is useful for making things which are a proportion of the viewport, but have to be square (or have a fixed aspect ratio).

**"strwidth"** Multiples of the width of the string specified in the `data` argument. The font size is determined by the `pointsize` of the viewport.

**"strheight"** Multiples of the height of the string specified in the `data` argument. The font size is determined by the `pointsize` of the viewport.

**"grobwidth"** Multiples of the width of the grob specified in the `data` argument.

**"grobheight"** Multiples of the height of the grob specified in the `data` argument.

The `data` argument must be a list when the `unit.length()` is greater than 1. For example, `unit(rep(1, 3), c("npc", "strwidth", "inches"), data=list(NULL, "my string", NULL))`.

It is possible to subset unit objects in the normal way (e.g., `unit(1:5, "npc")[2:4]`), but a special function `unit.c` is provided for combining unit objects.

Certain arithmetic and summary operations are defined for unit objects. In particular, it is possible to add and subtract unit objects (e.g., `unit(1, "npc") - unit(1, "inches")`), and to specify the minimum or maximum of a list of unit objects (e.g., `min(unit(0.5, "npc"), unit(1, "inches"))`).

**Value**

An object of class "unit".

**WARNING**

A special function `unit.length` is provided for determining the number of unit values in a unit object.

The `length` function will work in some cases, but in general will not give the right answer.

There is also a special function `unit.c` for concatenating several unit objects.

The `c` function will not give the right answer.

There used to be "mylines", "mychar", "mystrwidth", "mystrheight" units. These will still be accepted, but work exactly the same as "lines", "char", "strwidth", "strheight".

**Author(s)**

Paul Murrell

**See Also**

[unit.c](#) and [unit.length](#)

**Examples**

```
unit(1, "npc")
unit(1:3/4, "npc")
unit(1:3/4, "npc") + unit(1, "inches")
min(unit(0.5, "npc"), unit(1, "inches"))
unit.c(unit(0.5, "npc"), unit(2, "inches") + unit(1:3/4, "npc"),
        unit(1, "strwidth", "hi there"))
```

---

`unit.c`

*Combine Unit Objects*

---

**Description**

This function produces a new unit object by combining the unit objects specified as arguments.

**Usage**

```
unit.c(...)
```

**Arguments**

...           An arbitrary number of unit objects.

**Value**

An object of class `unit`.

**Author(s)**

Paul Murrell

**See Also**

[unit](#).

---

<code>unit.length</code>	<i>Length of a Unit Object</i>
--------------------------	--------------------------------

---

**Description**

The length of a unit object is defined as the number of unit values in the unit object.

**Usage**

```
unit.length(unit)
```

**Arguments**

`unit`          A unit object.

**Value**

An object of class `unit`.

**Author(s)**

Paul Murrell

**See Also**

[unit](#)

unit.pmin

*Parallel Unit Minima and Maxima*

---

**Description**

Returns a unit object whose *i*'th value is the minimum (or maximum) of the *i*'th values of the arguments.

**Usage**

```
unit.pmin(...)  
unit.pmax(...)
```

**Arguments**

...            One or more unit objects.

**Details**

The length of the result is the maximum of the lengths of the arguments; shorter arguments are recycled in the usual manner.

**Value**

A unit object.

**Author(s)**

Paul Murrell

**Examples**

```
max(unit(1:3, "cm"), unit(0.5, "npc"))  
unit.pmax(unit(1:3, "cm"), unit(0.5, "npc"))
```

---

unit.rep*Replicate Elements of Unit Objects*

---

**Description**

Replicates the units according to the values given in `times` and `length.out`.

**Usage**

```
unit.rep(x, times, length.out)
```

**Arguments**

x	An object of class "unit".
times	integer. A vector giving the number of times to repeat each element. Either of length 1 or length(x).
length.out	integer. (Optional.) The desired length of the output vector.

**Value**

An object of class "unit".

**Author(s)**

Paul Murrell

**See Also**

[rep](#)

**Examples**

```
unit.rep(unit(1:3, "npc"), 3)
unit.rep(unit(1:3, "npc"), 1:3)
unit.rep(unit(1:3, "npc") + unit(1, "inches"), 3)
unit.rep(max(unit(1:3, "npc") + unit(1, "inches")), 3)
unit.rep(max(unit(1:3, "npc") + unit(1, "strwidth", "a"))*4, 3)
unit.rep(unit(1:3, "npc") + unit(1, "strwidth", "a")*4, 3)
```

---

validDetails      *Customising grid grob Validation*

---

**Description**

This generic hook function is called whenever a grid grob is created or edited via `grob`, `gTree`, `grid.edit` or `editGrob`. This provides an opportunity for customising the validation of a new class derived from `grob` (or `gTree`).

**Usage**

```
validDetails(x)
```

**Arguments**

x	A grid grob.
---	--------------

**Details**

This function is called by `grob`, `gTree`, `grid.edit` and `editGrob`. A method should be written for classes derived from `grob` or `gTree` to validate the values of slots specific to the new class. (e.g., see `grid::validDetails.axis`).

Note that the standard slots for grobs and gTrees are automatically validated (e.g., `vp`, `gp` slots for grobs and, in addition, `children`, and `childrenvp` slots for gTrees) so only slots specific to a new class need to be addressed.

**Value**

The function MUST return the validated grob.

**Author(s)**

Paul Murrell

**See Also**

[grid.edit](#)

---

 vpPath

---

*Concatenate Viewport Names*


---

**Description**

This function can be used to generate a viewport path for use in `downViewport` or `seekViewport`.

A viewport path is a list of nested viewport names.

**Usage**

```
vpPath(...)
```

**Arguments**

... Character values which are viewport names.

**Details**

Viewport names must only be unique amongst viewports which share the same parent in the viewport tree.

This function can be used to generate a specification for a viewport that includes the viewport's parent's name (and the name of its parent and so on).

For interactive use, it is possible to directly specify a path, but it is strongly recommended that this function is used otherwise in case the path separator is changed in future versions of grid.

**Value**

A vpPath object.

**See Also**

[viewport](#), [pushViewport](#), [popViewport](#), [downViewport](#), [seekViewport](#), [upViewport](#)

**Examples**

```
vpPath("vp1", "vp2")
```

---

widthDetails	<i>Width and Height of a grid grob</i>
--------------	--

---

**Description**

These generic functions are used to determine the size of grid grobs.

**Usage**

```
widthDetails(x)  
heightDetails(x)
```

**Arguments**

x                   A grid grob.

**Details**

These functions are called in the calculation of "grobwidth" and "grobheight" units. Methods should be written for classes derived from grob or gTree where the size of the grob can be determined (see, for example `grid::widthDetails.frame`).

**Value**

A unit object.

**Author(s)**

Paul Murrell

**See Also**

[absolute.size](#).

---

 Working with Viewports

*Maintaining and Navigating the Grid Viewport Tree*


---

**Description**

Grid maintains a tree of viewports — nested drawing contexts.

These functions provide ways to add or remove viewports and to navigate amongst viewports in the tree.

**Usage**

```
pushViewport(..., recording=TRUE)
popViewport(n, recording=TRUE)
downViewport(name, strict=FALSE, recording=TRUE)
seekViewport(name, recording=TRUE)
upViewport(n, recording=TRUE)
```

**Arguments**

<code>...</code>	One or more objects of class "viewport".
<code>n</code>	An integer value indicating how many viewports to pop or navigate up. The special value 0 indicates to pop or navigate viewports right up to the root viewport.
<code>name</code>	A character value to identify a viewport in the tree.
<code>strict</code>	A boolean indicating whether the vpPath must be matched exactly.
<code>recording</code>	A logical value to indicate whether the viewport operation should be recorded on the Grid display list.

**Details**

Objects created by the `viewport()` function are only descriptions of a drawing context. A viewport object must be pushed onto the viewport tree before it has any effect on drawing.

The viewport tree always has a single root viewport (created by the system) which corresponds to the entire device (and default graphical parameter settings). Viewports may be added to the tree using `pushViewport()` and removed from the tree using `popViewport()`.

There is only ever one current viewport, which is the current position within the viewport tree. All drawing and viewport operations are relative to the current viewport. When a viewport is pushed it becomes the current viewport. When a viewport is popped, the parent viewport becomes the current viewport. Use `upViewport` to navigate to the parent of the current viewport, without removing the current viewport from the viewport tree. Use `downViewport` to navigate to a viewport further down the viewport tree and `seekViewport` to navigate to a viewport anywhere else in the tree.

If a viewport is pushed and it has the same name as a viewport at the same level in the tree, then it replaces the existing viewport in the tree.

**Value**

`downViewport` returns the number of viewports it went down.

This can be useful for returning to your starting point by doing something like `depth <- downViewport()` then `upViewport(depth)`.

**Author(s)**

Paul Murrell

**See Also**

[viewport](#) and [vpPath](#).

**Examples**

```
# push the same viewport several times
grid.newpage()
vp <- viewport(width=0.5, height=0.5)
pushViewport(vp)
grid.rect(gp=gpar(col="blue"))
grid.text("Quarter of the device",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="blue"))
pushViewport(vp)
grid.rect(gp=gpar(col="red"))
grid.text("Quarter of the parent viewport",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="red"))
popViewport(2)
# push several viewports then navigate amongst them
grid.newpage()
grid.rect(gp=gpar(col="grey"))
grid.text("Top-level viewport",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="grey"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(width=0.8, height=0.7, name="A"))
grid.rect(gp=gpar(col="blue"))
grid.text("1. Push Viewport A",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(x=0.1, width=0.3, height=0.6,
  just="left", name="B"))
grid.rect(gp=gpar(col="red"))
grid.text("2. Push Viewport B (in A)",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="red"))
if (interactive()) Sys.sleep(1.0)
upViewport(1)
grid.text("3. Up from B to A",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(x=0.5, width=0.4, height=0.8,
  just="left", name="C"))
grid.rect(gp=gpar(col="green"))
```

```
grid.text("4. Push Viewport C (in A)",
  y=unit(1, "npc") - unit(1, "lines"), gp=gpar(col="green"))
if (interactive()) Sys.sleep(1.0)
pushViewport(viewport(width=0.8, height=0.6, name="D"))
grid.rect()
grid.text("5. Push Viewport D (in C)",
  y=unit(1, "npc") - unit(1, "lines"))
if (interactive()) Sys.sleep(1.0)
upViewport(0)
grid.text("6. Up from D to top-level",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="grey"))
if (interactive()) Sys.sleep(1.0)
downViewport("D")
grid.text("7. Down from top-level to D",
  y=unit(1, "npc") - unit(2, "lines"))
if (interactive()) Sys.sleep(1.0)
seekViewport("B")
grid.text("8. Seek from D to B",
  y=unit(1, "npc") - unit(2, "lines"), gp=gpar(col="red"))
pushViewport(viewport(width=0.9, height=0.5, name="A"))
grid.rect()
grid.text("9. Push Viewport A (in B)",
  y=unit(1, "npc") - unit(1, "lines"))
if (interactive()) Sys.sleep(1.0)
seekViewport("A")
grid.text("10. Seek from B to A (in ROOT)",
  y=unit(1, "npc") - unit(3, "lines"), gp=gpar(col="blue"))
if (interactive()) Sys.sleep(1.0)
seekViewport(vpPath("B", "A"))
grid.text("11. Seek from\nA (in ROOT)\nto A (in B)")
popViewport(0)
```

## Chapter 6

# The methods package

---

`.BasicFunsList`      *List of Builtin and Special Functions*

---

### Description

A named list providing instructions for turning builtin and special functions into generic functions.

Functions in R that are defined as `.Primitive(<name>)` are not suitable for formal methods, because they lack the basic reflectance property. You can't find the argument list for these functions by examining the function object itself.

Future versions of R may fix this by attaching a formal argument list to the corresponding function. While generally the names of arguments are not checked by the internal code implementing the function, the number of arguments frequently is.

In any case, some definition of a formal argument list is needed if users are to define methods for these functions. In particular, if methods are to be merged from multiple packages, the different sets of methods need to agree on the formal arguments.

In the absence of reflectance, this list provides the relevant information via a dummy function associated with each of the known specials for which methods are allowed.

At the same, the list flags those specials for which methods are meaningless (e.g., `for`) or just a very bad idea (e.g., `.Primitive`).

A generic function created via `setMethod`, for example, for one of these special functions will have the argument list from `.BasicFunsList`. If no entry exists, the argument list `(x, ...)` is assumed.

as

*Force an Object to Belong to a Class***Description**

These functions manage the relations that allow coercing an object to a given class.

**Usage**

```
as(object, Class, strict=TRUE, ext)
```

```
as(object, Class) <- value
```

```
setAs(from, to, def, replace, where = topenv(parent.frame()))
```

**Arguments**

object	Any object.
Class	The name of the class to which <code>object</code> should be coerced.
strict	A logical flag. If <code>TRUE</code> , the returned object must be strictly from the target class (unless that class is a virtual class, in which case the object will be from the closest actual class (often the original object, if that class extends the virtual class directly). If <code>strict = FALSE</code> , any simple extension of the target class will be returned, without further change. A simple extension is, roughly, one that just adds slots to an existing class.
value	The value to use to modify <code>object</code> (see the discussion below). You should supply an object with class <code>Class</code> ; some coercion is done, but you're unwise to rely on it.
from, to	The classes between which <code>def</code> performs coercion. (In the case of the <code>coerce</code> function these are objects from the classes, not the names of the classes, but you're not expected to call <code>coerce</code> directly.)
def	A function of one argument. It will get an object from class <code>from</code> and had better return an object of class <code>to</code> . (If you want to save <code>setAs</code> a little work, make the name of the argument <code>from</code> , but don't worry about it, <code>setAs</code> will do the conversion.)
replace	If supplied, the function to use as a replacement method.
where	the position or environment in which to store the resulting method for <code>coerce</code> .
ext	the optional object defining how <code>Class</code> is extended by the class of the object (as returned by <code>possibleExtends</code> ). This argument is used internally (to provide essential information for non-public classes), but you are unlikely to want to use it directly.

## Summary of Functions

**as:** Returns the version of this object coerced to be the given `Class`.

If the corresponding `is(object, Class)` relation is true, it will be used. In particular, if the relation has a `coerce` method, the method will be invoked on `object`. However, if the object's class extends `Class` in a simple way (e.g, by including the superclass in the definition, then the actual coercion will be done only if `strict` is `TRUE` (non-strict coercion, is used in passing objects to methods).

Coerce methods are pre-defined for basic classes (including all the types of vectors, functions and a few others). See `showMethods(coerce)` for a list of these.

Beyond these two sources of methods, further methods are defined by calls to the `setAs` function.

**coerce:** Coerce `from` to be of the same class as `to`.

Not a function you should usually call explicitly. The function `setAs` creates methods for `coerce` for the `as` function to use.

**setAs:** The function supplied as the third argument is to be called to implement `as(x, to)` when `x` has class `from`. Need we add that the function should return a suitable object with class `to`.

## How Functions 'as' and 'setAs' Work

The function `as` contrives to turn `object` into an object with class `Class`. In doing so, it uses information about classes and methods, but in a somewhat special way. Keep in mind that objects from one class can turn into objects from another class either automatically or by an explicit call to the `as` function. Automatic conversion is special, and comes from the designer of one class of objects asserting that this class extends another class (see `setClass` and `setIs`).

Because inheritance is a powerful assertion, it should be used sparingly (otherwise your computations may produce unexpected, and perhaps incorrect, results). But objects can also be converted explicitly, by calling `as`, and that conversion is designed to use any inheritance information, as well as explicit methods.

As a first step in conversion, the `as` function determines whether `is(object, Class)` is `TRUE`. This can be the case either because the class definition of `object` includes `Class` as a "super class" (directly or indirectly), or because a call to `setIs` established the relationship.

Either way, the inheritance relation defines a method to coerce `object` to `Class`. In the most common case, the method is just to extract from `object` the slots needed for `Class`, but it's also possible to specify a method explicitly in a `setIs` call.

So, if inheritance applies, the `as` function calls the appropriate method. If inheritance does not apply, and `coerceFlag` is `FALSE`, `NULL` is returned.

By default, `coerceFlag` is `TRUE`. In this case the `as` function goes on to look for a method for the function `coerce` for the signature `c(from = class(object), to = Class)`.

Method selection is used in the `as` function in two special ways. First, inheritance is applied for the argument `from` but not for the argument `to` (if you think about it, you'll probably agree that you wouldn't want the result to be from some class other than the `Class` specified). Second, the function tries to use inheritance information to convert the object indirectly, by first converting it to an inherited class. It does this by examining the classes that the `from` class extends, to see if

any of them has an explicit conversion method. Suppose class "by" does: Then the `as` function implicitly computes `as(as(object, "by"), Class)`.

With this explanation as background, the function `setAs` does a fairly obvious computation: It constructs and sets a method for the function `coerce` with signature `c(from, to)`, using the `def` argument to define the body of the method. The function supplied as `def` can have one argument (interpreted as an object to be coerced) or two arguments (the `from` object and the `to` class). Either way, `setAs` constructs a function of two arguments, with the second defaulting to the name of the `to` class. The method will be called from `as` with the object as the only argument: The default for the second argument is provided so the method can know the intended `to` class.

The function `coerce` exists almost entirely as a repository for such methods, to be selected as described above by the `as` function. In fact, it would usually be a bad idea to call `coerce` directly, since then you would get inheritance on the `to` argument; as mentioned, this is not likely to be what you want.

### The Function 'as' Used in Replacements

When `as` appears on the left of an assignment, the intuitive meaning is "Replace the part of `object` that was inherited from `Class` by the `value` on the right of the assignment."

This usually has a straightforward interpretation, but you can control explicitly what happens, and sometimes you should to avoid possible corruption of objects.

When `object` inherits from `Class` in the usual way, by including the slots of `Class`, the default `as` method is to set the corresponding slots in `object` to those in `value`.

The default computation may be reasonable, but usually only if all *other* slots in `object` are unrelated to the slots being changed. Often, however, this is not the case. The class of `object` may have extended `Class` with a new slot whose value depends on the inherited slots. In this case, you may want to define a method for replacing the inherited information that recomputes all the dependent information. Or, you may just want to prohibit replacing the inherited information directly.

The way to control such replacements is through the `replace` argument to function `setIs`. This argument is a method that function `as` calls when used for replacement. It can do whatever you like, including calling `stop` if you want to prohibit replacements. It should return a modified object with the same class as the `object` argument to `as`.

In R, you can also explicitly supply a replacement method, even in the case that inheritance does not apply, through the `replace` argument to `setAs`. It works essentially the same way, but in this case by constructing a method for `"coerce<-"`. (Replace methods for coercion without inheritance are not in the original description and so may not be compatible with S-Plus, at least not yet.)

When inheritance does apply, `coerce` and `replace` methods can be specified through either `setIs` or `setAs`; the effect is essentially the same.

### Basic Coercion Methods

Methods are pre-defined for coercing any object to one of the basic datatypes. For example, `as(x, "numeric")` uses the existing `as.numeric` function. These built-in methods can be listed by `showMethods("coerce")`.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## Examples

```
## using the definition of class "track" from Classes

setAs("track", "numeric", function(from)from@y)

t1 <- new("track", x=1:20, y=(1:20)^2)

as(t1, "numeric")

## The next example shows:
## 1. A virtual class to define setAs for several classes at once.
## 2. as() using inherited information

setClass("ca", representation(a = "character", id = "numeric"))

setClass("cb", representation(b = "character", id = "numeric"))

setClass("id")
setIs("ca", "id")
setIs("cb", "id")

setAs("id", "numeric", function(from) from@id)

CA <- new("ca", a = "A", id = 1)

CB <- new("cb", b = "B", id = 2)

setAs("cb", "ca", function(from, to )new(to, a=from@b, id = from@id))

as(CB, "numeric")
```

**Description**

Formal classes exist corresponding to the basic R data types, allowing these types to be used in method signatures, as slots in class definitions, and to be extended by new classes.

**Usage**

```
### The following are all basic vector classes.
### They can appear as class names in method signatures,
### in calls to as(), is(), and new().
"character"
"complex"
"double"
"expression"
"integer"
"list"
"logical"
"numeric"
"single"

### the class
"vector"
### is a virtual class, extended by all the above

### The following are additional basic classes
"NULL"      # NULL objects
"function"  # function objects, including primitives
"externalptr" # raw external pointers for use in C code

"ANY"      # virtual classes used by the methods package itself
"VIRTUAL"
"missing"
```

**Objects from the Classes**

Objects can be created by calls of the form `new(Class, ...)`, where `Class` is the quoted class name, and the remaining arguments if any are objects to be interpreted as vectors of this class. Multiple arguments will be concatenated.

The class "expression" is slightly odd, in that the `...` arguments will *not* be evaluated; therefore, don't enclose them in a call to `quote()`.

**Extends**

Class "vector", directly.

## Methods

**coerce** Methods are defined to coerce arbitrary objects to these classes, by calling the corresponding basic function, for example, `as(x, "numeric")` calls `as.numeric(x)`.

---

<code>callNextMethod</code>	<i>Call an Inherited Method</i>
-----------------------------	---------------------------------

---

## Description

A call to `callNextMethod` can only appear inside a method definition. It then results in a call to the first inherited method after the current method, with the arguments to the current method passed down to the next method. The value of that method call is the value of `callNextMethod`.

## Usage

```
callNextMethod(...)
```

## Arguments

...      Optionally, the arguments to the function in its next call (but note that the dispatch is as in the detailed description below; the arguments have no effect on selecting the next method.)

If no arguments are included in the call to `callNextMethod`, the effect is to call the method with the current arguments. See the detailed description for what this really means.

Calling with no arguments is often the natural way to use `callNextMethod`; see the examples.

## Details

The “next” method (i.e., the first inherited method) is defined to be that method which *would* have been called if the current method did not exist. This is more-or-less literally what happens: The current method is deleted from a copy of the methods for the current generic, and `selectMethod` is called to find the next method (the result is cached in a special object, so the search only typically happens once per session per combination of argument classes).

It is also legal, and often useful, for the method called by `callNextMethod` to itself have a call to `callNextMethod`. This generally works as you would expect, but for completeness be aware that it is possible to have ambiguous inheritance in the S structure, in the sense that the same two classes can appear as superclasses *in the opposite order* in two other class definitions. In this case the effect of a nested instance of `callNextMethod` is not well defined. Such inconsistent class hierarchies are both rare and nearly always the result of bad design, but they are possible, and currently undetected.

The statement that the method is called with the current arguments is more precisely as follows. Arguments that were missing in the current call are still missing (remember that “missing” is a valid class in a method signature). For a formal argument, say `x`, that appears in the original call, there is a corresponding argument in the next method call equivalent to “`x = x`”. In effect, this means that the next method sees the same actual arguments, but arguments are evaluated only once.

**Value**

The value returned by the selected method.

**References**

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

**See Also**

[Methods](#) for the general behavior of method dispatch

**Examples**

```
## some class definitions with simple inheritance
setClass("B0" , representation(b0 = "numeric"))

setClass("B1", representation(b1 = "character"), contains = "B0")

setClass("B2", representation(b2 = "logical"), contains = "B1")

## and a rather silly function to illustrate callNextMethod

f <- function(x) class(x)

setMethod("f", "B0", function(x) c(x@b0^2, callNextMethod()))
setMethod("f", "B1", function(x) c(paste(x@b1, ":"), callNextMethod()))
setMethod("f", "B2", function(x) c(x@b2, callNextMethod()))

b1 <- new("B1", b0 = 2, b1 = "Testing")

b2 <- new("B2", b2 = FALSE, b1 = "More testing", b0 = 10)

f(b2)

f(b1)
```

## Description

Class definitions are objects that contain the formal definition of a class of R objects.

## Details

When a class is defined, an object is stored that contains the information about that class, including:

**slots** Each slot is a component object. Like elements of a list these may be extracted (by name) and set. However, they differ from list components in important ways.

All the objects from a particular class have the same set of slot names; specifically, the slot names that are contained in the class definition. Each slot in each object always has the same class; again, this is defined by the overall class definition.

Classes don't need to have any slots, and many useful classes do not. These objects usually extend other, simple objects, such as numeric or character vectors. Finally, classes can have no data at all—these are known as *virtual* classes and are in fact very important programming tools. They are used to group together ordinary classes that want to share some programming behavior, without necessarily restricting how the behavior is implemented.

**extends** The names of the classes that this class extends. A class `Fancy`, say, extends a class `Simple` if an object from the `Fancy` class has all the capabilities of the `Simple` class (and probably some more as well). In particular, and very usefully, any method defined to work for a `Simple` object can be applied to a `Fancy` object as well.

In other programming languages, this relationship is sometimes expressed by saying that `Simple` is a superclass of `Fancy`, or that `Fancy` is a subclass of `Simple`.

The actual class definition object contains the names of all the classes this class extends. But those classes can themselves extend other classes also, so the complete extension can only be known by obtaining all those class definitions.

Class extension is usually defined when the class itself is defined, by including the names of superclasses as unnamed elements in the representation argument to `setClass`.

An object from a given class will then have all the slots defined for its own class *and* all the slots defined for its superclasses as well.

Note that `extends` relations can be defined in other ways as well, by using the `setIs` function.

**prototype** Each class definition contains a prototype object from the class. This must have all the slots, if any, defined by the class definition.

The prototype most commonly just consists of the prototypes of all its slots. But that need not be the case: the definition of the class can specify any valid object for any of the slots.

There are a number of “basic” classes, corresponding to the ordinary kinds of data occurring in R. For example, `numeric` is a class corresponding to numeric vectors. These classes are predefined and can then be used as slots or as superclasses for any other class definitions. The prototypes for the vector classes are vectors of length 0 of the corresponding type.

There are also a few basic virtual classes, the most important being "vector", grouping together all the vector classes; and "language", grouping together all the types of objects making up the R language.

**Author(s)**

John Chambers

**References**

The web page <http://www.omegahat.org/RMethods/index.html> is the primary documentation.

The functions in this package emulate the facility for classes and methods described in *Programming with Data* (John M. Chambers, Springer, 1998). See this book for further details and examples.

**See Also**

[Methods](#), [setClass](#), [is](#), [as](#), [new](#), [slot](#)

---

classRepresentation-class  
*Class Objects*

---

**Description**

These are the objects that hold the definition of classes of objects. They are constructed and stored as meta-data by calls to the function [setClass](#). Don't manipulate them directly, except perhaps to look at individual slots.

**Details**

Class definitions are stored as metadata in various packages. Additional metadata supplies information on inheritance (the result of calls to [setIs](#)). Inheritance information implied by the class definition itself (because the class contains one or more other classes) is also constructed automatically.

When a class is to be used in an R session, this information is assembled to complete the class definition. The completion is a second object of class "classRepresentation", cached for the session or until something happens to change the information. A call to [getClass](#) returns the completed definition of a class; a call to [getClassDef](#) returns the stored definition (uncompleted).

In particular, completion fills in the upward- and downward-pointing inheritance information for the class, in slots `contains` and `subclasses` respectively. It's in principle important to note that this information can depend on which packages are installed, since these may define additional subclasses or superclasses.

## Slots

**slots:** A named list of the slots in this class; the elements of the list are the classes to which the slots must belong (or extend), and the names of the list gives the corresponding slot names.

**contains:** A named list of the classes this class “contains”; the elements of the list are objects of [SClassExtension-class](#). The list may be only the direct extensions or all the currently known extensions (see the details).

**virtual:** Logical flag, set to TRUE if this is a virtual class.

**prototype:** The object that represents the standard prototype for this class; i.e., the data and slots returned by a call to [new](#) for this class with no special arguments. Don't mess with the prototype object directly.

**validity:** Optionally, a function to be used to test the validity of objects from this class. See [validObject](#).

**access:** Access control information. Not currently used.

**className:** The character string name of the class.

**package:** The character string name of the package to which the class belongs. Nearly always the package on which the metadata for the class is stored, but in operations such as constructing inheritance information, the internal package name rules.

**subclasses:** A named list of the classes known to extend this class'; the elements of the list are objects of [SClassExtension-class](#). The list is currently only filled in when completing the class definition (see the details).

**versionKey:** Object of class "externalptr"; eventually will perhaps hold some versioning information, but not currently used.

**sealed:** Object of class "logical"; is this class sealed? If so, no modifications are allowed.

## See Also

See function [setClass](#) to supply the information in the class definition. See [Classes](#) for a more basic discussion of class information.

## Description

Special documentation can be supplied to describe the classes and methods that are created by the software in the methods package. Techniques to access this documentation and to create it in R help files are described here.

### Getting documentation on classes and methods

You can ask for on-line help for class definitions, for specific methods for a generic function, and for general discussion of methods for a generic function. These requests use the `?` operator (see [help](#) for a general description of the operator). Of course, you are at the mercy of the implementer as to whether there *is* any documentation on the corresponding topics.

Documentation on a class uses the argument `class` on the left of the `?`, and the name of the class on the right; for example,

```
class ? genericFunction
```

to ask for documentation on the class "genericFunction".

When you want documentation for the methods defined for a particular function, you can ask either for a general discussion of the methods or for documentation of a particular method (that is, the method that would be selected for a particular set of actual arguments).

Overall methods documentation is requested by calling the `?` operator with `methods` as the left-side argument and the name of the function as the right-side argument. For example,

```
methods ? initialize
```

asks for documentation on the methods for the `initialize` function.

Asking for documentation on a particular method is done by giving a function call expression as the right-hand argument to the `"?"` operator. There are two forms, depending on whether you prefer to give the class names for the arguments or expressions that you intend to use in the actual call.

If you planned to evaluate a function call, say `myFun(x, sqrt(wt))` and wanted to find out something about the method that would be used for this call, put the call on the right of the `"?"` operator:

```
?myFun(x, sqrt(wt))
```

A method will be selected, as it would be for the call itself, and documentation for that method will be requested. If `myFun` is not a generic function, ordinary documentation for the function will be requested.

If you know the actual classes for which you would like method documentation, you can supply these explicitly in place of the argument expressions. In the example above, if you want method documentation for the first argument having class "maybeNumber" and the second "logical", call the `"?"` operator, this time with a left-side argument `method`, and with a function call on the right using the class names as arguments:

```
method ? myFun("maybeNumber", "logical")
```

Once again, a method will be selected, this time corresponding to the specified classes, and method documentation will be requested. This version only works with generic functions.

The two forms each have advantages. The version with actual arguments doesn't require you to figure out (or guess at) the classes of the arguments. On the other hand, evaluating the arguments may take some time, depending on the example. The version with class names does require you to pick classes, but it's otherwise unambiguous. It has a subtler advantage, in that the classes supplied may be virtual classes, in which case no actual argument will have specifically this class. The class "maybeNumber", for example, might be a class union (see the example for [setClassUnion](#)).

In either form, methods will be selected as they would be in actual computation, including use of inheritance and group generic functions. See [selectMethod](#) for the details, since it is the function used to find the appropriate method.

## Writing Documentation for Methods

The on-line documentation for methods and classes uses some extensions to the R documentation format to implement the requests for class and method documentation described above. See the document *Writing R Extensions* for the available markup commands (you should have consulted this document already if you are at the stage of documenting your software).

In addition to the specific markup commands to be described, you can create an initial, overall file with a skeleton of documentation for the methods defined for a particular generic function:

```
promptMethods("myFun")
```

will create a file, 'myFun-methods.Rd' with a skeleton of documentation for the methods defined for function myFun. The output from `promptMethods` is suitable if you want to describe all or most of the methods for the function in one file, separate from the documentation of the generic function itself. Once the file has been filled in and moved to the 'man' subdirectory of your source package, requests for methods documentation will use that file, both for specific methods documentation as described above, and for overall documentation requested by

```
methods ? myFun
```

You are not required to use `promptMethods`, and if you do, you may not want to use the entire file created:

- If you want to document the methods in the file containing the documentation for the generic function itself, you can cut-and-paste to move the `\alias` lines and the `Methods` section from the file created by `promptMethods` to the existing file.
- On the other hand, if these are auxiliary methods, and you only want to document the added or modified software, you should strip out all but the relevant `\alias` lines for the methods of interest, and remove all but the corresponding `\item` entries in the `Methods` section. Note that in this case you will usually remove the first `\alias` line as well, since that is the marker for general methods documentation on this function (in the example, `\alias{myfun-methods}`).

If you simply want to direct documentation for one or more methods to a particular R documentation file, insert the appropriate alias.

---

```
environment-class Class "environment"
```

---

### Description

A formal class for R environments.

### Objects from the Class

Objects can be created by calls of the form `new("environment", ...)`. The arguments in ..., if any, should be named and will be assigned to the newly created environment.

**Methods**

**coerce** signature(from = "ANY", to = "environment"): calls [as.environment](#).

**initialize** signature(object = "environment"): Implements the assignments in the new environment. Note that the `object` argument is ignored; a new environment is *always* created, since environments are not protected by copying.

**See Also**

[new.env](#)

---

 fixPre1.8

---

*Fix Objects Saved from R Versions Previous to 1.8*


---

**Description**

Beginning with R version 1.8.0, the class of an object contains the identification of the package in which the class is defined. The function `fixPre1.8` fixes and re-assigns objects missing that information (typically because they were loaded from a file saved with a previous version of R.)

**Usage**

```
fixPre1.8(names, where)
```

**Arguments**

<code>names</code>	Character vector of the names of all the objects to be fixed and re-assigned.
<code>where</code>	The environment from which to look for the objects, and for class definitions. Defaults to the top environment of the call to <code>fixPre1.8</code> , the global environment if the function is used interactively.

**Details**

The named object will be saved where it was found. Its class attribute will be changed to the full form required by R 1.8; otherwise, the contents of the object should be unchanged.

Objects will be fixed and re-assigned only if all the following conditions hold:

1. The named object exists.
2. It is from a defined class (not a basic datatype which has no actual class attribute).
3. The object appears to be from an earlier version of R.
4. The class is currently defined.
5. The object is consistent with the current class definition.

If any condition except the second fails, a warning message is generated.

Note that `fixPre1.8` currently fixes *only* the change in class attributes. In particular, it will not fix binary versions of packages installed with earlier versions of R if these use incompatible features. Such packages must be re-installed from source, which is the wise approach always when major version changes occur in R.

**Value**

The names of all the objects that were in fact re-assigned.

---

genericFunction-class

*Generic Function Objects*

---

**Description**

Generic functions (objects from or extending class `genericFunction`) are extended function objects, containing information used in creating and dispatching methods for this function. They also identify the package associated with the function and its methods.

**Objects from the Class**

Generic functions are created and assigned by `setGeneric` or `setGroupGeneric` and, indirectly, by `setMethod`.

As you might expect `setGeneric` and `setGroupGeneric` create objects of class "genericFunction" and "groupGenericFunction" respectively.

**Slots**

**.Data:** Object of class "function", the function definition of the generic, usually created automatically as a call to `standardGeneric`.

**generic:** Object of class "character", the name of the generic function.

**package:** Object of class "character", the name of the package to which the function definition belongs (and *not* necessarily where the generic function is stored). If the package is not specified explicitly in the call to `setGeneric`, it is usually the package on which the corresponding non-generic function exists.

**group:** Object of class "list", the group or groups to which this generic function belongs. Empty by default.

**valueClass:** Object of class "character"; if not an empty character vector, identifies one or more classes. It is asserted that all methods for this function return objects from these class (or from classes that extend them).

**signature:** Object of class "character", the vector of formal argument names that can appear in the signature of methods for this generic function. By default, it is all the formal arguments, except for `...`. Order matters for efficiency: the most commonly used arguments in specifying methods should come first.

**default:** Object of class "OptionalMethods", the default method for this function. Generated automatically and used to initialize method dispatch.

**skeleton:** Object of class "call", a slot used internally in method dispatch. Don't expect to use it directly.

**Extends**

Class "function", from data part.  
Class "OptionalMethods", by class "function".  
Class "PossibleMethod", by class "function".

**Methods**

Generic function objects are used in the creation and dispatch of formal methods; information from the object is used to create methods list objects and to merge or update the existing methods for this generic.

---

GenericFunctions    *Tools for Managing Generic Functions*

---

**Description**

The functions documented here manage collections of methods associated with a generic function, as well as providing information about the generic functions themselves.

**Usage**

```
isGeneric(f, where, fdef, getName = FALSE)
isGroup(f, where, fdef)
removeGeneric(f, where)
standardGeneric(f)
dumpMethod(f, signature, file, where, def)
findFunction(f, generic=TRUE)
dumpMethods(f, file, signature, methods, where)
signature(...)
removeMethods(f, where)
setReplaceMethod(f, ...)
getGenerics(where, searchForm = FALSE)
allGenerics(where, searchForm = FALSE)
callGeneric(...)
```

**Arguments**

<code>f</code>	The character string naming the function.
<code>where</code>	The environment, namespace, or search-list position from which to search for objects. By default, start at the top-level environment of the calling function, typically the global environment (i.e., use the search list), or the namespace of a package from which the call came. It is important to supply this argument when calling any of these functions indirectly. With package namespaces, the default is likely to be wrong in such calls.
<code>signature</code>	The class signature of the relevant method. A signature is a named or unnamed vector of character strings. If named, the names must be formal argument names for the generic function. If <code>signature</code> is unnamed, the default is to use the first <code>length(signature)</code> formal arguments of the function.
<code>file</code>	The file on which to dump method definitions.
<code>def</code>	The function object defining the method; if omitted, the current method definition corresponding to the signature.
<code>...</code>	Named or unnamed arguments to form a signature.
<code>generic</code>	In testing or finding functions, should generic functions be included. Supply as <code>FALSE</code> to get only non-generic functions.
<code>fdef</code>	Optional, the generic function definition. Usually omitted in calls to <code>isGeneric</code>
<code>getName</code>	If <code>TRUE</code> , <code>isGeneric</code> returns the name of the generic. By default, it returns <code>TRUE</code> .
<code>methods</code>	The methods object containing the methods to be dumped. By default, the methods defined for this generic (optionally on the specified <code>where</code> location).
<code>searchForm</code>	In <code>getGenerics</code> , if <code>TRUE</code> , the <code>package</code> slot of the returned result is in the form used by <code>search()</code> , otherwise as the simple package name (e.g, "package:base" vs "base").

**Summary of Functions**

**isGeneric:** Is there a function named `f`, and if so, is it a generic?

The `getName` argument allows a function to find the name from a function definition. If it is `TRUE` then the name of the generic is returned, or `FALSE` if this is not a generic function definition.

The behavior of `isGeneric` and `getGeneric` for primitive functions is slightly different. These functions don't exist as formal function objects (for efficiency and historical reasons), regardless of whether methods have been defined for them. A call to `isGeneric` tells you whether methods have been defined for this primitive function, anywhere in the current search list, or in the specified position `where`. In contrast, a call to `getGeneric` will return what the generic for that function would be, even if no methods have been currently defined for it.

**removeGeneric, removeMethods:** Remove the all the methods for the generic function of this name. In addition, `removeGeneric` removes the function itself; `removeMethods` restores the non-generic function which was the default method. If there was no default method, `removeMethods` leaves a generic function with no methods.

**standardGeneric:** Dispatches a method from the current function call for the generic function `f`. It is an error to call `standardGeneric` anywhere except in the body of the corresponding generic function.

**getMethods:** The list of methods for the specified generic.

**dumpMethod:** Dump the method for this generic function and signature.

**findFunction:** return a list of either the positions on the search list, or the current top-level environment, on which a function object for name exists. The returned value is *always* a list, use the first element to access the first visible version of the function. See the example.

*NOTE:* Use this rather than `find` with `mode="function"`, which is not as meaningful, and has a few subtle bugs from its use of regular expressions. Also, `findFunction` works correctly in the code for a package when attaching the package via a call to `library`.

**selectMethod:** Returns the method (a function) that R would use to evaluate a call to this generic, with arguments corresponding to the specified signature.

`f` = the name of the generic function, `signature` is the signature of classes to match to the arguments of `f`.

**dumpMethods:** Dump all the methods for this generic.

**signature:** Returns a named list of classes to be matched to arguments of a generic function.

**getGenerics:** Returns the names of the generic functions that have methods defined on `where`; this argument can be an environment or an index into the search list. By default, the whole search list is used.

The methods definitions are stored with package qualifiers; for example, methods for function "initialize" might refer to two different functions of that name, on different packages. The package names corresponding to the method list object are contained in the slot `package` of the returned object. The form of the returned name can be plain (e.g., "base"), or in the form used in the search list ("`package:base`") according to the value of `searchForm`

**callGeneric:** In the body of a method, this function will make a call to the current generic function. If no arguments are passed to `callGeneric`, the arguments to the current call are passed down; otherwise, the arguments are interpreted as in a call to the generic function.

## Details

**setGeneric:** If there is already a non-generic function of this name, it will be used to define the generic unless `def` is supplied, and the current function will become the default method for the generic.

If `def` is supplied, this defines the generic function, and no default method will exist (often a good feature, if the function should only be available for a meaningful subset of all objects).

Arguments `group` and `valueClass` are retained for consistency with S-Plus, but are currently not used.

**isGeneric:** If the `fdef` argument is supplied, take this as the definition of the generic, and test whether it is really a generic, with `f` as the name of the generic. (This argument is not available in S-Plus.)

**removeGeneric:** If `where` supplied, just remove the version on this element of the search list; otherwise, removes the first version encountered.

**standardGeneric:** Generic functions should usually have a call to `standardGeneric` as their entire body. They can, however, do any other computations as well.

The usual `setGeneric` (directly or through calling `setMethod`) creates a function with a call to `standardGeneric`.

**getMethods:** If the function is not a generic function, returns `NULL`. The `f` argument can be either the character string name of the generic or the object itself.

The `where` argument optionally says where to look for the function, if `f` is given as the name.

**dumpMethod:** The resulting source file will recreate the method.

**findFunction:** If `generic` is `FALSE`, ignore generic functions.

**selectMethod:** The vector of strings for the classes can be named or not. If named, the names must match formal argument names of `f`. If not named, the signature is assumed to apply to the arguments of `f` in order.

If `mustFind` is `TRUE`, an error results if there is no method (or no unique method) corresponding to this signature. Otherwise may return `NULL` or a `MethodsList` object.

**dumpMethods:** If `signature` is supplied only the methods matching this initial signature are dumped. (This feature is not found in S-Plus: don't use it if you want compatibility.)

**signature:** The advantage of using `signature` is to provide a check on which arguments you meant, as well as clearer documentation in your method specification. In addition, `signature` checks that each of the elements is a single character string.

**removeMethods:** Returns `TRUE` if `f` was a generic function, `FALSE` (silently) otherwise.

If there is a default method, the function will be re-assigned as a simple function with this definition. Otherwise, the generic function remains but with no methods (so any call to it will generate an error). In either case, a following call to `setMethod` will consistently re-establish the same generic function as before.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the `methods` package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[setGeneric](#), [setClass](#), [showMethods](#)

## Examples

```
## Not run:
## get the function "myFun" -- throw an error if 0 or > 1 versions visible
allF <- findFunction("myFun")
```

```

if(length(allF) == 0)
  stop("No versions of myFun visible")
else if(length(allF) > 1)
  stop("myFun is ambiguous: ", length(allF), " versions")
else
  fdef <- get("myFun", allF[[1]])
## End(Not run)

```

---

 getClass

*Get Class Definition*


---

### Description

Get the definition of a class.

### Usage

```

getClass(Class, .Force = FALSE, where)
getClassDef(Class, where, package)

```

### Arguments

Class	the character-string name of the class.
.Force	if TRUE, return NULL if the class is undefined; otherwise, an undefined class results in an error.
where	environment from which to begin the search for the definition; by default, start at the top-level (global) environment and proceed through the search list.
package	the name of the package asserted to hold the definition. Supplied instead of <i>where</i> , with the distinction that the package need not be currently attached.

### Details

A call to `getClass` returns the complete definition of the class supplied as a string, including all slots, etc. in classes that this class extends. A call to `getClassDef` returns the definition of the class from the environment *where*, unadorned. It's usually `getClass` you want.

If you really want to know whether a class is formally defined, call `isClass`.

### Value

The object defining the class. This is an object of class "classRepEnvironment". However, *do not* deal with the contents of the object directly unless you are very sure you know what you're doing. Even then, it is nearly always better practice to use functions such as `setClass` and `setIs`. Messing up a class object will cause great confusion.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the methods package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[Classes](#), [setClass](#), [isClass](#).

## Examples

```
getClass("numeric") ## a built in class
```

---

getMethod

*Get or Test for the Definition of a Method*

---

## Description

The functions `getMethod` and `selectMethod` get the definition of a particular method; the functions `existsMethod` and `hasMethod` test for the existence of a method. In both cases the first function only gets direct definitions and the second uses inheritance. The function `findMethod` returns the package(s) in the search list (or in the packages specified by the `where` argument) that contain a method for this function and signature.

The other functions are support functions: see the details below.

## Usage

```
getMethod(f, signature=character(), where, optional=FALSE, mlist)
```

```
findMethod(f, signature, where)
```

```
getMethods(f, where)
```

```
existsMethod(f, signature = character(), where)
```

```
hasMethod(f, signature=character(), where)
```

```
selectMethod(f, signature, optional = FALSE, useInherited = TRUE,
```

```
mlist = (if (is.null(fdef)) NULL else
          getMethodsForDispatch(f, fdef)),
fdef = getGeneric(f, !optional))
```

```
MethodsListSelect(f, env, mlist, fEnv, finalDefault, evalArgs,
                  useInherited, fdef, resetAllowed)
```

## Arguments

<code>f</code>	The character-string name of the generic function.
<code>signature</code>	The signature of classes to match to the arguments of <code>f</code> . See the details below. For <code>selectMethod</code> , the signature can optionally be an environment with classes assigned to the names of the corresponding arguments. Note: the names correspond to the names of the classes, <i>not</i> to the objects supplied in a call to the generic function. (You are not likely to find this approach convenient, but it is used internally and is marginally more efficient.)
<code>where</code>	The position or environment in which to look for the method(s): by default, anywhere in the current search list.
<code>optional</code>	If the selection does not produce a unique result, an error is generated, unless this argument is <code>TRUE</code> . In that case, the value returned is either a <code>MethodsList</code> object, if more than one method matches this signature, or <code>NULL</code> if no method matches.
<code>mlist</code>	Optionally, the list of methods in which to search. By default, the function finds the methods for the corresponding generic function. To restrict the search to a particular package or environment, e.g., supply this argument as <code>getMethodsMetaData(f, where)</code> . For <code>selectMethod</code> , see the discussion of argument <code>fdef</code> .
<code>fdef</code>	In <code>selectMethod</code> , the <code>MethodsList</code> object and/or the generic function object can be explicitly supplied. (Unlikely to be used, except in the recursive call that finds matches to more than one argument.)
<code>env</code>	The environment in which argument evaluations are done in <code>MethodsListSelect</code> . Currently must be supplied, but should usually be <code>sys.frame(sys.parent())</code> when calling the function explicitly for debugging purposes.
<code>fEnv, finalDefault, evalArgs, useInherited, resetAllowed</code>	Internal-use arguments for the function's environment, the method to use as the overall default, whether to evaluate arguments, which arguments should use inheritance, and whether the cached methods are allowed to be reset.

## Details

The `signature` argument specifies classes, in an extended sense, corresponding to formal arguments of the generic function. As supplied, the argument may be a vector of strings identifying classes, and may be named or not. Names, if supplied, match the names of those formal arguments included in the signature of the generic. That signature is normally all the arguments except ...

However, generic functions can be specified with only a subset of the arguments permitted, or with the signature taking the arguments in a different order.

It's a good idea to name the arguments in the signature to avoid confusion, if you're dealing with a generic that does something special with its signature. In any case, the elements of the signature are matched to the formal signature by the same rules used in matching arguments in function calls (see [match.call](#)).

The strings in the signature may be class names, "missing" or "ANY". See [Methods](#) for the meaning of these in method selection. Arguments not supplied in the signature implicitly correspond to class "ANY"; in particular, giving an empty signature means to look for the default method.

A call to `getMethod` returns the method for a particular function and signature. As with other `get` functions, argument `where` controls where the function looks (by default anywhere in the search list) and argument `optional` controls whether the function returns `NULL` or generates an error if the method is not found. The search for the method makes no use of inheritance.

The function `selectMethod` also looks for a method given the function and signature, but makes full use of the method dispatch mechanism; i.e., inherited methods and group generics are taken into account just as they would be in dispatching a method for the corresponding signature, with the one exception that conditional inheritance is not used. Like `getMethod`, `selectMethod` returns `NULL` or generates an error if the method is not found, depending on the argument `optional`.

The functions `existsMethod` and `hasMethod` return `TRUE` or `FALSE` according to whether a method is found, the first corresponding to `getMethod` (no inheritance) and the second to `selectMethod`.

The function `getMethods` returns all the methods for a particular generic (in the form of a generic function with the methods information in its environment). The function is called from the evaluator to merge method information, and is not intended to be called directly. Note that it gets *all* the visible methods for the specified functions. If you want only the methods defined explicitly in a particular environment, use the function `getMethodsMetaData` instead.

The function `MethodsListSelect` performs a full search (including all inheritance and group generic information: see the [Methods](#) documentation page for details on how this works). The call returns a possibly revised methods list object, incorporating any method found as part of the `allMethods` slot.

Normally you won't call `MethodsListSelect` directly, but it is possible to use it for debugging purposes (only for distinctly advanced users!).

Note that the statement that `MethodsListSelect` corresponds to the selection done by the evaluator is a fact, not an assertion, in the sense that the evaluator code constructs and executes a call to `MethodsListSelect` when it does not already have a cached method for this generic function and signature. (The value returned is stored by the evaluator so that the search is not required next time.)

## Value

The call to `selectMethod` or `getMethod` returns a [MethodDefinition-class](#) object, the selected method, if a unique selection exists. (This class extends `function`, so you can use the result directly as a function if that is what you want.) Otherwise an error is thrown if `optional` is `FALSE`. If `optional` is `TRUE`, the value returned is `NULL` if no method matched, or a `MethodsList` object if multiple methods matched.

The call to `getMethods` returns the `MethodsList` object containing all the methods requested. If there are none, `NULL` is returned: `getMethods` does not generate an error in this case.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## Examples

```
setGeneric("testFun", function(x)standardGeneric("testFun"))
setMethod("testFun", "numeric", function(x)x+1)
hasMethod("testFun", "numeric")
## Not run: [1] TRUE
hasMethod("testFun", "integer") #inherited
## Not run: [1] TRUE
existsMethod("testFun", "integer")
## Not run: [1] FALSE
hasMethod("testFun") # default method
## Not run: [1] FALSE
hasMethod("testFun", "ANY")
## Not run: [1] FALSE
```

---

<code>getPackageName</code>	<i>The name associated with a given package</i>
-----------------------------	---

---

## Description

The functions below produce the package associated with a particular environment or position on the search list, or of the package containing a particular function. They are primarily used to support computations that need to differentiate objects on multiple packages.

## Usage

```
getPackageName(where)

packageSlot(object)
packageSlot(object) <- value
```

**Arguments**

where	The environment or position on the search list associated with the desired package.
object	An object providing a character string name, plus the package in which this object is to be found.
value	the name of the package.

**Details**

Package names are normally installed during loading of the package, by the `INSTALL` script or by the `library` function. (Currently, the name is stored as the object `.packageName` but don't trust this for the future.)

**Value**

`packageName` return the character-string name of the package (without the extraneous "package:" found in the search list).

`packageSlot` returns or sets the package name slot (currently an attribute, not a formal slot, but this will likely change).

**See Also**

[search](#)

**Examples**

```
## both the following usually return "base"  
getPackageName(length(search()))
```

---

hasArg

*Look for an Argument in the Call*

---

**Description**

Returns TRUE if name corresponds to an argument in the call, either a formal argument to the function, or a component of `...`, and FALSE otherwise.

**Usage**

```
hasArg(name)
```

**Arguments**

name	The unquoted name of a potential argument.
------	--

**Details**

The expression `hasArg(x)`, for example, is similar to `!missing(x)`, with two exceptions. First, `hasArg` will look for an argument named `x` in the call if `x` is not a formal argument to the calling function, but `...` is. Second, `hasArg` never generates an error if given a name as an argument, whereas `missing(x)` generates an error if `x` is not a formal argument.

**Value**

Always TRUE or FALSE as described above.

**See Also**

[missing](#)

**Examples**

```
fctest <- function(x1, ...) c(hasArg(x1), hasArg(y2))

fctest(1) ## c(TRUE, FALSE)
fctest(1, 2) ## c(TRUE, FALSE)
fctest(y2=2) ## c(FALSE, TRUE)
fctest(y=2) ## c(FALSE, FALSE) (no partial matching)
fctest(y2 = 2, x=1) ## c(TRUE, TRUE) partial match x1
```

---

initialize-methods *Methods to Initialize New Objects from a Class*

---

**Description**

The arguments to function `new` to create an object from a particular class can be interpreted specially for that class, by the definition of a method for function `initialize` for the class. This documentation describes some existing methods, and also outlines how to write new ones.

**Methods**

**.Object = "ANY"** The default method for `initialize` takes either named or unnamed arguments. Argument names must be the names of slots in this class definition, and the corresponding arguments must be valid objects for the slot (that is, have the same class as specified for the slot, or some superclass of that class). If the object comes from a superclass, it is not coerced strictly, so normally it will retain its current class (specifically, `as(object, Class, strict = FALSE)`).

Unnamed arguments must be objects of this class, of one of its superclasses, or one of its subclasses (from the class, from a class this class extends, or from a class that extends this class). If the object is from a superclass, this normally defines some of the slots in the object. If the object is from a subclass, the new object is that argument, coerced to the current class.

Unnamed arguments are processed first, in the order they appear. Then named arguments are processed. Therefore, explicit values for slots always override any values inferred from superclass or subclass arguments.

**.Object = "traceable"** Objects of a class that extends `traceable` are used to implement debug tracing (see [traceable-class](#) and [trace](#)).

The `initialize` method for these classes takes special arguments `def`, `tracer`, `exit`, `at`, `print`. The first of these is the object to use as the original definition (e.g., a function). The others correspond to the arguments to [trace](#).

**.Object = "environment"** The `initialize` method for environments takes a named list of objects to be used to initialize the environment.

**.Object = "signature"** This is a method for internal use only. It takes an optional `functionDef` argument to provide a generic function with a `signature` slot to define the argument names. See [Methods](#) for details.

### Writing Initialization Methods

Initialization methods provide a general mechanism corresponding to generator functions in other languages.

The arguments to `initialize` are `.Object` and `....`. Nearly always, `initialize` is called from `new`, not directly. The `.Object` argument is then the prototype object from the class.

Two techniques are often appropriate for `initialize` methods: special argument names and `callNextMethod`.

You may want argument names that are more natural to your users than the (default) slot names. These will be the formal arguments to your method definition, in addition to `.Object` (always) and `...` (optionally). For example, the method for class "traceable" documented above would be created by a call to `setMethod` of the form:

```
setMethod("initialize", "traceable",
  function(.Object, def, tracer, exit, at, print) ...
)
```

In this example, no other arguments are meaningful, and the resulting method will throw an error if other names are supplied.

When your new class extends another class, you may want to call the `initialize` method for this superclass (either a special method or the default). For example, suppose you want to define a method for your class, with special argument `x`, but you also want users to be able to set slots specifically. If you want `x` to override the slot information, the beginning of your method definition might look something like this:

```
function(.Object, x, ...) \{
  Object <- callNextMethod(.Object, ...)
  if(!missing(x)) \{ \# do something with x
```

You could also choose to have the inherited method override, by first interpreting `x`, and then calling the next method.

is

*Is an Object from a Class***Description**

`is`: With two arguments, tests whether `object` can be treated as from `class2`.

With one argument, returns all the super-classes of this object's class.

`extends`: Does the first class extend the second class? Returns maybe if the extension includes a test.

`setIs`: Defines `class1` to be an extension of `class2`.

**Usage**

```
is(object, class2)
```

```
extends(class1, class2, maybe=TRUE, fullInfo = FALSE)
```

```
setIs(class1, class2, test=NULL, coerce=NULL, replace=NULL,
      by = character(), where = toplev(parent.frame()), classDef =,
      extensionObject = NULL, doComplete = TRUE)
```

**Arguments**

`object` any R object.

`class1, class2`

the names of the classes between which `is` relations are to be defined.

`maybe, fullInfo`

In a call to `extends`, `maybe` is a flag to include/exclude conditional relations, and `fullInfo` is a flag, which if `TRUE` causes `object(s)` of class `classExtension` to be returned, rather than just the names of the classes or a logical value. See the details below.

`extensionObject`

alternative to the `test`, `coerce`, `replace`, `by` arguments; an object from class `SClassExtension` describing the relation. (Used in internal calls.)

`doComplete` when `TRUE`, the class definitions will be augmented with indirect relations as well. (Used in internal calls.)

`test, coerce, replace`

In a call to `setIs`, functions optionally supplied to test whether the relation is defined, to coerce the object to `class2`, and to alter the object so that `is(object, class2)` is identical to `value`.

`by`

In a call to `setIs`, the name of an intermediary class. Coercion will proceed by first coercing to this class and from there to the target class. (The intermediate coercions have to be valid.)

where	In a call to <code>setIs</code> , where to store the metadata defining the relationship. Default is the global environment.
classDef	Optional class definition for <code>class</code> , required internally when <code>setIs</code> is called during the initial definition of the class by a call to <code>setClass</code> . <i>Don't</i> use this argument, unless you really know why you're doing so.

## Details

**extends:** Given two class names, `extends` by default says whether the first class extends the second; that is, it does for class names what `is` does for an object and a class. Given one class name, it returns all the classes that class extends (the “superclasses” of that class), including the class itself. If the flag `fullInfo` is `TRUE`, the result is a list, each element of which is an object describing the relationship; otherwise, and by default, the value returned is only the names of the classes.

**setIs:** This function establishes an inheritance relation between two classes, by some means other than having one class contain the other. It should *not* be used for ordinary relationships: either include the second class in the `contains=` argument to `setClass` if the class is contained in the usual way, or consider `setClassUnion` to define a virtual class that is extended by several ordinary classes. A call to `setIs` makes sense, for example, if one class ought to be automatically convertible into a second class, but they have different representations, so that the conversion must be done by an explicit computation, not just be inheriting slots, for example. In this case, you will typically need to provide both a `coerce=` and `replace=` argument to `setIs`.

The `coerce`, `replace`, and `by` arguments behave as described for the `setAs` function. It's unlikely you would use the `by` argument directly, but it is used in defining cached information about classes. The value returned (invisibly) by `setIs` is the extension information, as a list.

The `coerce` argument is a function that turns a `class1` object into a `class2` object. The `replace` argument is a function of two arguments that modifies a `class1` object (the first argument) to replace the part of it that corresponds to `class2` (supplied as `value`, the second argument). It then returns the modified object as the value of the call. In other words, it acts as a replacement method to implement the expression `as(object, class2) <- value`.

The easiest way to think of the `coerce` and `replace` functions is by thinking of the case that `class1` contains `class2` in the usual sense, by including the slots of the second class. (To repeat, in this situation you would not call `setIs`, but the analogy shows what happens when you do.)

The `coerce` function in this case would just make a `class2` object by extracting the corresponding slots from the `class1` object. The `replace` function would replace in the `class1` object the slots corresponding to `class2`, and return the modified object as its value.

The relationship can also be conditional, if a function is supplied as the `test` argument. This should be a function of one argument that returns `TRUE` or `FALSE` according to whether the object supplied satisfies the relation `is(object, class2)`. If you worry about such things, conditional relations between classes are slightly deprecated because they cannot be implemented as efficiently as ordinary relations and because they sometimes can lead to confusion (in thinking about what methods are dispatched for a particular function, for example). But they can correspond to useful distinctions, such as when two classes have the same representation, but only one of them obeys certain additional constraints.

Because only global environment information is saved, it rarely makes sense to give a value other than the default for argument `where`. One exception is `where=0`, which modifies the cached (i.e., session-scope) information about the class. Class completion computations use this version, but don't use it yourself unless you are quite sure you know what you're doing.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## Examples

```
## a class definition (see setClass for the example)
setClass("trackCurve",
  representation("track", smooth = "numeric"))
## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
  representation(x="numeric", y="matrix", smooth="matrix"),
  prototype = structure(list(), x=numeric(), y=matrix(0,0,0),
                        smooth= matrix(0,0,0)))
## Automatically convert an object from class "trackCurve" into
## "trackMultiCurve", by making the y, smooth slots into 1-column matrices
setIs("trackCurve",
  "trackMultiCurve",
  coerce = function(obj) {
    new("trackMultiCurve",
      x = obj@x,
      y = as.matrix(obj@y),
      curve = as.matrix(obj@smooth))
  },
  replace = function(obj, value) {
    obj@y <- as.matrix(value@y)
    obj@x <- value@x
    obj@smooth <- as.matrix(value@smooth)
    obj})

## Automatically convert the other way, but ONLY
## if the y data is one variable.
setIs("trackMultiCurve",
  "trackCurve",
  test = function(obj) {ncol(obj@y) == 1},
```

```

coerce = function(obj) {
  new("trackCurve",
      x = slot(obj, "x"),
      y = as.numeric(obj@y),
      smooth = as.numeric(obj@smooth))
},
replace = function(obj, value) {
  obj@y <- matrix(value@y, ncol=1)
  obj@x <- value@x
  obj@smooth <- value@smooth
  obj})

```

---

isSealedMethod      *Check for a Sealed Method or Class*

---

### Description

These functions check for either a method or a class that has been “sealed” when it was defined, and which therefore cannot be re-defined.

### Usage

```

isSealedMethod(f, signature, fdef, where)
isSealedClass(Class, where)

```

### Arguments

f	The quoted name of the generic function.
signature	The class names in the method’s signature, as they would be supplied to <a href="#">setMethod</a> .
fdef	Optional, and usually omitted: the generic function definition for f.
Class	The quoted name of the class.
where	where to search for the method or class definition. By default, searches from the top environment of the call to <code>isSealedMethod</code> or <code>isSealedClass</code> , typically the global environment or the namespace of a package containing a call to one of the functions.

### Details

In the R implementation of classes and methods, it is possible to seal the definition of either a class or a method. The basic classes (numeric and other types of vectors, matrix and array data) are sealed. So also are the methods for the primitive functions on those data types. The effect is that programmers cannot re-define the meaning of these basic data types and computations. More precisely, for primitive functions that depend on only one data argument, methods cannot be specified for basic classes. For functions (such as the arithmetic operators) that depend on two arguments, methods can be specified if *one* of those arguments is a basic class, but not if both are.

Programmers can seal other class and method definitions by using the `sealed` argument to [setClass](#) or [setMethod](#).

**Value**

The functions return `FALSE` if the method or class is not sealed (including the case that it is not defined); `TRUE` if it is.

**References**

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

**Examples**

```
## these are both TRUE
isSealedMethod("+", c("numeric", "character"))
isSealedClass("matrix")

setClass("track",
         representation(x="numeric", y="numeric"))
## but this is FALSE
isSealedClass("track")
## and so is this
isSealedClass("A Name for an undefined Class")
## and so are these, because only one of the two arguments is basic
isSealedMethod("+", c("track", "numeric"))
isSealedMethod("+", c("numeric", "track"))
```

---

 language-class

---

*Classes to Represent Unevaluated Language Objects*


---

**Description**

The virtual class "language" and the specific classes that extend it represent unevaluated objects, as produced for example by the parser or by functions such as `quote`.

**Usage**

```
### each of these classes corresponds to an unevaluated object
### in the S language. The class name can appear in method signatures,
### and in a few other contexts (such as some calls to as()).
```

```

" ( "
" <- "
"call"
"for"
"if"
"repeat"
"while"
"name"
" { "

### Each of the classes above extends the virtual class

"language"

```

### Objects from the Class

"language" is a virtual class; no objects may be created from it.

Objects from the other classes can be generated by a call to `new(Class, ...)`, where `Class` is the quoted class name, and the `...` arguments are either empty or a *single* object that is from this class (or an extension).

### Methods

**coerce** signature(`from = "ANY"`, `to = "call"`). A method exists for `as(object, "call")`, calling `as.call()`.

---

```

LinearMethodsList-class
      Class "LinearMethodsList"

```

---

### Description

A version of methods lists that has been “linearized” for producing summary information. The actual objects from class "MethodsList" used for method dispatch are defined recursively over the arguments involved.

### Objects from the Class

The function `linearizeMlist` converts an ordinary methods list object into the linearized form.

### Slots

**methods:** Object of class "list", the method definitions.

**arguments:** Object of class "list", the corresponding formal arguments.

**classes:** Object of class "list", the corresponding classes in the signatures.

**fromClasses:** Object of class "list"

**Future Note**

The current version of `linearizeMlist` does not take advantage of the `MethodDefinition` class, and therefore does more work for less effect than it could. In particular, we may move to redefine both the function and the class to take advantage of the stored signatures. Don't write code depending precisely on the present form, although all the current information will be obtainable in the future.

**See Also**

Function `linearizeMlist` for the computation, and `MethodsList-class` for the original, recursive form.

---

makeClassRepresentation  
*Create a Class Definition*

---

**Description**

Constructs a `classRepresentation-class` object to describe a particular class. Mostly a utility function, but you can call it to create a class definition without assigning it, as `setClass` would do.

**Usage**

```
makeClassRepresentation(name, slots=list(), superClasses=character(),
                        prototype=NULL, package, validity, access,
                        version, sealed, virtual=NA, where)
```

**Arguments**

<code>name</code>	character string name for the class
<code>slots</code>	named list of slot classes as would be supplied to <code>setClass</code> , but <i>without</i> the unnamed arguments for <code>superClasses</code> if any.
<code>superClasses</code>	what classes does this class extend
<code>prototype</code>	an object providing the default data for the class, e.g, the result of a call to <code>prototype</code> .
<code>package</code>	The character string name for the package in which the class will be stored; see <code>getPackageName</code> .
<code>validity</code>	Optional validity method. See <code>validObject</code> , and the discussion of validity methods in the reference.
<code>access</code>	Access information. Not currently used.
<code>version</code>	Optional version key for version control. Currently generated, but not used.
<code>sealed</code>	Is the class sealed? See <code>setClass</code> .
<code>virtual</code>	Is this known to be a virtual class?
<code>where</code>	The environment from which to look for class definitions needed (e.g., for slots or superclasses). See the discussion of this argument under <code>GenericFunctions</code> .

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[setClass](#)

---

MethodDefinition-class

*Classes to Represent Method Definitions*

---

## Description

These classes extend the basic class "function" when functions are to be stored and used as method definitions.

## Details

Method definition objects are functions with additional information defining how the function is being used as a method. The `target` slot is the class signature for which the method will be dispatched, and the `defined` slot the signature for which the method was originally specified (that is, the one that appeared in some call to `setMethod`).

## Objects from the Class

The action of setting a method by a call to `setMethod` creates an object of this class. It's unwise to create them directly.

The class "SealedMethodDefinition" is created by a call to `setMethod` with argument `sealed = TRUE`. It has the same representation as "MethodDefinition".

## Slots

**.Data:** Object of class "function"; the data part of the definition.

**target:** Object of class "signature"; the signature for which the method was wanted.

**defined:** Object of class "signature"; the signature for which a method was found. If the method was inherited, this will not be identical to `target`.

**Extends**

Class "function", from data part.  
 Class "PossibleMethod", directly.  
 Class "OptionalMethods", by class "function".

**See Also**

class [MethodsList-class](#) for the objects defining sets of methods associated with a particular generic function. The individual method definitions stored in these objects are from class [MethodDefinition](#), or an extension. [MethodWithNext-class](#) for an extension used by [callNextMethod](#).

---

 Methods

---

*General Information on Methods*


---

**Description**

This documentation section covers some general topics on how methods work and how the **methods** package interacts with the rest of R. The information is usually not needed to get started with methods and classes, but may be helpful for moderately ambitious projects, or when something doesn't work as expected.

The section **How Methods Work** describes the underlying mechanism; **Class Inheritance and Method Selection** provides more details on how class definitions determine which methods are used.

The section **Changes with the Methods Package** outlines possible effects on other computations when running with package **methods**.

**How Methods Work**

A generic function is a function that has associated with it a collection of other functions (the methods), all of which agree in formal arguments with the generic. In R, the "collection" is an object of class "[MethodsList](#)", which contains a named list of methods (the `methods` slot), and the name of one of the formal arguments to the function (the `argument` slot). The names of the methods are the names of classes, and the corresponding element defines the method or methods to be used if the corresponding argument has that class. For example, suppose a function `f` has formal arguments `x` and `y`. The methods list object for that function has the object `as.name("x")` as its `argument` slot. An element of the methods named `"track"` is selected if the actual argument corresponding to `x` is an object of class `"track"`. If there is such an element, it can generally be either a function or another methods list object.

In the first case, the function defines the method to use for any call in which `x` is of class `"track"`. In the second case, the new methods list object defines the selection of methods depending on the remaining formal arguments, in this example, `y`. The same selection process takes place, recursively, using the new methods list. Eventually, the selection returns either a function or `NULL`, meaning that no method matched the actual arguments.

Each method selected corresponds conceptually to a *signature*; that is a named list of classes, with names corresponding to some or all of the formal arguments. In the previous example, if selecting

class "track" for x, finding that the selection was another methods list and then selecting class "numeric" for y would produce a method associated with the signature `x = "track", y = "numeric"`.

The actual selection is done recursively, but you can see the methods arranged by signature by calling the function `showMethods`, and objects with the methods arranged this way (in two different forms) are returned by the functions `listFromMlist` and `linearizeMlist`.

In an R session, each generic function has a single methods list object defining all the currently available methods. The session methods list object is created the first time the function is called by merging all the relevant method definitions currently visible. Whenever something happens that might change the definitions (such as attaching or detaching a package with methods for this function, or explicitly defining or removing methods), the merged methods list object is removed. The next call to the function will recompute the merged definitions.

When methods list are merged, they can come from two sources:

1. Methods list objects for the same function anywhere on the current search list. These are merged so that methods in an environment earlier in the search list override methods for the same function later in the search list. A method overrides only another method for the same signature. See the comments on class "ANY" in the section on **Inheritance**.
2. Methods list objects corresponding the group generic functions, if any, for this function. Any generic function can be defined to belong to a group generic. The methods for the group generic are available as methods for this function. The group generic can itself be defined as belong to a group; as a result there is a list of group generic functions. A method defined for a function and a particular signature overrides a method for the same signature for that function's group generic.

Merging is done first on all methods for a particular function, and then over the generic and its group generics.

The result is a single methods list object that contains all the methods *directly* defined for this function. As calls to the function occur, this information may be supplemented by *inherited* methods, which we consider next.

### Class Inheritance and Method Selection

If no method is found directly for the actual arguments in a call to a generic function, an attempt is made to match the available methods to the arguments by using *inheritance*.

Each class definition potentially includes the names of one or more classes that the new class contains. (These are sometimes called the *superclasses* of the new class.) These classes themselves may extend other classes. Putting all this information together produces the full list of superclasses for this class. (You can see this list for any class "A" from the expression `extends("A")`.) In addition, any class implicitly extends class "ANY". When all the superclasses are needed, as they are for dispatching methods, they are ordered by how direct they are: first, the direct classes contained directly in the definition of this class, then the superclasses of these classes, etc.

The S language has an additional, explicit mechanism for defining superclasses, the `setIs` mechanism. This mechanism allows a class to extend another even though they do not have the same representation. The extension is made possible by defining explicit methods to `coerce` an object to its superclass and to `replace` the data in the object corresponding to the superclass. The `setIs` mechanism will be used less often and only when directly including the superclass does not

make sense, but once defined, the superclass acts just as directly contained classes as far as method selection is concerned.

A method will be selected by inheritance if we can find a method in the methods list for a signature corresponding to any combination of superclasses for each of the relevant arguments. The search for such a method is performed by the function `MethodsListSelect`, working as follows.

The generic, `f` say, has a signature, which by default is all its formal arguments, except ... (see `setGeneric`). For each of the formal arguments in that signature, in order, the class of the actual argument is matched against available methods. A missing argument corresponds to class "missing". If no method corresponds to the class of the argument, the evaluator looks for a method corresponding to the the superclasses (the other classes that the actual class extends, always including "ANY"). If no match is found, the dispatch fails, with an error. (But if there is a default method, that will always match.)

If the match succeeds, it can find either a single method, or a methods list. In the first case, the search is over, and returns the method. In the second case, the search proceeds, with the next argument in the signature of the generic. *That* search may succeed or fail. If it fails, the dispatch will try again with the next best match for the current argument, if there is one. The last match always corresponds to class "ANY".

The effect of this definition of the selection process is to order all possible inherited methods, first by the superclasses for the first argument, then within this by the superclasses for the second argument, and so on.

### Changes with the Methods Package

The **methods** package is designed to leave other computations in R unchanged. There are, however, a few areas where the default functions and behavior are overridden when running with the methods package attached. This section outlines those known to have some possible effect.

**class:** The **methods** package enforces the notion that every object has a class; in particular, `class(x)` is never NULL, as it would be for basic vectors, for example, when not using **methods**.

In addition, when assigning a class, the value is required to be a single string. (However, objects can have multiple class names if these were generated by old-style class computations. The methods package does not hide the "extra" class names.)

Computations using the notion of NULL class attributes or of class attributes with multiple class names are not really compatible with the ideas in the **methods** package. Formal classes and class inheritance are designed to give more flexible and reliable implementations of similar ideas.

If you do have to mix the two approaches, any operations that use class attributes in the old sense should be written in terms of `attr(x, "class")`, not `class(x)`. In particular, test for no class having been assigned with `is.null(attr(x, "class"))`.

**Printing:** To provide appropriate printing automatically for objects with formal class definitions, the **methods** package overrides `print.default`, to look for methods for the generic function `show`, and to use a default method for objects with formal class definitions.

The revised version of `print.default` is intended to produce identical printing to the original version for any object that does *not* have a formally defined class, including honoring old-style print methods. So far, no exceptions are known.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[setGeneric](#), [setClass](#)

---

MethodsList-class    *Class MethodsList, Representation of Methods for a Generic Function*

---

## Description

Objects from this class are generated and revised by the definition of methods for a generic function.

## Slots

**argument:** Object of class "name". The name of the argument being used for dispatch at this level.

**methods:** A named list of the methods (and method lists) defined *explicitly* for this argument, with the names being the classes for which the methods have been defined.

**allMethods:** A named list, which may be empty if this object has not been used in dispatch yet. Otherwise, it contains all the directly defined methods from the `methods` slot, plus any inherited methods.

## Extends

Class "OptionalMethods", directly.

---

MethodWithNext-class

*Class MethodWithNext*

---

### Description

Class of method definitions set up for `callNextMethod`

### Objects from the Class

Objects from this class are generated as a side-effect of calls to `callNextMethod`.

### Slots

**.Data:** Object of class "function"; the actual function definition.

**nextMethod:** Object of class "PossibleMethod" the method to use in response to a `callNextMethod()` call.

**excluded:** Object of class "list"; one or more signatures excluded in finding the next method.

**target:** Object of class "signature", from class "MethodDefinition"

**defined:** Object of class "signature", from class "MethodDefinition"

### Extends

Class "MethodDefinition", directly.

Class "function", from data part.

Class "PossibleMethod", by class "MethodDefinition".

Class "OptionalMethods", by class "MethodDefinition".

### Methods

**findNextMethod** signature(method = "MethodWithNext"): used internally by method dispatch.

**loadMethod** signature(method = "MethodWithNext"): used internally by method dispatch.

**show** signature(object = "MethodWithNext")

### See Also

`callNextMethod`, and `MethodDefinition-class`.

---

`new`*Generate an Object from a Class*

---

## Description

Given the name or the definition of a class, plus optionally data to be included in the object, `new` returns an object from that class.

## Usage

```
new(Class, ...)
```

```
initialize(.Object, ...)
```

## Arguments

<code>Class</code>	Either the name of a class (the usual case) or the object describing the class (e.g., the value returned by <code>getClass</code> ).
<code>...</code>	Data to include in the new object. Named arguments correspond to slots in the class definition. Unnamed arguments must be objects from classes that this class extends.
<code>.Object</code>	An object: see the Details section.

## Details

The function `new` begins by copying the prototype object from the class definition. Then information is inserted according to the `...` arguments, if any.

The interpretation of the `...` arguments can be specialized to particular classes, if an appropriate method has been defined for the generic function `"initialize"`. The `new` function calls `initialize` with the object generated from the prototype as the `.Object` argument to `initialize`.

By default, unnamed arguments in the `...` are interpreted as objects from a superclass, and named arguments are interpreted as objects to be assigned into the correspondingly named slots. Thus, explicit slots override inherited information for the same slot, regardless of the order in which the arguments appear.

The `initialize` methods do not have to have `...` as their second argument (see the examples), and generally it is better design *not* to have `...` as a formal argument, if only a fixed set of arguments make sense.

For examples of `initialize` methods, see [initialize-methods](#) for existing methods for classes `"traceable"` and `"environment"`, among others.

Note that the basic vector classes, `"numeric"`, etc. are implicitly defined, so one can use `new` for these classes.

## References

The web page <http://www.omegahat.org/RMethods/index.html> is the primary documentation.

The functions in this package emulate the facility for classes and methods described in *Programming with Data* (John M. Chambers, Springer, 1998). See this book for further details and examples.

## See Also

[Classes](#)

## Examples

```
## using the definition of class "track" from Classes

## a new object with two slots specified
t1 <- new("track", x = seq(along=ydata), y = ydata)

# a new object including an object from a superclass, plus a slot
t2 <- new("trackCurve", t1, smooth = ysmooth)

### define a method for initialize, to ensure that new objects have
### equal-length x and y slots.

setMethod("initialize",
          "track",
          function(.Object, x = numeric(0), y = numeric(0)) {
            if(nargs() > 1) {
              if(length(x) != length(y))
                stop("specified x and y of different lengths")
              .Object@x <- x
              .Object@y <- y
            }
            .Object
          })

### the next example will cause an error (x will be numeric(0)),
### because we didn't build in defaults for x,
### although we could with a more elaborate method for initialize

try(new("track", y = sort(rnorm(10))))

## a better way to implement the previous initialize method.
## Why? By using callNextMethod to call the default initialize method
## we don't inhibit classes that extend "track" from using the general
## form of the new() function. In the previous version, they could only
## use x and y as arguments to new, unless they wrote their own
## initialize method.

setMethod("initialize", "track", function(.Object, ...) {
```

```

.Object <- callNextMethod()
if(length(.Object@x) != length(.Object@y))
  stop("specified x and y of different lengths")
.Object
})

```

---

ObjectsWithPackage-class

*A Vector of Object Names, with associated Package Names*

---

### Description

This class of objects is used to represent ordinary character string object names, extended with a package slot naming the package associated with each object.

### Objects from the Class

The function `getGenerics` returns an object of this class.

### Slots

**.Data:** Object of class "character": the object names.

**package:** Object of class "character" the package names.

### Extends

Class "character", from data part.

Class "vector", by class "character".

### See Also

Methods for general background.

---

promptClass

*Generate a Shell for Documentation of a Formal Class*

---

### Description

Assembles all relevant slot and method information for a class, with minimal markup for Rd processing; no QC facilities at present.

### Usage

```

promptClass(clName, filename = NULL, type = "class",
            keywords = "classes", where = topenv(parent.frame()))

```

## Arguments

<code>className</code>	a character string naming the class to be documented.
<code>filename</code>	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is the topic name for the class documentation, followed by ".Rd". Can also be NA (see below).
<code>type</code>	the documentation type to be declared in the output file.
<code>keywords</code>	the keywords to include in the shell of the documentation. The keyword "classes" should be one of them.
<code>where</code>	where to look for the definition of the class and of methods that use it.

## Details

The class definition is found on the search list. Using that definition, information about classes extended and slots is determined.

In addition, the currently available generics with methods for this class are found (using `getGenerics`). Note that these methods need not be in the same environment as the class definition; in particular, this part of the output may depend on which packages are currently in the search list.

As with other prompt-style functions, unless `filename` is NA, the documentation shell is written to a file, and a message about this is given. The file will need editing to give information about the *meaning* of the class. The output of `promptClass` can only contain information from the metadata about the formal definition and how it is used.

If `filename` is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

## Value

If `filename` is NA, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

## Author(s)

VJ Carey <stvjc@channing.harvard.edu> and John Chambers

## References

The web page <http://www.omegahat.org/RMethods/index.html> is the primary documentation.

The functions in this package emulate the facility for classes and methods described in *Programming with Data* (John M. Chambers, Springer, 1998). See this book for further details and examples.

**See Also**

[prompt](#) for documentation of functions, [promptMethods](#) for documentation of method definitions.

For processing of the edited documentation, either use R CMD [Rdconv](#), or include the edited file in the ‘man’ subdirectory of a package.

**Examples**

```
## Not run:
> promptClass("track")
A shell of class documentation has been written to the
file "track-class.Rd".
## End(Not run)
```

---

promptMethods

*Generate a Shell for Documentation of Formal Methods*

---

**Description**

Generates a shell of documentation for the methods of a generic function.

**Usage**

```
promptMethods(f, filename = NULL, methods)
```

**Arguments**

<code>f</code>	a character string naming the generic function whose methods are to be documented.
<code>filename</code>	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to the coded topic name for these methods (currently, <code>f</code> followed by <code>"-methods.Rd"</code> ). Can also be <code>FALSE</code> or <code>NA</code> (see below).
<code>methods</code>	Optional methods list object giving the methods to be documented. By default, the first methods object for this generic is used (for example, if the current global environment has some methods for <code>f</code> , these would be documented). If this argument is supplied, it is likely to be <code>getMethods(f, where)</code> , with <code>where</code> some package containing methods for <code>f</code> .

**Details**

If `filename` is `FALSE`, the text created is returned, presumably to be inserted some other documentation file, such as the documentation of the generic function itself (see [prompt](#)).

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

Otherwise, the documentation shell is written to the file specified by `filename`.

**Value**

If `filename` is `FALSE`, the text generated; if `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

**References**

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

**See Also**

[prompt](#) and [promptClass](#)

---

representation

*Construct a Representation or a Prototype for a Class Definition*

---

**Description**

In calls to `setClass`, these two functions construct, respectively, the `representation` and `prototype` arguments. They do various checks and handle special cases. You're encouraged to use them when defining classes that, for example, extend other classes as a data part or have multiple superclasses, or that combine extending a class and slots.

**Usage**

```
representation(...)  
prototype(...)
```

**Arguments**

...

The call to `representation` takes arguments that are single character strings. Unnamed arguments are classes that a newly defined class extends; named arguments name the explicit slots in the new class, and specify what class each slot should have.

In the call to `prototype`, if an unnamed argument is supplied, it unconditionally forms the basis for the prototype object. Remaining arguments are taken to correspond to slots of this object. It is an error to supply more than one unnamed argument.

## Details

The `representation` function applies tests for the validity of the arguments. Each must specify the name of a class.

The classes named don't have to exist when `representation` is called, but if they do, then the function will check for any duplicate slot names introduced by each of the inherited classes.

The arguments to `prototype` are usually named initial values for slots, plus an optional first argument that gives the object itself. The unnamed argument is typically useful if there is a data part to the definition (see the examples below).

## Value

The value of `representation` is just the list of arguments, after these have been checked for validity.

The value of `prototype` is the object to be used as the prototype. Slots will have been set consistently with the arguments, but the construction does *not* use the class definition to test validity of the contents (it hardly can, since the prototype object is usually supplied to create the definition).

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[setClass](#)

## Examples

```
## representation for a new class with a directly define slot "smooth"
## which should be a "numeric" object, and extending class "track"
representation("track", smooth = "numeric")

setClass("Character", representation("character"))
setClass("TypedCharacter", representation("Character", type="character"),
         prototype(character(0), type="plain"))
ttt <- new("TypedCharacter", "foo", type = "character")

setClass("num1", representation(comment = "character"),
         contains = "numeric",
         prototype = prototype(pi, comment = "Start with pi"))
```

---

 SClassExtension-class

*Class to Represent Inheritance (Extension) Relations*


---

### Description

An object from this class represents a single “is” relationship; lists of these objects are used to represent all the extensions (superclasses) and subclasses for a given class. The object contains information about how the relation is defined and methods to coerce, test, and replace correspondingly.

### Objects from the Class

Objects from this class are generated by `setIs`, both from direct calls .

### Slots

**subClass, superClass:** The classes being extended: corresponding to the `from`, and `to` arguments to `setIs`.

**package:** The package to which that class belongs.

**coerce:** A function to carry out the `as()` computation implied by the relation. Note that these functions should *not* be used directly. They only deal with the `strict=TRUE` calls to the `as` function, with the full method constructed from this mechanically.

**test:** The function that would test whether the relation holds. Except for explicitly specified `test` arguments to `setIs`, this function is trivial.

**replace:** The method used to implement `as(x, Class) <- value`.

**simple:** A "logical" flag, TRUE if this is a simple relation, either because one class is contained in the definition of another, or because a class has been explicitly stated to extend a virtual class. For simple extensions, the three methods are generated automatically.

**by:** If this relation has been constructed transitively, the first intermediate class from the subclass.

**dataPart:** A "logical" flag, TRUE if the extended class is in fact the data part of the subclass. In this case the extended class is a basic class (i.e., a type).

### Methods

No methods defined with class "SClassExtension" in the signature.

### See Also

`is`, `as`, and `classRepresentation-class`.

---

setClass

*Create a Class Definition*


---

### Description

Functions to create (`setClass`) and manipulate class definitions.

### Usage

```
setClass(Class, representation, prototype, contains=character(),
         validity, access, where, version, sealed, package)
```

```
removeClass(Class, where)
```

```
isClass(Class, formal=TRUE, where)
```

```
getClasses(where, inherits = missing(where))
```

```
findClass(Class, where, unique = "")
```

```
resetClass(Class, classDef, where)
```

```
sealClass(Class, where)
```

### Arguments

`Class` character string name for the class. Other than `setClass`, the functions will usually take a class definition instead of the string (allowing the caller to identify the class uniquely).

`representation` the slots that the new class should have and/or other classes that this class extends. Usually a call to the [representation](#) function.

`prototype` an object (usually a list) providing the default data for the slots specified in the representation.

`contains` what classes does this class extend? (These are called *superclasses* in some languages.) When these classes have slots, all their slots will be contained in the new class as well.

`where` For `setClass` and `removeClass`, the environment in which to store or remove the definition. Defaults to the top-level environment of the calling function (the global environment for ordinary computations, but the environment or namespace of a package when loading that package).

For other functions, `where` defines where to do the search for the class definition, and the default is to search from the top-level environment or namespace of the caller to this function.

<code>unique</code>	if <code>findClass</code> expects a unique location for the class, <code>unique</code> is a character string explaining the purpose of the search (and is used in warning and error messages). By default, multiple locations are possible and the function always returns a list.
<code>inherits</code>	in a call to <code>getClasses</code> , should the value returned include all parent environments of <code>where</code> , or that environment only? Defaults to <code>TRUE</code> if <code>where</code> is omitted, and to <code>FALSE</code> otherwise.
<code>validity</code>	if supplied, should be a validity-checking method for objects from this class (a function that returns <code>TRUE</code> if its argument is a valid object of this class and one or more strings describing the failures otherwise). See <code>validObject</code> for details.
<code>access</code>	Access list for the class. Saved in the definition, but not currently used.
<code>version</code>	A version indicator for this definition. Saved in the definition, but not currently used.
<code>sealed</code>	If <code>TRUE</code> , the class definition will be sealed, so that another call to <code>setClass</code> will fail on this class name.
<code>package</code>	An optional package name for the class. By default (and usually) the package where the class definition is assigned will be used.
<code>formal</code>	Should a formal definition be required?
<code>classDef</code>	For <code>removeClass</code> , the optional class definition (but usually it's better for <code>Class</code> to be the class definition, and to omit <code>classDef</code> ).

## Details

These are the functions that create and manipulate formal class definitions. Brief documentation is provided below. See the references for an introduction and for more details.

**setClass:** Define `Class` to be an S-style class. The effect is to create an object, of class "`classRepEnvironment`", and store this (hidden) in the specified environment or database. Objects can be created from the class (e.g., by calling `new`), manipulated (e.g., by accessing the object's slots), and methods may be defined including the class name in the signature (see `setMethod`).

**removeClass:** Remove the definition of this class, from the environment `where` if this argument is supplied; if not, `removeClass` will search for a definition, starting in the top-level environment of the call to `removeClass`, and remove the (first) definition found.

**isClass:** Is this a the name of a formally defined class? (Argument `formal` is for compatibility and is ignored.)

**getClasses:** The names of all the classes formally defined on `where`. If called with no argument, all the classes visible from the calling function (if called from the top-level, all the classes in any of the environments on the search list). The `inherits` argument can be used to search a particular environment and all its parents, but usually the default setting is what you want.

**findClass:** The list of environments or positions on the search list in which a class definition of `Class` is found. If `where` is supplied, this is an environment (or namespace) from which the search takes place; otherwise the top-level environment of the caller is used. If `unique`

is supplied as a character string, `findClass` returns a single environment or position. By default, it always returns a list. The calling function should select, say, the first element as a position or environment for functions such as `get`.

If `unique` is supplied as a character string, `findClass` will warn if there is more than one definition visible (using the string to identify the purpose of the call), and will generate an error if no definition can be found.

**resetClass:** Reset the internal definition of a class. Causes the complete definition of the class to be re-computed, from the representation and superclasses specified in the original call to `setClass`.

This function is called when aspects of the class definition are changed. You would need to call it explicitly if you changed the definition of a class that this class extends (but doing that in the middle of a session is living dangerously, since it may invalidate existing objects).

**sealClass:** Seal the current definition of the specified class, to prevent further changes. It is possible to seal a class in the call to `setClass`, but sometimes further changes have to be made (e.g., by calls to `setIs`). If so, call `sealClass` after all the relevant changes have been made.

## Inheritance and Prototypes

Defining new classes that inherit from (“extend”) other classes is a powerful technique, but has to be used carefully and not over-used. Otherwise, you will often get unintended results when you start to compute with objects from the new class.

As shown in the examples below, the simplest and safest form of inheritance is to start with an explicit class, with some slots, that does not extend anything else. It only does what we say it does.

Then extensions will add some new slots and new behavior.

Another variety of extension starts with one of the built-in data types, perhaps with the intension of modifying R’s standard behavior for that class. In this case, the new class inherits the built-in data type as its “data” part. See the “numWithId” example below.

When such a class definition is printed, the data part shows up as a pseudo-slot named “.Data”.

## S3 Classes

Earlier, informal classes of objects (usually referred to as “S3” classes) are used by many R functions. It’s natural to consider including them as the class for a slot in a formal class, or even as a class to be extended by the new class. This isn’t prohibited but there are some disadvantages, and if you do want to include S3 classes, they should be declared by including them in a call to `setOldClass`. Here are some considerations:

- Using S3 classes somewhat defeats the purpose of defining a formal class: An important advantage to your users is that a formal class provides guarantees of what the object contains (minimally, the classes of the slots and therefore what data they contain; optionally, any other requirements imposed by a validity method).

But there is no guarantee whatever about the data in an object from an S3 class. It’s entirely up to the functions that create or modify such objects. If you want to provide guarantees to your users, you will need a validity method that explicitly checks the contents of S3-class objects.

- To get the minimal guarantee (that the object in a slot has, or extends, the class for the slot) you should ensure that the S3 classes are known to *be* S3 classes, with the possible inheritance. To do this, include a call to `setOldClass` for the S3 classes used. Otherwise, the S3 class is undefined (and the code used by `setClass` will issue a warning). Slot assignments, for example, will not then check for possible errors.
- These caveats apply to S3 classes; that is, objects with a class assigned by some R function but without a formal class definition. In contrast, the built-in data types (`numeric`, `list`, etc.) are generally fine as slots or for `contains=` classes (see the previous section). These data types don't have formal slots, but the base code in the system essentially forces them to contain the type of data they claim to have.

The data types `matrix` and `array` are somewhat in between. They do not have an explicit S3 class, but do have one or two attributes. There is no general problem in having these as slots, but because there is no guarantee of a `dimnames` slot, they don't work as formal classes. The `ts` class is treated as a formal class, extending class `vector`.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[setClassUnion](#), [Methods](#), [makeClassRepresentation](#)

## Examples

```
## A simple class with two slots
setClass("track",
        representation(x="numeric", y="numeric"))
## A class extending the previous, adding one more slot
setClass("trackCurve",
        representation("track", smooth = "numeric"))
## A class similar to "trackCurve", but with different structure
## allowing matrices for the "y" and "smooth" slots
setClass("trackMultiCurve",
        representation(x="numeric", y="matrix", smooth="matrix"),
        prototype = list(x=numeric(), y=matrix(0,0,0),
                        smooth= matrix(0,0,0)))
##
## Suppose we want trackMultiCurve to be like trackCurve when there's
## only one column.
## First, the wrong way.
```

```

try(setIs("trackMultiCurve", "trackCurve",
  test = function(obj) {ncol(slot(obj, "y")) == 1}))

## Why didn't that work? You can only override the slots "x", "y",
## and "smooth" if you provide an explicit coerce function to correct
## any inconsistencies:

setIs("trackMultiCurve", "trackCurve",
  test = function(obj) {ncol(slot(obj, "y")) == 1},
  coerce = function(obj) {
    new("trackCurve",
      x = slot(obj, "x"),
      y = as.numeric(slot(obj, "y")),
      smooth = as.numeric(slot(obj, "smooth")))
  })

## A class that extends the built-in data type "numeric"

setClass("numWithId", representation(id = "character"),
  contains = "numeric")

new("numWithId", 1:3, id = "An Example")

```

---

setClassUnion

*Classes Defined as the Union of Other Classes*


---

### Description

A class may be defined as the *union* of other classes; that is, as a virtual class defined as a superclass of several other classes. Class unions are useful in method signatures or as slots in other classes, when we want to allow one of several classes to be supplied.

### Usage

```

setClassUnion(name, members, where)
isClassUnion(Class)

```

### Arguments

name	the name for the new union class.
members	the classes that should be members of this union.
where	where to save the new class definition; by default, the environment of the package in which the setClassUnion call appears, or the global environment if called outside of the source of a package.
Class	the name or definition of a class.

## Details

The classes in `members` must be defined before creating the union. However, members can be added later on to an existing union, as shown in the example below. Class unions can be members of other class unions.

Class unions are the only way to create a class that is extended by a class whose definition is sealed (for example, the basic datatypes or other classes defined in the base or methods package in R are sealed). You cannot say `setIs("function", "other")` unless "other" is a class union. In general, a `setIs` call of this form changes the definition of the first class mentioned (adding "other" to the list of superclasses contained in the definition of "function").

Class unions get around this by not modifying the first class definition, relying instead on storing information in the subclasses slot of the class union. In order for this technique to work, the internal computations for expressions such as `extends(class1, class2)` work differently for class unions than for regular classes; specifically, they test whether any class is in common between the superclasses of `class1` and the subclasses of `class2`.

The different behavior for class unions is made possible because the class definition object for class unions has itself a special class, "ClassUnionRepresentation", an extension of "classRepresentation" (see [classRepresentation-class](#)).

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## Examples

```
## a class for either numeric or logical data
setClassUnion("maybeNumber", c("numeric", "logical"))

## use the union as the data part of another class
setClass("withId", representation("maybeNumber", id = "character"))

w1 <- new("withId", 1:10, id = "test 1")
w2 <- new("withId", sqrt(w1)%1 < .01, id = "Perfect squares")

## add class "complex" to the union "maybeNumber"
setIs("complex", "maybeNumber")

w3 <- new("withId", complex(real = 1:10, imaginary = sqrt(1:10)))

## a class union containing the existing class union "OptionalFunction"
```

```

setClassUnion("maybeCode",
  c("expression", "language", "OptionalFunction"))

is(quote(sqrt(1:10)), "maybeCode") ## TRUE

```

---

setGeneric

*Define a New Generic Function*


---

### Description

Create a new generic function of the given name, for which formal methods can then be defined. Typically, an existing non-generic function becomes the default method, but there is much optional control. See the details section.

### Usage

```

setGeneric(name, def= , group=list(), valueClass=character(), where= ,
  package= , signature= , useAsDefault= , genericFunction= )

setGroupGeneric(name, def= , group=list(), valueClass=character(),
  knownMembers=list(), package= , where= )

```

### Arguments

- |            |   |
|------------|---|
| name       | The character string name of the generic function. In the simplest and most common case, a function of this name is already defined. The existing function may be non-generic or already a generic (see the details).   |
| def        | An optional function object, defining the generic. This argument is usually only needed (and is then required) if there is no current function of this name. In that case, the formal arguments and default values for the generic are taken from <code>def</code> . You can also supply this argument if you want the generic function to do something other than just dispatch methods (an advanced topic best left alone unless you are sure you want it).<br>Note that <code>def</code> is <i>not</i> the default method; use argument <code>useAsDefault</code> if you want to specify the default separately. |
| group      | Optionally, a character string giving the group of generic functions to which this function belongs. Methods can be defined for the corresponding group generic, and these will then define methods for this specific generic function, if no method has been explicitly defined for the corresponding signature. See the references for more discussion.   |
| valueClass | An optional character vector or unevaluated expression. The value returned by the generic function must have (or extend) this class, or one of the classes; otherwise, an error is generated. See the details section for supplying an expression.  |

package	The name of the package with which this function is associated. Usually determined automatically (as the package containing the non-generic version if there is one, or else the package where this generic is to be saved).
where	Where to store the resulting initial methods definition, and possibly the generic function; by default, stored into the top-level environment.
signature	Optionally, the signature of arguments in the function that can be used in methods for this generic. By default, all arguments other than <code>...</code> can be used. The signature argument can prohibit methods from using some arguments. The argument, if provided, is a vector of formal argument names.
genericFunction	The object to be used as a (nonstandard) generic function definition. Supply this explicitly <i>only</i> if you know what you are doing!
useAsDefault	Override the usual choice of default argument (an existing non-generic function or no default if there is no such function). Argument <code>useAsDefault</code> can be supplied, either as a function to use for the default, or as a logical value. <code>FALSE</code> says not to have a default method at all, so that an error occurs if there is not an explicit or inherited method for a call. <code>TRUE</code> says to use the existing function as default, unconditionally (hardly ever needed as an explicit argument). See the section on details.
knownMembers	(For <code>setGroupGeneric</code> only) The names of functions that are known to be members of this group. This information is used to reset cached definitions of the member generics when information about the group generic is changed.

## Details

The `setGeneric` function is called to initialize a generic function in an environment (usually the global environment), as preparation for defining some methods for that function.

The simplest and most common situation is that `name` is already an ordinary non-generic function, and you now want to turn this function into a generic. In this case you will most often supply only `name`. The existing function becomes the default method, and the special `group` and `valueClass` properties remain unspecified.

A second situation is that you want to create a new, generic function, unrelated to any existing function. In this case, you need to supply a skeleton of the function definition, to define the arguments for the function. The body of a generic function is usually a standard form, `standardGeneric(name)` where `name` is the quoted name of the generic function.

When calling `setGeneric` in this form, you would normally supply the `def` argument as a function of this form. If not told otherwise, `setGeneric` will try to find a non-generic version of the function to use as a default. If you don't want this to happen, supply the argument `useAsDefault`. That argument can be the function you want to be the default method. You can supply the argument as `FALSE` to force no default (i.e., to cause an error if there is not direct or inherited method on call to the function).

The same no-default situation occurs if there is no non-generic form of the function, and `useAsDefault=FALSE`. Remember, though, you can also just assign the default you want (even one that generates an error) rather than relying on the prior situation.

You cannot (and never need to) create an explicit generic for the primitive functions in the base library. These are dispatched from C code for efficiency and are not to be redefined in any case.

As mentioned, the body of a generic function usually does nothing except for dispatching methods by a call to `standardGeneric`. Under some circumstances you might just want to do some additional computation in the generic function itself. As long as your function eventually calls `standardGeneric` that is permissible (though perhaps not a good idea, in that it makes the behavior of your function different from the usual S model). If your explicit definition of the generic function does *not* call `standardGeneric` you are in trouble, because none of the methods for the function will ever be dispatched.

By default, the generic function can return any object. If `valueClass` is supplied, it should be a vector of class names; the value returned by a method is then required to satisfy `is(object, Class)` for one of the specified classes. An empty (i.e., zero length) vector of classes means anything is allowed. Note that more complicated requirements on the result can be specified explicitly, by defining a non-standard generic function.

The `setGroupGeneric` function behaves like `setGeneric` except that it constructs a group generic function, differing in two ways from an ordinary generic function. First, this function cannot be called directly, and the body of the function created will contain a stop call with this information. Second, the group generic function contains information about the known members of the group, used to keep the members up to date when the group definition changes, through changes in the search list or direct specification of methods, etc.

## Value

The `setGeneric` function exists for its side effect: saving the generic function to allow methods to be specified later. It returns `name`.

## Generic Functions and Primitive Functions

A number of the basic R functions are specially implemented as primitive functions, to be evaluated directly in the underlying C code rather than by evaluating an S language definition. Primitive functions are eligible to have methods, but are handled differently by `setGeneric` and `setGroupGeneric`. A call to `setGeneric` for a primitive function does not create a new definition of the function, and the call is allowed only to “turn on” methods for that function.

A call to `setGeneric` for a primitive causes the evaluator to look for methods for that generic; a call to `setGroupGeneric` for any of the groups that include primitives (`"Arith"`, `"Logic"`, `"Compare"`, `"Ops"`, `"Math"`, `"Math2"`, `"Summary"`, and `"Complex"`) does the same for each of the functions in that group.

You usually only need to use either function if the methods are being defined only for the group generic. Defining a method for a primitive function, say `"+"`, by a call to `setMethod` turns on method dispatch for that function. But in R defining a method for the corresponding group generic, `"Arith"`, does not currently turn on method dispatch (for efficiency reasons). If there are no non-group methods for the functions, you have two choices.

You can turn on method dispatch for *all* the functions in the group by calling `setGroupGeneric("Arith")`, or you can turn on method dispatch for only some of the functions by calling `setGeneric("+")`, etc. Note that in either case you should give the name of the generic function as the only argument.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[Methods](#) for a discussion of other functions to specify and manipulate the methods of generic functions.

## Examples

```
### A non-standard generic function. It insists that the methods
### return a non-empty character vector (a stronger requirement than
### valueClass = "character" in the call to setGeneric)

setGeneric("authorNames",
  function(text) {
    value <- standardGeneric("authorNames")
    if(!(is(value, "character") && any(nchar(value)>0)))
      stop("authorNames methods must return non-empty strings")
    value
  })

## An example of group generic methods, using the class
## "track"; see the documentation of setClass for its definition

#define a method for the Arith group

setMethod("Arith", c("track", "numeric"),
  function(e1, e2){
    e1@y <- callGeneric(e1@y , e2)
    e1
  })

setMethod("Arith", c("numeric", "track"),
  function(e1, e2){
    e2@y <- callGeneric(e1, e2@y)
    e2
  })
```

```
# now arithmetic operators will dispatch methods:
t1 <- new("track", x=1:10, y=sort(rnorm(10)))

t1 - 100

1/t1
```

---

setMethod

*Create and Save a Method*


---

### Description

Create and save a formal method for a given function and list of classes.

### Usage

```
setMethod(f, signature=character(), definition,
          where = toplev(parent.frame()),
          valueClass = NULL, sealed = FALSE)

removeMethod(f, signature, where)
```

### Arguments

<code>f</code>	The character-string name of the generic function.
<code>signature</code>	A match of formal argument names for <code>f</code> with the character-string names of corresponding classes. This argument can also just be the vector of class names, in which case the first name corresponds to the first formal argument, the next to the second formal argument, etc.
<code>definition</code>	A function definition, which will become the method called when the arguments in a call to <code>f</code> match the classes in <code>signature</code> , directly or through inheritance.
<code>where</code>	the database in which to store the definition of the method; For <code>removeMethod</code> , the default is the location of the (first) instance of the method for this signature.
<code>valueClass</code>	If supplied, this argument asserts that the method will return a value of this class. (At present this argument is stored but not explicitly used.)
<code>sealed</code>	If <code>TRUE</code> , the method so defined cannot be redefined by another call to <code>setMethod</code> (although it can be removed and then re-assigned). Note that this argument is an extension to the definition of <code>setMethod</code> in the reference.

## Details

R methods for a particular generic function are stored in an object of class `MethodsList`. The effect of calling `setMethod` is to store definition in a `MethodsList` object on database where. If `f` doesn't exist as a generic function, but there is an ordinary function of the same name and the same formal arguments, a new generic function is created, and the previous non-generic version of `f` becomes the default method. This is equivalent to the programmer calling `setGeneric` for the same function; it's better practice to do the call explicitly, since it shows that you intend to turn `f` into a generic function.

Methods are stored in a hierarchical structure: see [Methods](#) for how the objects are used to select a method, and [MethodsList](#) for functions that manipulate the objects.

The class names in the signature can be any formal class, plus predefined basic classes such as "numeric", "character", and "matrix". Two additional special class names can appear: "ANY", meaning that this argument can have any class at all; and "missing", meaning that this argument *must not* appear in the call in order to match this signature. Don't confuse these two: if an argument isn't mentioned in a signature, it corresponds implicitly to class "ANY", not to "missing". See the example below. Old-style ("S3") classes can also be used, if you need compatibility with these, but you should definitely declare these classes by calling `setOldClass` if you want S3-style inheritance to work.

While `f` can correspond to methods defined on several packages or environments, the underlying model is that these together make up the definition for a single generic function. When R proceeds to select and evaluate methods for `f`, the methods on the current search list are merged to form a single generic function and associated methods list. When `f` is called and a method is "dispatched", the evaluator matches the classes of the actual arguments to the signatures of the available methods. When a match is found, the body of the corresponding method is evaluated, but without rematching the arguments to `f`. Aside from not rematching the arguments, the computation proceeds as if the call had been to the method. In particular, the lexical scope of the method is used.

Method definitions can have default expressions for arguments. If those arguments are then missing in the call to the generic function, the default expression in the method is used. If the method definition has no default for the argument, then the expression (if any) supplied in the definition of the generic function itself is used. But note that this expression will be evaluated in the environment defined by the method.

It is possible to have some differences between the formal arguments to a method supplied to `setMethod` and those of the generic. Roughly, if the generic has `...as` one of its arguments, then the method may have extra formal arguments, which will be matched from the arguments matching `...` in the call to `f`. (What actually happens is that a local function is created inside the method, with its formal arguments, and the method is re-defined to call that local function.)

Method dispatch tries to match the class of the actual arguments in a call to the available methods collected for `f`. Roughly, for each formal argument in turn, we look for the best match (the exact same class or the nearest element in the value of `extends` for that class) for which there is any possible method matching the remaining arguments. See [Methods](#) for more details.

## Value

These functions exist for their side-effect, in setting or removing a method in the object defining methods for the specified generic.

The value returned by `removeMethod` is `TRUE` if a method was found to be removed.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[Methods](#), [MethodsList](#) for details of the implementation

## Examples

```
## methods for plotting track objects (see the example for setClass)
##
## First, with only one object as argument:
setMethod("plot", signature(x="track", y="missing"),
  function(x, y, ...) plot(slot(x, "x"), slot(x, "y"), ...)
)
## Second, plot the data from the track on the y-axis against anything
## as the x data.
setMethod("plot", signature(y = "track"),
  function(x, y, ...) plot(x, slot(y, "y"), ...)
)
## and similarly with the track on the x-axis (using the short form of
## specification for signatures)
setMethod("plot", "track",
  function(x, y, ...) plot(slot(x, "y"), y, ...)
)
t1 <- new("track", x=1:20, y=(1:20)^2)
tc1 <- new("trackCurve", t1)
slot(tc1, "smooth") <- smooth.spline(slot(tc1, "x"), slot(tc1, "y"))$y ##
plot(t1)
plot(qnorm(ppoints(20)), t1)
## An example of inherited methods, and of conforming method arguments
## (note the dotCurve argument in the method, which will be pulled out
## of ... in the generic.
setMethod("plot", c("trackCurve", "missing"),
  function(x, y, dotCurve = FALSE, ...) {
    plot(as(x, "track"))
    if(length(slot(x, "smooth") > 0))
      lines(slot(x, "x"), slot(x, "smooth"),
        lty = if(dotCurve) 2 else 1)
  }
)
```

```

## the plot of tcl alone has an added curve; other uses of tcl
## are treated as if it were a "track" object.
plot(tcl, dotCurve = TRUE)
plot(qnorm(ppoints(20)), tcl)

## defining methods for a special function.
## Although "[" and "length" are not ordinary functions
## methods can be defined for them.
setMethod("[", "track",
  function(x, i, j, ..., drop) {
    x@x <- x@x[i]; x@y <- x@y[i]
    x
  })
plot(tl[1:15])

setMethod("length", "track", function(x)length(x@y))
length(tl)

## methods can be defined for missing arguments as well
setGeneric("summary") ## make the function into a generic

## A method for summary()
## The method definition can include the arguments, but
## if they're omitted, class "missing" is assumed.

setMethod("summary", "missing", function() "<No Object>")

```

---

setOldClass

*Specify Names for Old-Style Classes*


---

### Description

Register an old-style (a.k.a. ‘S3’) class as a formally defined class. The `Classes` argument is the character vector used as the `class` attribute; in particular, if there is more than one string, old-style class inheritance is mimiced. Registering via `setOldClass` allows S3 classes to appear as slots or in method signatures.

### Usage

```
setOldClass(Classes, where, test = FALSE)
```

### Arguments

<code>Classes</code>	A character vector, giving the names for old-style classes, as they would appear on the right side of an assignment of the <code>class</code> attribute.
<code>where</code>	Where to store the class definitions, the global or top-level environment by default. (When either function is called in the source for a package, the class definitions will be included in the package’s environment by default.)

`test` flag, if TRUE, inheritance must be tested explicitly for each object, needed if the S3 class can have a different set of class strings, with the same first string. See the details below.

## Details

Each of the names will be defined as a virtual class, extending the remaining classes in `Classes`, and the class `oldClass`, which is the “root” of all old-style classes. See [Methods](#) for the details of method dispatch and inheritance. See the section **Register or Convert?** for comments on the alternative of defining “real” S4 classes rather than using `setOldClass`.

S3 classes have no formal definition, and some of them cannot be represented as an ordinary combination of S4 classes and superclasses. It is still possible to register the classes as S4 classes, but now the inheritance has to be verified for each object, and you must call `setOldClass` with argument `test=TRUE`.

For example, ordered factors *always* have the S3 class `c("ordered", "factor")`. This is proper behavior, and maps simply into two S4 classes, with “ordered” extending “factor”.

But objects whose class attribute has “POSIXt” as the first string may have either (or neither) of “POSIXct” or “POSIXlt” as the second string. This behavior can be mapped into S4 classes but now to evaluate `is(x, "POSIXlt")`, for example, requires checking the S3 class attribute on each object. Supplying the `test=TRUE` argument to `setOldClass` causes an explicit test to be included in the class definitions. It’s never wrong to have this test, but since it adds significant overhead to methods defined for the inherited classes, you should only supply this argument if it’s known that object-specific tests are needed.

The list `.OldClassesList` contains the old-style classes that are defined by the methods package. Each element of the list is an old-style list, with multiple character strings if inheritance is included. Each element of the list was passed to `setOldClass` when creating the `methods` package; therefore, these classes can be used in `setMethod` calls, with the inheritance as implied by the list.

## Register or Convert?

A call to `setOldClass` creates formal classes corresponding to S3 classes, allows these to be used as slots in other classes or in a signature in `setMethod`, and mimics the S3 inheritance.

However, all such classes are created as virtual classes, meaning that you cannot generally create new objects from the class by calling `new`, and that objects cannot be coerced automatically from or to these classes. All these restrictions just reflect the fact that nothing is inherently known about the “structure” of S3 classes, or whether in fact they define a consistent set of attributes that can be mapped into slots in a formal class definition.

*If* your class does in fact have a consistent structure, so that every object from the class has the same structure, you may prefer to take some extra time to write down a specific definition in a call to `setClass` to convert the class to a fully functional formal class. On the other hand, if the actual contents of the class vary from one object to another, you may have to redesign most of the software using the class, in which case converting it may not be worth the effort. You should still register the class via `setOldClass`, unless its class attribute is hopelessly unpredictable.

An S3 class has consistent structure if each object has the same set of attributes, both the names and the classes of the attributes being the same for every object in the class. In practice, you can convert classes that are slightly less well behaved. If a few attributes appear in some but not all

objects, you can include these optional attributes as slots that *always* appear in the objects, if you can supply a default value that is equivalent to the attribute being missing. Sometimes NULL can be that value: A slot (but not an attribute) can have the value NULL. If `version`, for example, was an optional attribute, the old test `is.null(attr(x, "version"))` for a missing version attribute could turn into `is.null(x@version)` for the formal class.

The requirement that slots have a fixed class can be satisfied indirectly as well. Slots *can* be specified with class "ANY", allowing an arbitrary object. However, this eliminates an important benefit of formal class definitions; namely, automatic validation of objects assigned to a slot. If just a few different classes are possible, consider using `setClassUnion` to define valid objects for a slot.

### See Also

[setClass](#), [setMethod](#)

### Examples

```
setOldClass(c("mlm", "lm"))
setGeneric("dfResidual", function(model)standardGeneric("dfResidual"))
setMethod("dfResidual", "lm", function(model)model$df.residual)

## dfResidual will work on mlm objects as well as lm objects
myData <- data.frame(time = 1:10, y = (1:10)^.5)
myLm <- lm(cbind(y, y^3) ~ time, myData)

rm(myData, myLm)
removeGeneric("dfResidual")
```

---

show

*Show an Object*

---

### Description

Display the object, by printing, plotting or whatever suits its class. This function exists to be specialized by methods. The default method calls `showDefault`.

Formal methods for `show` will usually be invoked for automatic printing (see the details).

### Usage

```
show(object)
```

### Arguments

`object` Any R object

## Details

The **methods** package overrides the base definition of `print.default` to arrange for automatic printing to honor methods for the function `show`. This does not quite manage to override old-style printing methods, since the automatic printing in the evaluator will look first for the old-style method.

If you have a class `myClass` and want to define a method for `show`, all will be well unless there is already a function named `print.myClass`. In that case, to get your method dispatched for automatic printing, it will have to be a method for `print`. A slight cheat is to override the function `print.myClass` yourself, and then call that function also in the method for `show` with signature `"myClass"`.

## Value

`show` returns an invisible `NULL`.

## See Also

[showMethods](#) prints all the methods for one or more functions; [showMlist](#) prints individual methods lists; [showClass](#) prints class definitions. Neither of the latter two normally needs to be called directly.

## Examples

```
## following the example shown in the setMethod documentation ...
setClass("track",
         representation(x="numeric", y="numeric"))
setClass("trackCurve",
         representation("track", smooth = "numeric"))

t1 <- new("track", x=1:20, y=(1:20)^2)

tc1 <- new("trackCurve", t1)

setMethod("show", "track",
         function(object)print(rbind(x = object@x, y=object@y))
)
## The method will now be used for automatic printing of t1

t1

## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9   10   11   12
y    1    4    9   16   25   36   49   64   81  100  121  144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x   13   14   15   16   17   18   19   20
y  169  196  225  256  289  324  361  400
## End(Not run)
## and also for tc1, an object of a class that extends "track"
tc1
```

```
## Not run:
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
x    1    2    3    4    5    6    7    8    9   10   11   12
y    1    4    9   16   25   36   49   64   81  100  121  144
  [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
x    13   14   15   16   17   18   19   20
y   169  196  225  256  289  324  361  400
## End(Not run)
```

---

showMethods                      *Show all the methods for the specified function(s)*

---

### Description

Show a summary of the methods for one or more generic functions, possibly restricted to those involving specified classes.

### Usage

```
showMethods(f = character(), where = topenv(parent.frame()),
            classes = NULL, includeDefs = FALSE, inherited = TRUE,
            printTo = stdout())
```

### Arguments

<code>f</code>	one or more function names. If omitted, all functions will be examined.
<code>where</code>	If <code>where</code> is supplied, the methods definition from that position will be used; otherwise, the current definition is used (which will include inherited methods that have arisen so far in the session). If <code>f</code> is omitted, <code>where</code> controls where to look for generic functions.
<code>classes</code>	If argument <code>classes</code> is supplied, it is a vector of class names that restricts the displayed results to those methods whose signatures include one or more of those classes.
<code>includeDefs</code>	If <code>includeDefs</code> is TRUE, include the definitions of the individual methods in the printout.
<code>inherited</code>	If <code>inherits</code> is TRUE, then methods that have been found by inheritance, so far in the session, will be included and marked as inherited. Note that an inherited method will not usually appear until it has been used in this session. See <a href="#">selectMethod</a> if you want to know what method is dispatched for particular classes of arguments.
<code>printTo</code>	The connection on which the printed information will be written. If <code>printTo</code> is FALSE, the output will be collected as a character vector and returned as the value of the call to <code>showMethod</code> . See <a href="#">show</a> .

### Details

The output style is different from S-Plus in that it does not show the database from which the definition comes, but can optionally include the method definitions.

**Value**

If `printTo` is `FALSE`, the character vector that would have been printed is returned; otherwise the value is the connection or filename.

**References**

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

**See Also**

[setMethod](#), and [GenericFunctions](#) for other tools involving methods; [selectMethod](#) will show you the method dispatched for a particular function and signature of classes for the arguments.

**Examples**

```
## assuming the methods for plot
## are set up as in the documentation for setMethod,
## print (without definitions) the methods that involve
## class "track"
showMethods("plot", classes = "track")
## Not run:
Function "plot":
x = ANY, y = track
x = track, y = missing
x = track, y = ANY
## End(Not run)
```

---

signature-class      *Class "signature" For Method Definitions*

---

**Description**

This class represents the mapping of some of the formal arguments of a function onto the names of some classes. It is used as one of two slots in the [MethodDefinition-class](#).

### Objects from the Class

Objects can be created by calls of the form `new("signature", functionDef, ...)`. The `functionDef` argument, if it is supplied as a function object, defines the formal names. The other arguments define the classes.

### Slots

**.Data:** Object of class "character" the classes.

**names:** Object of class "character" the corresponding argument names.

### Extends

Class "character", from data part. Class "vector", by class "character".

### Methods

**initialize** `signature(object = "signature")`: see the discussion of objects from the class, above.

### See Also

[MethodDefinition-class](#) for the use of this class

---

slot

*The Slots in an Object from a Formal Class*

---

### Description

These functions return or set information about the individual slots in an object.

### Usage

```
object@name
object@name <- value

slot(object, name)
slot(object, name, check = TRUE) <- value

slotNames(x)
```

### Arguments

<code>object</code>	An object from a formally defined class.
<code>name</code>	The character-string name of the slot. The name must be a valid slot name: see Details below.
<code>value</code>	A new value for the named slot. The value must be valid for this slot in this object's class.

x	Either the name of a class or an object from that class. Print <code>getClass(class)</code> to see the full description of the slots.
check	If TRUE, check the assigned value for validity as the value of this slot. You should never set this to FALSE in normal use, since the result can create invalid objects.

### Details

The "@" operator and the `slot` function extract or replace the formally defined slots for the object. The operator takes a fixed name, which can be unquoted if it is syntactically a name in the language. A slot name can be any non-empty string, but if the name is not made up of letters, numbers, and ". ", it needs to be quoted.

In the case of the `slot` function, the slot name can be any expression that evaluates to a valid slot in the class definition. Generally, the only reason to use the functional form rather than the simpler operator is *because* the slot name has to be computed.

The definition of the class contains the names of all slots directly and indirectly defined. Each slot has a name and an associated class. Extracting a slot returns an object from that class. Setting a slot first coerces the value to the specified slot and then stores it.

Unlike attributes, slots are not partially matched, and asking for (or trying to set) a slot with an invalid name for that class generates an error.

### References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

### See Also

[@](#), [Classes](#), [Methods](#), [getClass](#)

### Examples

```
## Not run:
slot(myTrack, "x")
slot(myTrack, "y") <- log(slot(myTrack, "x"))

slotNames("track")
## End(Not run)
```

---

 StructureClasses    *Classes Corresponding to Basic Structures*


---

**Description**

The virtual class `structure` and classes that extend it are formal classes analogous to S language structures such as arrays and time-series

**Usage**

```
## The following class names can appear in method signatures,
## as the class in as() and is() expressions, and, except for
## the classes commented as VIRTUAL, in calls to new()
```

```
"matrix"
"array"
"ts"

"structure" ## VIRTUAL
```

**Objects from the Classes**

Objects can be created by calls of the form `new(Class, ...)`, where `Class` is the quoted name of the specific class (e.g., `"matrix"`), and the other arguments, if any, are interpreted as arguments to the corresponding function, e.g., to function `matrix()`. There is no particular advantage over calling those functions directly, unless you are writing software designed to work for multiple classes, perhaps with the class name and the arguments passed in.

**Extends**

The specific classes all extend class `"structure"`, directly, and class `"vector"`, by class `"structure"`.

**Methods**

**coerce** Methods are defined to coerce arbitrary objects to these classes, by calling the corresponding basic function, for example, `as(x, "matrix")` calls `as.matrix(x)`.

---

 TraceClasses    *Classes Used Internally to Control Tracing*


---

**Description**

The classes described here are used by the R function `trace` to create versions of functions and methods including browser calls, etc., and also to `untrace` the same objects.

**Usage**

```

### Objects from the following classes are generated
### by calling trace() on an object from the corresponding
### class without the "WithTrace" in the name.

"functionWithTrace"
"MethodDefinitionWithTrace"
"MethodWithNextWithTrace"
"genericFunctionWithTrace"
"groupGenericFunctionWithTrace"

### the following is a virtual class extended by each of the
### classes above

"traceable"

```

**Objects from the Class**

Objects will be created from these classes by calls to `trace`. (There is an `initialize` method for class "traceable", but you are unlikely to need it directly.)

**Slots**

**.Data:** The data part, which will be "function" for class "functionWithTrace", and similarly for the other classes.

**original:** Object of the original class; e.g., "function" for class "functionWithTrace".

**Extends**

Each of the classes extends the corresponding untraced class, from the data part; e.g., "functionWithTrace" extends "function". Each of the specific classes extends "traceable", directly, and class "VIRTUAL", by class "traceable".

**Methods**

The point of the specific classes is that objects generated from them, by function `trace()`, remain callable or dispatchable, in addition to their new trace information.

**See Also**

function `trace`

---

 validObject

*Test the Validity of an Object*


---

### Description

The validity of `object` related to its class definition is tested. If the object is valid, `TRUE` is returned; otherwise, either a vector of strings describing validity failures is returned, or an error is generated (according to whether `test` is `TRUE`).

The function `setValidity` sets the validity method of a class (but more normally, this method will be supplied as the `validity` argument to `setClass`). The method should be a function of one object that returns `TRUE` or a description of the non-validity.

### Usage

```
validObject(object, test)
```

```
setValidity(Class, method, where = toplev(parent.frame()) )
```

### Arguments

<code>object</code>	Any object, but not much will happen unless the object's class has a formal definition.
<code>test</code>	If <code>test</code> is <code>TRUE</code> , and validity fails the function returns a vector of strings describing the problems. If <code>test</code> is <code>FALSE</code> (the default) validity failure generates an error.
<code>Class</code>	the name or class definition of the class whose validity method is to be set.
<code>method</code>	a validity method; that is, either <code>NULL</code> or a function of one argument (the <code>object</code> ). Like <code>validObject</code> , the function should return <code>TRUE</code> if the object is valid, and one or more descriptive strings if any problems are found. Unlike <code>validObject</code> , it should never generate an error.
<code>where</code>	the modified class definition will be stored in this environment.

Note that validity methods do not have to check validity of any slots or superclasses: the logic of `validObject` ensures these tests are done once only. As a consequence, if one validity method wants to use another, it should extract and call the method from the other definition of the other class by calling `getValidity`: it should *not* call `validObject`.

### Details

Validity testing takes place “bottom up”: first the validity of the object's slots, if any, is tested. Then for each of the classes that this class extends (the “superclasses”), the explicit validity method of that class is called, if one exists. Finally, the validity method of `object`'s class is called, if there is one.

Testing generally stops at the first stage of finding an error, except that all the slots will be examined even if a slot has failed its validity test.

## Value

`validObject` returns `TRUE` if the object is valid. Otherwise a vector of strings describing problems found, except that if `test` is `FALSE`, validity failure generates an error, with the corresponding strings in the error message.

## References

The R package **methods** implements, with a few exceptions, the programming interface for classes and methods in the book *Programming with Data* (John M. Chambers, Springer, 1998), in particular sections 1.6, 2.7, 2.8, and chapters 7 and 8.

While the programming interface for the **methods** package follows the reference, the R software is an original implementation, so details in the reference that reflect the S4 implementation may appear differently in R. Also, there are extensions to the programming interface developed more recently than the reference. For a discussion of details and ongoing development, see the web page <http://developer.r-project.org/methodsPackage.html> and the pointers from that page.

## See Also

[setClass](#).

## Examples

```
setClass("track",
         representation(x="numeric", y = "numeric"))
t1 <- new("track", x=1:10, y=sort(rnorm(10)))
## A valid "track" object has the same number of x, y values
validTrackObject <- function(x){
  if(length(x@x) == length(x@y)) TRUE
  else paste("Unequal x,y lengths: ", length(x@x), ", ", length(x@y),
            sep="")
}
## assign the function as the validity method for the class
setValidity("track", validTrackObject)
## t1 should be a valid "track" object
validObject(t1)
## Now we do something bad
t1@x <- 1:20
## This should generate an error
## Not run: try(validObject(t1))
```



## Chapter 7

# The stats package

---

`.checkMFClasses`      *Functions to Check the Type of Variables passed to Model Frames*

---

### Description

`.checkMFClasses` checks if the variables used in a predict method agree in type with those used for fitting.

`.MFclass` categories variables for this purpose.

### Usage

```
.checkMFClasses(cl, m, ordNotOK = FALSE)
.MFclass(x)
.getXlevels(Terms, m)
```

### Arguments

<code>cl</code>	a character vector of class descriptions to match.
<code>m</code>	a model frame.
<code>x</code>	any R object.
<code>ordNotOK</code>	logical: are ordered factors different?
<code>Terms</code>	a terms object.

### Details

For applications involving `model.matrix` such as linear models we do not need to differentiate between ordered factors and factors as although these affect the coding, the coding used in the fit is already recorded and imposed during prediction. However, other applications may treat ordered factors differently: `rpart` does, for example.

**Value**

`.MFclass` returns a character string, one of "logical", "ordered", "factor", "numeric", "nmatrix.\*" (a numeric matrix with a number of columns appended) or "other".

`.getXlevels` returns a named character vector, or NULL.

---

 acf

---

*Auto- and Cross- Covariance and -Correlation Function Estimation*


---

**Description**

The function `acf` computes (and by default plots) estimates of the autocovariance or autocorrelation function. Function `pacf` is the function used for the partial autocorrelations. Function `ccf` computes the cross-correlation or cross-covariance of two univariate series.

**Usage**

```
acf(x, lag.max = NULL,
    type = c("correlation", "covariance", "partial"),
    plot = TRUE, na.action = na.fail, demean = TRUE, ...)

pacf(x, lag.max, plot, na.action, ...)

## Default S3 method:
pacf(x, lag.max = NULL, plot = TRUE, na.action = na.fail,
     ...)

ccf(x, y, lag.max = NULL, type = c("correlation", "covariance"),
    plot = TRUE, na.action = na.fail, ...)

acf.obj[i, j]
```

**Arguments**

<code>x, y</code>	a univariate or multivariate (not <code>ccf</code> ) numeric time series object or a numeric vector or matrix.
<code>lag.max</code>	maximum number of lags at which to calculate the acf. Default is $10 \log_{10}(N/m)$ where $N$ is the number of observations and $m$ the number of series.
<code>type</code>	character string giving the type of acf to be computed. Allowed values are "correlation" (the default), "covariance" or "partial".
<code>plot</code>	logical. If TRUE (the default) the acf is plotted.
<code>na.action</code>	function to be called to handle missing values. <code>na.pass</code> can be used.
<code>demean</code>	logical. Should the covariances be about the sample means?
<code>...</code>	further arguments to be passed to <code>plot.acf</code> .

<code>acf.obj</code>	an object of class "acf" resulting from a call to <code>acf</code> .
<code>i</code>	a set of lags to retain.
<code>j</code>	a set of series to retain.

### Details

For `type = "correlation"` and `"covariance"`, the estimates are based on the sample covariance.

By default, no missing values are allowed. If the `na.action` function passes through missing values (as `na.pass` does), the covariances are computed from the complete cases. This means that the estimate computed may well not be a valid autocorrelation sequence, and may contain missing values. Missing values are not allowed when computing the PACF of a multivariate time series.

The partial correlation coefficient is estimated by fitting autoregressive models of successively higher orders up to `lag.max`.

The generic function `plot` has a method for objects of class "acf".

The lag is returned and plotted in units of time, and not numbers of observations.

There are `print` and subsetting methods for objects of class "acf".

### Value

An object of class "acf", which is a list with the following elements:

<code>lag</code>	A three dimensional array containing the lags at which the acf is estimated.
<code>acf</code>	An array with the same dimensions as <code>lag</code> containing the estimated acf.
<code>type</code>	The type of correlation (same as the <code>type</code> argument).
<code>n.used</code>	The number of observations in the time series.
<code>series</code>	The name of the series <code>x</code> .
<code>snames</code>	The series names for a multivariate time series.

The result is returned invisibly if `plot` is `TRUE`.

### Author(s)

Original: Paul Gilbert, Martyn Plummer. Extensive modifications and univariate case of `pacf` by B.D. Ripley.

### See Also

[plot.acf](#)

**Examples**

```
## Examples from Venables & Ripley
acf(lh)
acf(lh, type = "covariance")
pacf(lh)

acf(ldeaths)
acf(ldeaths, ci.type = "ma")
acf(ts.union(mdeaths, fdeaths))
ccf(mdeaths, fdeaths) # just the cross-correlations.

presidents # contains missing values
acf(presidents, na.action = na.pass)
pacf(presidents, na.action = na.pass)
```

---

 acf2AR

---

*Compute an AR Process Exactly Fitting an ACF*


---

**Description**

Compute an AR process exactly fitting an autocorrelation function.

**Usage**

```
acf2AR(acf)
```

**Arguments**

`acf`                    An autocorrelation or autocovariance sequence.

**Value**

A matrix, with one row for the computed AR(p) coefficients for  $1 \leq p \leq \text{length}(\text{acf})$ .

**See Also**

[ARMAacf](#), [ar.yw](#) which does this from an empirical ACF.

**Examples**

```
(Acf <- ARMAacf(c(0.6, 0.3, -0.2)))
acf2AR(Acf)
```

---

add1 *Add or Drop All Possible Single Terms to a Model*

---

### Description

Compute all the single terms in the `scope` argument that can be added to or dropped from the model, fit those models and compute a table of the changes in fit.

### Usage

```
add1(object, scope, ...)

## Default S3 method:
add1(object, scope, scale = 0, test = c("none", "Chisq"),
      k = 2, trace = FALSE, ...)

## S3 method for class 'lm':
add1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
      x = NULL, k = 2, ...)

## S3 method for class 'glm':
add1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
      x = NULL, k = 2, ...)

drop1(object, scope, ...)

## Default S3 method:
drop1(object, scope, scale = 0, test = c("none", "Chisq"),
       k = 2, trace = FALSE, ...)

## S3 method for class 'lm':
drop1(object, scope, scale = 0, all.cols = TRUE,
       test = c("none", "Chisq", "F"), k = 2, ...)

## S3 method for class 'glm':
drop1(object, scope, scale = 0, test = c("none", "Chisq", "F"),
       k = 2, ...)
```

### Arguments

<code>object</code>	a fitted model object.
<code>scope</code>	a formula giving the terms to be considered for adding or dropping.
<code>scale</code>	an estimate of the residual mean square to be used in computing $C_p$ . Ignored if 0 or NULL.
<code>test</code>	should the results include a test statistic relative to the original model? The F test is only appropriate for <code>lm</code> and <code>av</code> models or perhaps for <code>glm</code> fits with estimated dispersion. The $\chi^2$ test can be an exact test ( <code>lm</code> models with known scale)

	or a likelihood-ratio test or a test of the reduction in scaled deviance depending on the method.
<code>k</code>	the penalty constant in $AIC / C_p$ .
<code>trace</code>	if TRUE, print out progress reports.
<code>x</code>	a model matrix containing columns for the fitted model and all terms in the upper scope. Useful if <code>add1</code> is to be called repeatedly.
<code>all.cols</code>	(Provided for compatibility with S.) Logical to specify whether all columns of the design matrix should be used. If FALSE then non-estimable columns are dropped, but the result is not usually statistically meaningful.
<code>...</code>	further arguments passed to or from other methods.

### Details

For `drop1` methods, a missing `scope` is taken to be all terms in the model. The hierarchy is respected when considering terms to be added or dropped: all main effects contained in a second-order interaction must remain, and so on.

The methods for `lm` and `glm` are more efficient in that they do not recompute the model matrix and call the `fit` methods directly.

The default output table gives AIC, defined as minus twice log likelihood plus  $2p$  where  $p$  is the rank of the model (the number of effective parameters). This is only defined up to an additive constant (like log-likelihoods). For linear Gaussian models with fixed scale, the constant is chosen to give Mallows'  $C_p$ ,  $RSS/scale + 2p - n$ . Where  $C_p$  is used, the column is labelled as  $C_p$  rather than AIC.

### Value

An object of class "anova" summarizing the differences in fit between the models.

### Warning

The model fitting must apply the models to the same dataset. Most methods will attempt to use a subset of the data with no missing values for any of the variables if `na.action=na.omit`, but this may give biased results. Only use these functions with data containing missing values with great care.

### Note

These are not fully equivalent to the functions in S. There is no `keep` argument, and the methods used are not quite so computationally efficient.

Their authors' definitions of Mallows'  $C_p$  and Akaike's AIC are used, not those of the authors of the models chapter of S.

### Author(s)

The design was inspired by the S functions of the same names described in Chambers (1992).

**References**

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[step](#), [aov](#), [lm](#), [extractAIC](#), [anova](#)

**Examples**

```
example(step)#-> swiss
add1(lm1, ~ I(Education^2) + .^2)
drop1(lm1, test="F") # So called 'type II' anova

example(glm)
drop1(glm.D93, test="Chisq")
drop1(glm.D93, test="F")
```

---

 addmargins

---

*Puts arbitrary margins on multidimensional tables or arrays.*


---

**Description**

For a given table one can specify which of the classifying factors to expand by one or more levels to hold margins to be calculated. One may for example form sums and means over the first dimension and medians over the second. The resulting table will then have two extra levels for the first dimension and one extra level for the second. The default is to sum over all margins in the table. Other possibilities may give results that depend on the order in which the margins are computed. This is flagged in the printed output from the function.

**Usage**

```
addmargins(A, margin = 1:length(dim(A)), FUN = sum, quiet = FALSE)
```

**Arguments**

A	A table or array. The function uses the presence of the "dim" and "dimnames" attributes of A
margin	Vector of dimensions over which to form margins. Margins are formed in the order in which dimensions are specified in margin.
FUN	List of the same length as margin, each element of the list being either a function or a list of functions. Names of the list elements will appear as levels in dimnames of the result. Unnamed list elements will have names constructed: the name of a function or a constructed name based on the position in the table.
quiet	Logical which suppresses the message telling the order in which the margins were computed.

**Details**

If the functions used to form margins are not commutative the result depends on the order in which margins are computed. Annotation of margins is done via naming the FUN list.

**Value**

A table with the same number of dimensions as A, but with extra levels of the dimensions mentioned in `margin`. The number of levels added to each dimension is the length of the entries in FUN. A message with the order of computation of margins is printed.

**Author(s)**

Bendix Carstensen, Steno Diabetes Center & Department of Biostatistics, University of Copenhagen, <http://www.biostat.ku.dk/~bxc>, autumn 2003. Margin naming enhanced by Duncan Murdoch.

**See Also**

[table](#), [ftable](#), [margin.table](#).

**Examples**

```
Aye <- sample( c("Yes","Si","Oui"), 177, replace=TRUE )
Bee <- sample( c("Hum","Buzz"), 177, replace=TRUE )
Sea <- sample( c("White","Black","Red","Dead"), 177, replace=TRUE )
A <- table( Aye, Bee, Sea )
A
addmargins( A )
ftable( A )
ftable( addmargins( A ) )

# Non commutative functions - note differences between resulting tables:
ftable(addmargins(A, c(1,3),
  FUN = list(Sum=sum, list(Min=min, Max=max))))
ftable(addmargins(A, c(3,1),
  FUN = list(list(Min=min, Max=max), Sum=sum)))

# Weird function needed to return the N when computing percentages
sqsm <- function( x ) sum( x )^2/100
B <- table(Sea, Bee)
round(sweep(addmargins(B, 1, list(list(All=sum, N=sqsm))), 2,
  apply( B, 2, sum )/100, "/" ), 1)
round(sweep(addmargins(B, 2, list(list(All=sum, N=sqsm))), 1,
  apply(B, 1, sum )/100, "/" ), 1)

# A total over Bee requires formation of the Bee-margin first:
mB <- addmargins(B, 2, FUN=list(list(Total=sum)) )
round(ftable(sweep(addmargins(mB, 1, list(list(All=sum, N=sqsm))), 2,
  apply(mB,2,sum )/100, "/" )), 1)
```

---

 aggregate

---

*Compute Summary Statistics of Data Subsets*


---

**Description**

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

**Usage**

```
aggregate(x, ...)

## Default S3 method:
aggregate(x, ...)

## S3 method for class 'data.frame':
aggregate(x, by, FUN, ...)

## S3 method for class 'ts':
aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,
          ts.eps = getOption("ts.eps"), ...)
```

**Arguments**

<code>x</code>	an R object.
<code>by</code>	a list of grouping elements, each as long as the variables in <code>x</code> . Names for the grouping variables are provided if they are not given. The elements of the list will be coerced to factors (if they are not already factors).
<code>FUN</code>	a scalar function to compute the summary statistics which can be applied to all data subsets.
<code>nfrequency</code>	new number of observations per unit of time; must be a divisor of the frequency of <code>x</code> .
<code>ndeltat</code>	new fraction of the sampling period between successive observations; must be a divisor of the sampling interval of <code>x</code> .
<code>ts.eps</code>	tolerance used to decide if <code>nfrequency</code> is a sub-multiple of the original frequency.
<code>...</code>	further arguments passed to or used by methods.

**Details**

`aggregate` is a generic function with methods for data frames and time series.

The default method `aggregate.default` uses the time series method if `x` is a time series, and otherwise coerces `x` to a data frame and calls the data frame method.

`aggregate.data.frame` is the data frame method. If `x` is not a data frame, it is coerced to one. Then, each of the variables (columns) in `x` is split into subsets of cases (rows) of identical

combinations of the components of `by`, and `FUN` is applied to each such subset with further arguments in `...` passed to it. (I.e., `tapply(VAR, by, FUN, ..., simplify = FALSE)` is done for each variable `VAR` in `x`, conveniently wrapped into one call to `lapply()`.) Empty subsets are removed, and the result is reformatted into a data frame containing the variables in `by` and `x`. The ones arising from `by` contain the unique combinations of grouping values used for determining the subsets, and the ones arising from `x` the corresponding summary statistics for the subset of the respective variables in `x`.

`aggregate.ts` is the time series method. If `x` is not a time series, it is coerced to one. Then, the variables in `x` are split into appropriate blocks of length `frequency(x) / nfrequency`, and `FUN` is applied to each such block, with further (named) arguments in `...` passed to it. The result returned is a time series with frequency `nfrequency` holding the aggregated values.

### Author(s)

Kurt Hornik

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[apply](#), [lapply](#), [tapply](#).

### Examples

```
## Compute the averages for the variables in 'state.x77', grouped
## according to the region (Northeast, South, North Central, West) that
## each state belongs to.
aggregate(state.x77, list(Region = state.region), mean)

## Compute the averages according to region and the occurrence of more
## than 130 days of frost.
aggregate(state.x77,
          list(Region = state.region,
               Cold = state.x77[,"Frost"] > 130),
          mean)
## (Note that no state in 'South' is THAT cold.)

## Compute the average annual approval ratings for American presidents.
aggregate(presidents, nf = 1, FUN = mean)
## Give the summer less weight.
aggregate(presidents, nf = 1, FUN = weighted.mean, w = c(1, 1, 0.5, 1))
```

**Description**

Generic function calculating the Akaike information criterion for one or several fitted model objects for which a log-likelihood value can be obtained, according to the formula  $-2\log\text{-likelihood} + kn_{par}$ , where  $n_{par}$  represents the number of parameters in the fitted model, and  $k = 2$  for the usual AIC, or  $k = \log(n)$  ( $n$  the number of observations) for the so-called BIC or SBC (Schwarz's Bayesian criterion).

**Usage**

```
AIC(object, ..., k = 2)
```

**Arguments**

object	a fitted model object, for which there exists a <code>logLik</code> method to extract the corresponding log-likelihood, or an object inheriting from class <code>logLik</code> .
...	optionally more fitted model objects.
k	numeric, the "penalty" per parameter to be used; the default $k = 2$ is the classical AIC.

**Details**

The default method for AIC, `AIC.default()` entirely relies on the existence of a `logLik` method computing the log-likelihood for the given class.

When comparing fitted objects, the smaller the AIC, the better the fit.

**Value**

If just one object is provided, returns a numeric value with the corresponding AIC (or BIC, or ..., depending on  $k$ ); if more than one object are provided, returns a `data.frame` with rows corresponding to the objects and columns representing the number of parameters in the model (`df`) and the AIC.

**Author(s)**

Jose Pinheiro and Douglas Bates

**References**

Sakamoto, Y., Ishiguro, M., and Kitagawa G. (1986). *Akaike Information Criterion Statistics*. D. Reidel Publishing Company.

**See Also**

[extractAIC](#), [logLik](#).

**Examples**

```
lm1 <- lm(Fertility ~ . , data = swiss)
AIC(lm1)
stopifnot(all.equal(AIC(lm1),
                    AIC(logLik(lm1))))
## a version of BIC or Schwarz' BC :
AIC(lm1, k = log(nrow(swiss)))
```

---

alias

*Find Aliases (Dependencies) in a Model*

---

**Description**

Find aliases (linearly dependent terms) in a linear model specified by a formula.

**Usage**

```
alias(object, ...)

## S3 method for class 'formula':
alias(object, data, ...)

## S3 method for class 'lm':
alias(object, complete = TRUE, partial = FALSE,
      partial.pattern = FALSE, ...)
```

**Arguments**

object	A fitted model object, for example from <code>lm</code> or <code>aov</code> , or a formula for <code>alias.formula</code> .
data	Optionally, a data frame to search for the objects in the formula.
complete	Should information on complete aliasing be included?
partial	Should information on partial aliasing be included?
partial.pattern	Should partial aliasing be presented in a schematic way? If this is done, the results are presented in a more compact way, usually giving the deciles of the coefficients.
...	further arguments passed to or from other methods.

**Details**

Although the main method is for class "lm", `alias` is most useful for experimental designs and so is used with fits from `aov`. Complete aliasing refers to effects in linear models that cannot be estimated independently of the terms which occur earlier in the model and so have their coefficients omitted from the fit. Partial aliasing refers to effects that can be estimated less precisely because of correlations induced by the design.

**Value**

A list (of class "listof") containing components

Model	Description of the model; usually the formula.
Complete	A matrix with columns corresponding to effects that are linearly dependent on the rows; may be of class "mtable" which has its own <code>print</code> method.
Partial	The correlations of the estimable effects, with a zero diagonal.

**Note**

The aliasing pattern may depend on the contrasts in use: Helmert contrasts are probably most useful. The defaults are different from those in `S`.

**Author(s)**

The design was inspired by the `S` function of the same name described in Chambers *et al.* (1992).

**References**

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments*. Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**Examples**

```
had.VR <- "package:MASS" %in% search()
## The next line is for fractions() which gives neater results
if(!had.VR) res <- require(MASS)
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,55.0,
          62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)
npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)

op <- options(contrasts=c("contr.helmert", "contr.poly"))
npk.aov <- aov(yield ~ block + N*P*K, npk)
alias(npk.aov)
if(!had.VR && res) detach(package:MASS)
options(op)# reset
```

---

`anova`*Anova Tables*

---

**Description**

Compute analysis of variance (or deviance) tables for one or more fitted model objects.

**Usage**

```
anova(object, ...)
```

**Arguments**

<code>object</code>	an object containing the results returned by a model fitting function (e.g., <code>lm</code> or <code>glm</code> ).
<code>...</code>	additional objects of the same type.

**Value**

This (generic) function returns an object of class `anova`. These objects represent analysis-of-variance and analysis-of-deviance tables. When given a single argument it produces a table which tests whether the model terms are significant.

When given a sequence of objects, `anova` tests the models against one another in the order specified.

The print method for `anova` objects prints tables in a “pretty” form.

**Warning**

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R’s default of `na.action = na.omit` is used.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole.

**See Also**

[coefficients](#), [effects](#), [fitted.values](#), [residuals](#), [summary](#), [dropl](#), [addl](#).

## Description

Compute an analysis of deviance table for one or more generalized linear model fits.

## Usage

```
## S3 method for class 'glm':  
anova(object, ..., dispersion = NULL, test = NULL)
```

## Arguments

`object, ...` objects of class `glm`, typically the result of a call to `glm`, or a list of objects for the `"glm"` method.

`dispersion` the dispersion parameter for the fitting family. By default it is obtained from `glm.obj`.

`test` a character string, (partially) matching one of `"ChiSq"`, `"F"` or `"Cp"`. See [stat.anova](#).

## Details

Specifying a single object gives a sequential analysis of deviance table for that fit. That is, the reductions in the residual deviance as each term of the formula is added in turn are given in as the rows of a table, plus the residual deviances themselves.

If more than one object is specified, the table has a row for the residual degrees of freedom and deviance for each model. For all but the first model, the change in degrees of freedom and deviance is also given. (This only makes statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

The table will optionally contain test statistics (and P values) comparing the reduction in deviance for the row to the residuals. For models with known dispersion (e.g., binomial and Poisson fits) the chi-squared test is most appropriate, and for those with dispersion estimated by moments (e.g., gaussian, quasibinomial and quasipoisson fits) the F test is most appropriate. Mallows'  $C_p$  statistic is the residual deviance plus twice the estimate of  $\sigma^2$  times the residual degrees of freedom, which is closely related to AIC (and a multiple of it if the dispersion is known).

## Value

An object of class `"anova"` inheriting from class `"data.frame"`.

## Warning

The comparison between two or more models by `anova` or `anova.glm` will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used, and `anova.glm` will detect this with an error.

## References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models*. Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

[glm](#), [anova](#).

[drop1](#) for so-called ‘type II’ anova where each term is dropped one at a time respecting their hierarchy.

## Examples

```
## --- Continuing the Example from '?glm':

anova(glm.D93)
anova(glm.D93, test = "Cp")
anova(glm.D93, test = "Chisq")
```

---

anova.lm

*ANOVA for Linear Model Fits*

---

## Description

Compute an analysis of variance table for one or more linear model fits.

## Usage

```
## S3 method for class 'lm':
anova(object, ...)

anova.lmlist(object, ..., scale = 0, test = "F")
```

## Arguments

`object, ...` objects of class `lm`, usually, a result of a call to [lm](#).

`test` a character string specifying the test statistic to be used. Can be one of "F", "Chisq" or "Cp", with partial matching allowed, or NULL for no test.

`scale` numeric. An estimate of the noise variance  $\sigma^2$ . If zero this will be estimated from the largest model considered.

## Details

Specifying a single object gives a sequential analysis of variance table for that fit. That is, the reductions in the residual sum of squares as each term of the formula is added in turn are given in as the rows of a table, plus the residual sum of squares.

The table will contain F statistics (and P values) comparing the mean square for the row to the residual mean square.

If more than one object is specified, the table has a row for the residual degrees of freedom and sum of squares for each model. For all but the first model, the change in degrees of freedom and sum of squares is also given. (This only make statistical sense if the models are nested.) It is conventional to list the models from smallest to largest, but this is up to the user.

Optionally the table can include test statistics. Normally the F statistic is most appropriate, which compares the mean square for a row to the residual sum of squares for the largest model considered. If `scale` is specified chi-squared tests can be used. Mallows'  $C_p$  statistic is the residual sum of squares plus twice the estimate of  $\sigma^2$  times the residual degrees of freedom.

## Value

An object of class "anova" inheriting from class "data.frame".

## Warning

The comparison between two or more models will only be valid if they are fitted to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used, and `anova.lm` will detect this with an error.

## Note

Versions of R prior to 1.2.0 based F tests on pairwise comparisons, and this behaviour can still be obtained by a direct call to `anova.lm`.

## References

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

The model fitting function [lm](#), [anova](#).

[drop1](#) for so-called 'type II' anova where each term is dropped one at a time respecting their hierarchy.

## Examples

```
## sequential table
fit <- lm(sr ~ ., data = LifeCycleSavings)
anova(fit)

## same effect via separate models
```

```

fit0 <- lm(sr ~ 1, data = LifeCycleSavings)
fit1 <- update(fit0, . ~ . + pop15)
fit2 <- update(fit1, . ~ . + pop75)
fit3 <- update(fit2, . ~ . + dpi)
fit4 <- update(fit3, . ~ . + ddpi)
anova(fit0, fit1, fit2, fit3, fit4, test="F")

anova(fit4, fit2, fit0, test="F") # unconventional order

```

---

ansari.test

*Ansari-Bradley Test*


---

### Description

Performs the Ansari-Bradley two-sample test for a difference in scale parameters.

### Usage

```

ansari.test(x, ...)

## Default S3 method:
ansari.test(x, y, alternative = c("two.sided", "less", "greater"),
            exact = NULL, conf.int = FALSE, conf.level = 0.95, ...)

## S3 method for class 'formula':
ansari.test(formula, data, subset, na.action, ...)

```

### Arguments

x	numeric vector of data values.
y	numeric vector of data values.
alternative	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter.
exact	a logical indicating whether an exact p-value should be computed.
conf.int	a logical, indicating whether a confidence interval should be computed.
conf.level	confidence level of the interval.
formula	a formula of the form <code>lhs ~ rhs</code> where <code>lhs</code> is a numeric variable giving the data values and <code>rhs</code> a factor with two levels giving the corresponding groups.
data	an optional data frame containing the variables in the model formula.
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
...	further arguments to be passed to or from methods.

### Details

Suppose that  $x$  and  $y$  are independent samples from distributions with densities  $f((t - m)/s)/s$  and  $f(t - m)$ , respectively, where  $m$  is an unknown nuisance parameter and  $s$ , the ratio of scales, is the parameter of interest. The Ansari-Bradley test is used for testing the null that  $s$  equals 1, the two-sided alternative being that  $s \neq 1$  (the distributions differ only in variance), and the one-sided alternatives being  $s > 1$  (the distribution underlying  $x$  has a larger variance, "greater") or  $s < 1$  ("less").

By default (if `exact` is not specified), an exact p-value is computed if both samples contain less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Optionally, a nonparametric confidence interval and an estimator for  $s$  are computed. If exact p-values are available, an exact confidence interval is obtained by the algorithm described in Bauer (1972), and the Hodges-Lehmann estimator is employed. Otherwise, the returned confidence interval and point estimate are based on normal approximations.

### Value

A list with class "htest" containing the following components:

<code>statistic</code>	the value of the Ansari-Bradley test statistic.
<code>p.value</code>	the p-value of the test.
<code>null.value</code>	the ratio of scales $s$ under the null, 1.
<code>alternative</code>	a character string describing the alternative hypothesis.
<code>method</code>	the string "Ansari-Bradley test".
<code>data.name</code>	a character string giving the names of the data.
<code>conf.int</code>	a confidence interval for the scale parameter. (Only present if argument <code>conf.int = TRUE</code> .)
<code>estimate</code>	an estimate of the ratio of scales. (Only present if argument <code>conf.int = TRUE</code> .)

### Note

To compare results of the Ansari-Bradley test to those of the F test to compare two variances (under the assumption of normality), observe that  $s$  is the ratio of scales and hence  $s^2$  is the ratio of variances (provided they exist), whereas for the F test the ratio of variances itself is the parameter of interest. In particular, confidence intervals are for  $s$  in the Ansari-Bradley test but for  $s^2$  in the F test.

### References

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 83–92.

David F. Bauer (1972), Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690.

**See Also**

`fligner.test` for a rank-based (nonparametric)  $k$ -sample test for homogeneity of variances; `mood.test` for another rank-based two-sample test for a difference in scale parameters; `var.test` and `bartlett.test` for parametric tests for the homogeneity in variance.

**Examples**

```
## Hollander & Wolfe (1973, p. 86f):
## Serum iron determination using Hyland control sera
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
           101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
                100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
ansari.test(ramsay, jung.parekh)

ansari.test(rnorm(10), rnorm(10, 0, 2), conf.int = TRUE)
```

---

aov

*Fit an Analysis of Variance Model*


---

**Description**

Fit an analysis of variance model by a call to `lm` for each stratum.

**Usage**

```
aov(formula, data = NULL, projections = FALSE, qr = TRUE,
     contrasts = NULL, ...)
```

**Arguments**

<code>formula</code>	A formula specifying the model.
<code>data</code>	A data frame in which the variables specified in the formula will be found. If missing, the variables are searched for in the standard way.
<code>projections</code>	Logical flag: should the projections be returned?
<code>qr</code>	Logical flag: should the QR decomposition be returned?
<code>contrasts</code>	A list of contrasts to be used for some of the factors in the formula. These are not used for any <code>Error</code> term, and supplying contrasts for factors only in the <code>Error</code> term will give a warning.
<code>...</code>	Arguments to be passed to <code>lm</code> , such as <code>subset</code> or <code>na.action</code> .

## Details

This provides a wrapper to `lm` for fitting linear models to balanced or unbalanced experimental designs.

The main difference from `lm` is in the way `print`, `summary` and so on handle the fit: this is expressed in the traditional language of the analysis of variance rather than that of linear models.

If the formula contains a single `ERROR` term, this is used to specify error strata, and appropriate models are fitted within each error stratum.

The formula can specify multiple responses.

Weights can be specified by a `weights` argument, but should not be used with an `ERROR` term, and are incompletely supported (e.g., not by `model.tables`).

## Value

An object of class `c("aov", "lm")` or for multiple responses of class `c("maov", "aov", "mlm", "lm")` or for multiple error strata of class `"aovlist"`. There are `print` and `summary` methods available for these.

## Note

`aov` is designed for balanced designs, and the results can be hard to interpret without balance: beware that missing values in the response(s) will likely lose the balance. If there are two or more error strata, the methods used are statistically inefficient without balance, and it may be better to use `lme`.

Balance can be checked with the `replications` function.

## Author(s)

The design was inspired by the `S` function of the same name described in Chambers *et al.* (1992).

## References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments*. Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

`lm`, `summary.aov`, `replications`, `alias`, `proj`, `model.tables`, `TukeyHSD`

## Examples

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,55.0,
          62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)
npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
```

```

K=factor(K), yield=yield)

op <- options(contrasts=c("contr.helmert", "contr.treatment"))
( npk.aov <- aov(yield ~ block + N*P*K, npk) )
summary(npk.aov)
coefficients(npk.aov)

## to show the effects of re-ordering terms contrast the two fits
aov(yield ~ block + N * P + K, npk)
aov(terms(yield ~ block + N * P + K, keep.order=TRUE), npk)

## as a test, not particularly sensible statistically
npk.aovE <- aov(yield ~ N*P*K + Error(block), npk)
npk.aovE
summary(npk.aovE)
options(op)# reset to previous

```

---

ar

*Fit Autoregressive Models to Time Series*


---

## Description

Fit an autoregressive time series model to the data, by default selecting the complexity by AIC.

## Usage

```

ar(x, aic = TRUE, order.max = NULL,
   method=c("yule-walker", "burg", "ols", "mle", "yw"), na.action,
   series, ...)

ar.burg(x, ...)
## Default S3 method:
ar.burg(x, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series,
        var.method = 1, ...)
## S3 method for class 'mts':
ar.burg(x, aic = TRUE, order.max = NULL,
        na.action = na.fail, demean = TRUE, series,
        var.method = 1, ...)

ar.yw(x, ...)
## Default S3 method:
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series, ...)
## S3 method for class 'mts':
ar.yw(x, aic = TRUE, order.max = NULL,
      na.action = na.fail, demean = TRUE, series, var.method = 1, ...)

```

```
ar.mle(x, aic = TRUE, order.max = NULL, na.action = na.fail,
       demean = TRUE, series, ...)
```

```
## S3 method for class 'ar':
predict(object, newdata, n.ahead = 1, se.fit = TRUE, ...)
```

### Arguments

<code>x</code>	A univariate or multivariate time series.
<code>aic</code>	Logical flag. If <code>TRUE</code> then the Akaike Information Criterion is used to choose the order of the autoregressive model. If <code>FALSE</code> , the model of order <code>order.max</code> is fitted.
<code>order.max</code>	Maximum order (or order) of model to fit. Defaults to $10 \log_{10}(N)$ where $N$ is the number of observations except for <code>method="mle"</code> where it is the minimum of this quantity and 12.
<code>method</code>	Character string giving the method used to fit the model. Must be one of the strings in the default argument (the first few characters are sufficient). Defaults to "yule-walker".
<code>na.action</code>	function to be called to handle missing values.
<code>demean</code>	should a mean be estimated during fitting?
<code>series</code>	names for the series. Defaults to <code>deparse(substitute(x))</code> .
<code>var.method</code>	the method to estimate the innovations variance (see Details).
<code>...</code>	additional arguments for specific methods.
<code>object</code>	a fit from <code>ar</code> .
<code>newdata</code>	data to which to apply the prediction.
<code>n.ahead</code>	number of steps ahead at which to predict.
<code>se.fit</code>	logical: return estimated standard errors of the prediction error?

### Details

For definiteness, note that the AR coefficients have the sign in

$$x_t - \mu = a_1(x_{t-1} - \mu) + \dots + a_p(x_{t-p} - \mu) + e_t$$

`ar` is just a wrapper for the functions `ar.yw`, `ar.burg`, [ar.ols](#) and `ar.mle`.

Order selection is done by AIC if `aic` is true. This is problematic, as of the methods here only `ar.mle` performs true maximum likelihood estimation. The AIC is computed as if the variance estimate were the MLE, omitting the determinant term from the likelihood. Note that this is not the same as the Gaussian likelihood evaluated at the estimated parameter values. In `ar.yw` the variance matrix of the innovations is computed from the fitted coefficients and the autocovariance of `x`.

`ar.burg` allows two methods to estimate the innovations variance and hence AIC. Method 1 is to use the update given by the Levinson-Durbin recursion (Brockwell and Davis, 1991, (8.2.6) on page 242), and follows S-PLUS. Method 2 is the mean of the sum of squares of the forward and

backward prediction errors (as in Brockwell and Davis, 1996, page 145). Percival and Walden (1998) discuss both. In the multivariate case the estimated coefficients will depend (slightly) on the variance estimation method.

Remember that `ar` includes by default a constant in the model, by removing the overall mean of `x` before fitting the AR model, or (`ar.mle`) estimating a constant to subtract.

### Value

For `ar` and its methods a list of class "ar" with the following elements:

<code>order</code>	The order of the fitted model. This is chosen by minimizing the AIC if <code>aic=TRUE</code> , otherwise it is <code>order.max</code> .
<code>ar</code>	Estimated autoregression coefficients for the fitted model.
<code>var.pred</code>	The prediction variance: an estimate of the portion of the variance of the time series that is not explained by the autoregressive model.
<code>x.mean</code>	The estimated mean of the series used in fitting and for use in prediction.
<code>x.intercept</code>	( <code>ar.ols</code> only.) The intercept in the model for $x - x.mean$ .
<code>aic</code>	The value of the <code>aic</code> argument.
<code>n.used</code>	The number of observations in the time series.
<code>order.max</code>	The value of the <code>order.max</code> argument.
<code>partialacf</code>	The estimate of the partial autocorrelation function up to lag <code>order.max</code> .
<code>resid</code>	residuals from the fitted model, conditioning on the first <code>order</code> observations. The first <code>order</code> residuals are set to NA. If <code>x</code> is a time series, so is <code>resid</code> .
<code>method</code>	The value of the <code>method</code> argument.
<code>series</code>	The name(s) of the time series.
<code>asy.var.coef</code>	(univariate case.) The asymptotic-theory variance matrix of the coefficient estimates.

For `predict.ar`, a time series of predictions, or if `se.fit = TRUE`, a list with components `pred`, the predictions, and `se`, the estimated standard errors. Both components are time series.

### Note

Only the univariate case of `ar.mle` is implemented.

Fitting by `method="mle"` to long series can be very slow.

### Author(s)

Martyn Plummer. Univariate case of `ar.yw`, `ar.mle` and C code for univariate case of `ar.burg` by B. D. Ripley.

## References

- Brockwell, P. J. and Davis, R. A. (1991) *Time Series and Forecasting Methods*. Second edition. Springer, New York. Section 11.4.
- Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting*. Springer, New York. Sections 5.1 and 7.6.
- Percival, D. P. and Walden, A. T. (1998) *Spectral Analysis for Physical Applications*. Cambridge University Press.
- Whittle, P. (1963) On the fitting of multivariate autoregressions and the approximate canonical factorization of a spectral density matrix. *Biometrika* **40**, 129–134.

## See Also

[ar.ols](#), [arima0](#) for ARMA models.

## Examples

```
ar(lh)
ar(lh, method="burg")
ar(lh, method="ols")
ar(lh, FALSE, 4) # fit ar(4)

(sunspot.ar <- ar(sunspot.year))
predict(sunspot.ar, n.ahead=25)
## try the other methods too

ar(ts.union(BJsales, BJsales.lead))
## Burg is quite different here, as is OLS (see ar.ols)
ar(ts.union(BJsales, BJsales.lead), method="burg")
```

---

ar.ols

*Fit Autoregressive Models to Time Series by OLS*

---

## Description

Fit an autoregressive time series model to the data by ordinary least squares, by default selecting the complexity by AIC.

## Usage

```
ar.ols(x, aic = TRUE, order.max = NULL, na.action = na.fail,
       demean = TRUE, intercept = demean, series, ...)
```

**Arguments**

<code>x</code>	A univariate or multivariate time series.
<code>aic</code>	Logical flag. If <code>TRUE</code> then the Akaike Information Criterion is used to choose the order of the autoregressive model. If <code>FALSE</code> , the model of order <code>order.max</code> is fitted.
<code>order.max</code>	Maximum order (or order) of model to fit. Defaults to $10 \log_{10}(N)$ where $N$ is the number of observations.
<code>na.action</code>	function to be called to handle missing values.
<code>demean</code>	should the AR model be for <code>x</code> minus its mean?
<code>intercept</code>	should a separate intercept term be fitted?
<code>series</code>	names for the series. Defaults to <code>deparse(substitute(x))</code> .
<code>...</code>	further arguments to be passed to or from methods.

**Details**

`ar.ols` fits the general AR model to a possibly non-stationary and/or multivariate system of series `x`. The resulting unconstrained least squares estimates are consistent, even if some of the series are non-stationary and/or co-integrated. For definiteness, note that the AR coefficients have the sign in

$$x_t - \mu = a_0 + a_1(x_{t-1} - \mu) + \dots + a_p(x_{t-p} - \mu) + e_t$$

where  $a_0$  is zero unless `intercept` is true, and  $\mu$  is the sample mean if `demean` is true, zero otherwise.

Order selection is done by AIC if `aic` is true. This is problematic, as `ar.ols` does not perform true maximum likelihood estimation. The AIC is computed as if the variance estimate (computed from the variance matrix of the residuals) were the MLE, omitting the determinant term from the likelihood. Note that this is not the same as the Gaussian likelihood evaluated at the estimated parameter values.

Some care is needed if `intercept` is true and `demean` is false. Only use this if the series are roughly centred on zero. Otherwise the computations may be inaccurate or fail entirely.

**Value**

A list of class "ar" with the following elements:

<code>order</code>	The order of the fitted model. This is chosen by minimizing the AIC if <code>aic=TRUE</code> , otherwise it is <code>order.max</code> .
<code>ar</code>	Estimated autoregression coefficients for the fitted model.
<code>var.pred</code>	The prediction variance: an estimate of the portion of the variance of the time series that is not explained by the autoregressive model.
<code>x.mean</code>	The estimated mean (or zero if <code>demean</code> is false) of the series used in fitting and for use in prediction.
<code>x.intercept</code>	The intercept in the model for <code>x</code> - <code>x.mean</code> , or zero if <code>intercept</code> is false.
<code>aic</code>	The value of the <code>aic</code> argument.

<code>n.used</code>	The number of observations in the time series.
<code>order.max</code>	The value of the <code>order.max</code> argument.
<code>partialacf</code>	NULL. For compatibility with <code>ar</code> .
<code>resid</code>	residuals from the fitted model, conditioning on the first order observations. The first order residuals are set to NA. If <code>x</code> is a time series, so is <code>resid</code> .
<code>method</code>	The character string "Unconstrained LS".
<code>series</code>	The name(s) of the time series.
<code>asy.se.coef</code>	The asymptotic-theory standard errors of the coefficient estimates.

**Author(s)**

Adrian Trapletti, Brian Ripley.

**References**

Luetkepohl, H. (1991): *Introduction to Multiple Time Series Analysis*. Springer Verlag, NY, pp. 368–370.

**See Also**

[ar](#)

**Examples**

```
ar(lh, method="burg")
ar.ols(lh)
ar.ols(lh, FALSE, 4) # fit ar(4)

ar.ols(ts.union(BJsales, BJsales.lead))

x <- diff(log(EuStockMarkets))
ar.ols(x, order.max=6, demean=FALSE, intercept=TRUE)
```

---

arima

*ARIMA Modelling of Time Series*


---

**Description**

Fit an ARIMA model to a univariate time series.

**Usage**

```
arima(x, order = c(0, 0, 0),
      seasonal = list(order = c(0, 0, 0), period = NA),
      xreg = NULL, include.mean = TRUE, transform.pars = TRUE,
      fixed = NULL, init = NULL, method = c("CSS-ML", "ML", "CSS"),
      n.cond, optim.control = list(), kappa = 1e6)
```

**Arguments**

<code>x</code>	a univariate time series
<code>order</code>	A specification of the non-seasonal part of the ARIMA model: the three components $(p, d, q)$ are the AR order, the degree of differencing, and the MA order.
<code>seasonal</code>	A specification of the seasonal part of the ARIMA model, plus the period (which defaults to <code>frequency(x)</code> ). This should be a list with components <code>order</code> and <code>period</code> , but a specification of just a numeric vector of length 3 will be turned into a suitable list with the specification as the <code>order</code> .
<code>xreg</code>	Optionally, a vector or matrix of external regressors, which must have the same number of rows as <code>x</code> .
<code>include.mean</code>	Should the ARIMA model include a mean term? The default is <code>TRUE</code> for undifferenced series, <code>FALSE</code> for differenced ones (where a mean would not affect the fit nor predictions).
<code>transform.pars</code>	Logical. If true, the AR parameters are transformed to ensure that they remain in the region of stationarity. Not used for <code>method = "CSS"</code> .
<code>fixed</code>	optional numeric vector of the same length as the total number of parameters. If supplied, only NA entries in <code>fixed</code> will be varied. <code>transform.pars = TRUE</code> will be overridden (with a warning) if any AR parameters are fixed. It may be wise to set <code>transform.pars = FALSE</code> when fixing MA parameters, especially near non-invertibility.
<code>init</code>	optional numeric vector of initial parameter values. Missing values will be filled in, by zeroes except for regression coefficients. Values already specified in <code>fixed</code> will be ignored.
<code>method</code>	Fitting method: maximum likelihood or minimize conditional sum-of-squares. The default (unless there are missing values) is to use conditional-sum-of-squares to find starting values, then maximum likelihood.
<code>n.cond</code>	Only used if fitting by conditional-sum-of-squares: the number of initial observations to ignore. It will be ignored if less than the maximum lag of an AR term.
<code>optim.control</code>	List of control parameters for <code>optim</code> .
<code>kappa</code>	the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model. Do not reduce this.

**Details**

Different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition here has

$$X_t = a_1 X_{t-1} + \dots + a_p X_{t-p} + e_t + b_1 e_{t-1} + \dots + b_q e_{t-q}$$

and so the MA coefficients differ in sign from those of S-PLUS. Further, if `include.mean` is true, this formula applies to  $X - m$  rather than  $X$ . For ARIMA models with differencing, the differenced

series follows a zero-mean ARMA model. If a `xreg` term is included, a linear regression (with a constant term if `include.mean` is true) is fitted with an ARMA model for the error term.

The variance matrix of the estimates is found from the Hessian of the log-likelihood, and so may only be a rough guide.

Optimization is done by `optim`. It will work best if the columns in `xreg` are roughly scaled to zero mean and unit variance, but does attempt to estimate suitable scalings.

## Value

A list of class "Arima" with components:

<code>coef</code>	a vector of AR, MA and regression coefficients, which can be extracted by the <code>coef</code> method.
<code>sigma2</code>	the MLE of the innovations variance.
<code>var.coef</code>	the estimated variance matrix of the coefficients <code>coef</code> , which can be extracted by the <code>vcov</code> method.
<code>loglik</code>	the maximized log-likelihood (of the differenced data), or the approximation to it used.
<code>arma</code>	A compact form of the specification, as a vector giving the number of AR, MA, seasonal AR and seasonal MA coefficients, plus the period and the number of non-seasonal and seasonal differences.
<code>aic</code>	the AIC value corresponding to the log-likelihood. Only valid for <code>method = "ML"</code> fits.
<code>residuals</code>	the fitted innovations.
<code>call</code>	the matched call.
<code>series</code>	the name of the series <code>x</code> .
<code>code</code>	the convergence value returned by <code>optim</code> .
<code>n.cond</code>	the number of initial observations not used in the fitting.
<code>model</code>	A list representing the Kalman Filter used in the fitting. See <code>KalmanLike</code> .

## Fitting methods

The exact likelihood is computed via a state-space representation of the ARIMA process, and the innovations and their variance found by a Kalman filter. The initialization of the differenced ARMA process uses stationarity and is based on Gardner *et al.* (1980). For a differenced process the non-stationary components are given a diffuse prior (controlled by `kappa`). Observations which are still controlled by the diffuse prior (determined by having a Kalman gain of at least  $1e4$ ) are excluded from the likelihood calculations. (This gives comparable results to `arima0` in the absence of missing values, when the observations excluded are precisely those dropped by the differencing.)

Missing values are allowed, and are handled exactly in method "ML".

If `transform.pars` is true, the optimization is done using an alternative parametrization which is a variation on that suggested by Jones (1980) and ensures that the model is stationary. For an AR(p) model the parametrization is via the inverse tanh of the partial autocorrelations: the same procedure is applied (separately) to the AR and seasonal AR terms. The MA terms are not

constrained to be invertible during optimization, but they will be converted to invertible form after optimization if `transform.pars` is true.

Conditional sum-of-squares is provided mainly for expositional purposes. This computes the sum of squares of the fitted innovations from observation `n.cond` on, (where `n.cond` is at least the maximum lag of an AR term), treating all earlier innovations to be zero. Argument `n.cond` can be used to allow comparability between different fits. The “part log-likelihood” is the first term, half the log of the estimated mean square. Missing values are allowed, but will cause many of the innovations to be missing.

When regressors are specified, they are orthogonalized prior to fitting unless any of the coefficients is fixed. It can be helpful to roughly scale the regressors to zero mean and unit variance.

### Note

The results are likely to be different from S-PLUS’s `arima.mle`, which computes a conditional likelihood and does not include a mean in the model. Further, the convention used by `arima.mle` reverses the signs of the MA coefficients.

`arima` is very similar to `arima0` for ARMA models or for differenced models without missing values, but handles differenced models with missing values exactly. It is somewhat slower than `arima0`, particularly for seasonally differenced models.

### References

- Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting*. Springer, New York. Sections 3.3 and 8.3.
- Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press.
- Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980) Algorithm AS154. An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics* **29**, 311–322.
- Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.
- Jones, R. H. (1980) Maximum likelihood fitting of ARMA models to time series with missing observations. *Technometrics* **20** 389–395.

### See Also

`predict.Arima`, `arima.sim` for simulating from an ARIMA model, `tsdiag`, `arima0`, `ar`

### Examples

```
arima(lh, order = c(1,0,0))
arima(lh, order = c(3,0,0))
arima(lh, order = c(1,0,1))

arima(lh, order = c(3,0,0), method = "CSS")

arima(USAccDeaths, order = c(0,1,1), seasonal = list(order=c(0,1,1)))
arima(USAccDeaths, order = c(0,1,1), seasonal = list(order=c(0,1,1)),
      method = "CSS") # drops first 13 observations.
```

```
# for a model with as few years as this, we want full ML
arima(LakeHuron, order = c(2,0,0), xreg = time(LakeHuron)-1920)

## presidents contains NAs
## graphs in example(acf) suggest order 1 or 3
(fit1 <- arima(presidents, c(1, 0, 0)))
tsdiag(fit1)
(fit3 <- arima(presidents, c(3, 0, 0))) # smaller AIC
tsdiag(fit3)
```

---

arima.sim

*Simulate from an ARIMA Model*


---

### Description

Simulate from an ARIMA model.

### Usage

```
arima.sim(model, n, rand.gen = rnorm, innov = rand.gen(n, ...),
          n.start = NA, ...)
```

### Arguments

model	A list with component <code>ar</code> and/or <code>ma</code> giving the AR and MA coefficients respectively. Optionally a component <code>order</code> can be used. An empty list gives an ARIMA(0, 0, 0) model, that is white noise.
n	length of output series, before un-differencing.
rand.gen	optional: a function to generate the innovations.
innov	an optional times series of innovations. If not provided, <code>rand.gen</code> is used.
n.start	length of “burn-in” period. If NA, the default, a reasonable value is computed.
...	additional arguments for <code>rand.gen</code> . Most usefully, the standard deviation of the innovations generated by <code>rnorm</code> can be specified by <code>sd</code> .

### Details

See [arima](#) for the precise definition of an ARIMA model.

The ARMA model is checked for stationarity.

ARIMA models are specified via the `order` component of `model`, in the same way as for [arima](#). Other aspects of the `order` component are ignored, but inconsistent specifications of the MA and AR orders are detected. The un-differencing assumes previous values of zero, and to remind the user of this, those values are returned.

### Value

A time-series object of class “ts”.

**See Also**[arima](#)**Examples**

```

arima.sim(n = 63, list(ar = c(0.8897, -0.4858), ma = c(-0.2279, 0.2488)),
          sd = sqrt(0.1796))
# mildly long-tailed
arima.sim(n = 63, list(ar=c(0.8897, -0.4858), ma=c(-0.2279, 0.2488)),
          rand.gen = function(n, ...) sqrt(0.1796) * rt(n, df = 5))

# An ARIMA simulation
ts.sim <- arima.sim(list(order = c(1,1,0), ar = 0.7), n = 200)
ts.plot(ts.sim)

```

arima0

*ARIMA Modelling of Time Series – Preliminary Version***Description**

Fit an ARIMA model to a univariate time series, and forecast from the fitted model.

**Usage**

```

arima0(x, order = c(0, 0, 0),
       seasonal = list(order = c(0, 0, 0), period = NA),
       xreg = NULL, include.mean = TRUE, delta = 0.01,
       transform.pars = TRUE, fixed = NULL, init = NULL,
       method = c("ML", "CSS"), n.cond, optim.control = list())

## S3 method for class 'arima0':
predict(object, n.ahead = 1, newxreg, se.fit = TRUE, ...)

```

**Arguments**

<code>x</code>	a univariate time series
<code>order</code>	A specification of the non-seasonal part of the ARIMA model: the three components $(p, d, q)$ are the AR order, the degree of differencing, and the MA order.
<code>seasonal</code>	A specification of the seasonal part of the ARIMA model, plus the period (which defaults to <code>frequency(x)</code> ). This should be a list with components <code>order</code> and <code>period</code> , but a specification of just a numeric vector of length 3 will be turned into a suitable list with the specification as the <code>order</code> .
<code>xreg</code>	Optionally, a vector or matrix of external regressors, which must have the same number of rows as <code>x</code> .
<code>include.mean</code>	Should the ARIMA model include a mean term? The default is <code>TRUE</code> for undifferenced series, <code>FALSE</code> for differenced ones (where a mean would not affect the fit nor predictions).

<code>delta</code>	A value to indicate at which point ‘fast recursions’ should be used. See the Details section.
<code>transform.pars</code>	Logical. If true, the AR parameters are transformed to ensure that they remain in the region of stationarity. Not used for <code>method = "CSS"</code> .
<code>fixed</code>	optional numeric vector of the same length as the total number of parameters. If supplied, only NA entries in <code>fixed</code> will be varied. <code>transform.pars = TRUE</code> will be overridden (with a warning) if any ARMA parameters are fixed.
<code>init</code>	optional numeric vector of initial parameter values. Missing values will be filled in, by zeroes except for regression coefficients. Values already specified in <code>fixed</code> will be ignored.
<code>method</code>	Fitting method: maximum likelihood or minimize conditional sum-of-squares.
<code>n.cond</code>	Only used if fitting by conditional-sum-of-squares: the number of initial observations to ignore. It will be ignored if less than the maximum lag of an AR term.
<code>optim.control</code>	List of control parameters for <a href="#">optim</a> .
<code>object</code>	The result of an <code>arima0</code> fit.
<code>newxreg</code>	New values of <code>xreg</code> to be used for prediction. Must have at least <code>n.ahead</code> rows.
<code>n.ahead</code>	The number of steps ahead for which prediction is required.
<code>se.fit</code>	Logical: should standard errors of prediction be returned?
<code>...</code>	arguments passed to or from other methods.

## Details

Different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition here has

$$X_t = a_1 X_{t-1} + \dots + a_p X_{t-p} + e_t + b_1 e_{t-1} + \dots + b_q e_{t-q}$$

and so the MA coefficients differ in sign from those of S-PLUS. Further, if `include.mean` is true, this formula applies to  $X - m$  rather than  $X$ . For ARIMA models with differencing, the differenced series follows a zero-mean ARMA model.

The variance matrix of the estimates is found from the Hessian of the log-likelihood, and so may only be a rough guide, especially for fits close to the boundary of invertibility.

Optimization is done by [optim](#). It will work best if the columns in `xreg` are roughly scaled to zero mean and unit variance, but does attempt to estimate suitable scalings.

Finite-history prediction is used. This is only statistically efficient if the MA part of the fit is invertible, so `predict.arima0` will give a warning for non-invertible MA models.

**Value**

For `arima0`, a list of class "arima0" with components:

<code>coef</code>	a vector of AR, MA and regression coefficients,
<code>sigma2</code>	the MLE of the innovations variance.
<code>var.coef</code>	the estimated variance matrix of the coefficients <code>coef</code> .
<code>loglik</code>	the maximized log-likelihood (of the differenced data), or the approximation to it used.
<code>arma</code>	A compact form of the specification, as a vector giving the number of AR, MA, seasonal AR and seasonal MA coefficients, plus the period and the number of non-seasonal and seasonal differences.
<code>aic</code>	the AIC value corresponding to the log-likelihood. Only valid for <code>method = "ML"</code> fits.
<code>residuals</code>	the fitted innovations.
<code>call</code>	the matched call.
<code>series</code>	the name of the series <code>x</code> .
<code>convergence</code>	the value returned by <code>optim</code> .
<code>n.cond</code>	the number of initial observations not used in the fitting.

For `predict.arima0`, a time series of predictions, or if `se.fit = TRUE`, a list with components `pred`, the predictions, and `se`, the estimated standard errors. Both components are time series.

**Fitting methods**

The exact likelihood is computed via a state-space representation of the ARMA process, and the innovations and their variance found by a Kalman filter based on Gardner *et al.* (1980). This has the option to switch to 'fast recursions' (assume an effectively infinite past) if the innovations variance is close enough to its asymptotic bound. The argument `delta` sets the tolerance: at its default value the approximation is normally negligible and the speed-up considerable. Exact computations can be ensured by setting `delta` to a negative value.

If `transform.pars` is true, the optimization is done using an alternative parametrization which is a variation on that suggested by Jones (1980) and ensures that the model is stationary. For an AR(p) model the parametrization is via the inverse tanh of the partial autocorrelations: the same procedure is applied (separately) to the AR and seasonal AR terms. The MA terms are also constrained to be invertible during optimization by the same transformation if `transform.pars` is true. Note that the MLE for MA terms does sometimes occur for MA polynomials with unit roots: such models can be fitted by using `transform.pars = FALSE` and specifying a good set of initial values (often obtainable from a fit with `transform.pars = TRUE`).

As from R 1.5.0 missing values are allowed, but any missing values will force `delta` to be ignored and full recursions used. Note that missing values will be propagated by differencing, so the procedure used in this function is not fully efficient in that case.

Conditional sum-of-squares is provided mainly for expositional purposes. This computes the sum of squares of the fitted innovations from observation `n.cond` on, (where `n.cond` is at least the

maximum lag of an AR term), treating all earlier innovations to be zero. Argument `n.cond` can be used to allow comparability between different fits. The “part log-likelihood” is the first term, half the log of the estimated mean square. Missing values are allowed, but will cause many of the innovations to be missing.

When regressors are specified, they are orthogonalized prior to fitting unless any of the coefficients is fixed. It can be helpful to roughly scale the regressors to zero mean and unit variance.

## Note

This is a preliminary version, and will be replaced by [arima](#).

The standard errors of prediction exclude the uncertainty in the estimation of the ARMA model and the regression coefficients.

The results are likely to be different from S-PLUS’s `arima.mle`, which computes a conditional likelihood and does not include a mean in the model. Further, the convention used by `arima.mle` reverses the signs of the MA coefficients.

## References

Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting*. Springer, New York. Sections 3.3 and 8.3.

Gardner, G, Harvey, A. C. and Phillips, G. D. A. (1980) Algorithm AS154. An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering. *Applied Statistics* **29**, 311–322.

Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.

Harvey, A. C. and McKenzie, C. R. (1982) Algorithm AS182. An algorithm for finite sample prediction from ARIMA processes. *Applied Statistics* **31**, 180–187.

Jones, R. H. (1980) Maximum likelihood fitting of ARMA models to time series with missing observations. *Technometrics* **20** 389–395.

## See Also

[arima](#), [ar](#), [tsdiag](#)

## Examples

```
## Not run: arima0(lh, order = c(1,0,0))
arima0(lh, order = c(3,0,0))
arima0(lh, order = c(1,0,1))
predict(arima0(lh, order = c(3,0,0)), n.ahead = 12)

arima0(lh, order = c(3,0,0), method = "CSS")

# for a model with as few years as this, we want full ML
(fit <- arima0(USAccDeaths, order = c(0,1,1),
              seasonal = list(order=c(0,1,1)), delta = -1))
predict(fit, n.ahead = 6)

arima0(LakeHuron, order = c(2,0,0), xreg = time(LakeHuron)-1920)
```

```
## Not run:
## presidents contains NAs
## graphs in example(acf) suggest order 1 or 3
(fit1 <- arima0(presidents, c(1, 0, 0), delta = -1)) # avoid warning
tsdiag(fit1)
(fit3 <- arima0(presidents, c(3, 0, 0), delta = -1)) # smaller AIC
tsdiag(fit3)
## End(Not run)
```

---

ARMAacf

*Compute Theoretical ACF for an ARMA Process*


---

### Description

Compute the theoretical autocorrelation function or partial autocorrelation function for an ARMA process.

### Usage

```
ARMAacf(ar = numeric(0), ma = numeric(0), lag.max = r, pacf = FALSE)
```

### Arguments

<code>ar</code>	numeric vector of AR coefficients
<code>ma</code>	numeric vector of MA coefficients
<code>lag.max</code>	integer. Maximum lag required. Defaults to $\max(p, q+1)$ , where $p$ , $q$ are the numbers of AR and MA terms respectively.
<code>pacf</code>	logical. Should the partial autocorrelations be returned?

### Details

The methods used follow Brockwell & Davis (1991, section 3.3). Their equations (3.3.8) are solved for the autocovariances at lags  $0, \dots, \max(p, q + 1)$ , and the remaining autocorrelations are given by a recursive filter.

### Value

A vector of (partial) autocorrelations, named by the lags.

### References

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*, Second Edition. Springer.

### See Also

[arima](#), [ARMAtoMA](#), [filter](#).

**Examples**

```
ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10)
## Example from Brockwell & Davis (1991, pp.92-4)
## answer  $2^{-n} * (32/3 + 8 * n) / (32/3)$ 
n <- 1:10; 2^(-n) * (32/3 + 8 * n) / (32/3)
ARMAacf(c(1.0, -0.25), 1.0, lag.max = 10, pacf = TRUE)
ARMAacf(c(1.0, -0.25), lag.max = 10, pacf = TRUE)
```

ARMAtoMA

*Convert ARMA Process to Infinite MA Process***Description**

Convert ARMA process to infinite MA process.

**Usage**

```
ARMAtoMA(ar = numeric(0), ma = numeric(0), lag.max)
```

**Arguments**

ar	numeric vector of AR coefficients
ma	numeric vector of MA coefficients
lag.max	Largest MA(Inf) coefficient required.

**Value**

A vector of coefficients.

**References**

Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*, Second Edition. Springer.

**See Also**

[arima](#), [ARMAacf](#).

**Examples**

```
ARMAtoMA(c(1.0, -0.25), 1.0, 10)
## Example from Brockwell & Davis (1991, p.92)
## answer  $(1 + 3*n)*2^{-n}$ 
n <- 1:10; (1 + 3*n)*2^(-n)
```

---

`as.hclust`*Convert Objects to Class hclust*

---

**Description**

Converts objects from other hierarchical clustering functions to class "hclust".

**Usage**

```
as.hclust(x, ...)
```

**Arguments**

<code>x</code>	Hierarchical clustering object
<code>...</code>	further arguments passed to or from other methods.

**Details**

Currently there is only support for converting objects of class "twins" as produced by the functions `diana` and `agnes` from the package **cluster**. The default method throws an error unless passed an "hclust" object.

**Value**

An object of class "hclust".

**See Also**

[hclust](#), [diana](#), [agnes](#)

**Examples**

```
x <- matrix(rnorm(30), ncol=3)
hc <- hclust(dist(x), method="complete")

if(require(cluster, quietly=TRUE)) {# is a recommended package
  ag <- agnes(x, method="complete")
  hcag <- as.hclust(ag)
  ## The dendrograms order slightly differently:
  op <- par(mfrow=c(1,2))
  plot(hc) ; mtext("hclust", side=1)
  plot(hcag); mtext("agnes", side=1)
}
```

---

asOneSidedFormula *Convert to One-Sided Formula*

---

**Description**

Names, expressions, numeric values, and character strings are converted to one-sided formulas. If `object` is a formula, it must be one-sided, in which case it is returned unaltered.

**Usage**

```
asOneSidedFormula(object)
```

**Arguments**

`object` a one-sided formula, an expression, a numeric value, or a character string.

**Value**

a one-sided formula representing `object`

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[formula](#)

**Examples**

```
asOneSidedFormula("age")
asOneSidedFormula(~ age)
```

---

ave *Group Averages Over Level Combinations of Factors*

---

**Description**

Subsets of `x[ ]` are averaged, where each subset consist of those observations with the same factor levels.

**Usage**

```
ave(x, ..., FUN = mean)
```

**Arguments**

`x` A numeric.  
`...` Grouping variables, typically factors, all of the same length as `x`.  
`FUN` Function to apply for each factor level combination.

**Value**

A numeric vector, say `y` of length `length(x)`. If `...` is `g1,g2`, e.g., `y[i]` is equal to `FUN(x[j], for all j with g1[j] == g1[i] and g2[j] == g2[i])`.

**See Also**

[mean](#), [median](#).

**Examples**

```
ave(1:3)# no grouping -> grand mean

attach(warpbreaks)
ave(breaks, wool)
ave(breaks, tension)
ave(breaks, tension, FUN = function(x)mean(x, trim=.1))
plot(breaks, main =
      "ave( Warpbreaks ) for wool x tension combinations")
lines(ave(breaks, wool, tension
          ), type='s', col = "blue")
lines(ave(breaks, wool, tension, FUN=median), type='s', col = "green")
legend(40,70, c("mean","median"), lty=1,col=c("blue","green"), bg="gray90")
detach()
```

---

bandwidth

*Bandwidth Selectors for Kernel Density Estimation*

---

**Description**

Bandwidth selectors for gaussian windows in [density](#).

**Usage**

```
bw.nrd0(x)
bw.nrd(x)
bw.ucv(x, nb = 1000, lower, upper)
bw.bcv(x, nb = 1000, lower, upper)
bw.SJ(x, nb = 1000, lower, upper, method = c("ste", "dpi"))
```

**Arguments**

<code>x</code>	A data vector.
<code>nb</code>	number of bins to use.
<code>lower</code> , <code>upper</code>	Range over which to minimize. The default is almost always satisfactory.
<code>method</code>	Either <code>"ste"</code> ("solve-the-equation") or <code>"dpi"</code> ("direct plug-in").

**Details**

`bw.nrd0` implements a rule-of-thumb for choosing the bandwidth of a Gaussian kernel density estimator. It defaults to 0.9 times the minimum of the standard deviation and the interquartile range divided by 1.34 times the sample size to the negative one-fifth power (= Silverman's "rule of thumb", Silverman (1986, page 48, eqn (3.31)) *unless* the quartiles coincide when a positive result will be guaranteed.

`bw.nrd` is the more common variation given by Scott (1992), using factor 1.06.

`bw.ucv` and `bw.bcv` implement unbiased and biased cross-validation respectively.

`bw.SJ` implements the methods of Sheather & Jones (1991) to select the bandwidth using pilot estimation of derivatives.

**Value**

A bandwidth on a scale suitable for the `bw` argument of `density`.

**References**

- Scott, D. W. (1992) *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley.
- Sheather, S. J. and Jones, M. C. (1991) A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society series B*, **53**, 683–690.
- Silverman, B. W. (1986) *Density Estimation*. London: Chapman and Hall.
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Springer.

**See Also**

[density](#).

[bandwidth.nrd](#), [ucv](#), [bcv](#) and [width.SJ](#) in package **MASS**, which are all scaled to the `width` argument of `density` and so give answers four times as large.

**Examples**

```
plot(density(precip, n = 1000))
rug(precip)
lines(density(precip, bw="nrd"), col = 2)
lines(density(precip, bw="ucv"), col = 3)
lines(density(precip, bw="bcv"), col = 4)
lines(density(precip, bw="SJ-ste"), col = 5)
lines(density(precip, bw="SJ-dpi"), col = 6)
legend(55, 0.035,
      legend = c("nrd0", "nrd", "ucv", "bcv", "SJ-ste", "SJ-dpi"),
      col = 1:6, lty = 1)
```

---

bartlett.test      *Bartlett Test for Homogeneity of Variances*

---

### Description

Performs Bartlett's test of the null that the variances in each of the groups (samples) are the same.

### Usage

```
bartlett.test(x, ...)

## Default S3 method:
bartlett.test(x, g, ...)

## S3 method for class 'formula':
bartlett.test(formula, data, subset, na.action, ...)
```

### Arguments

<code>x</code>	a numeric vector of data values, or a list of numeric data vectors representing the respective samples, or fitted linear model objects (inheriting from class "lm").
<code>g</code>	a vector or factor object giving the group for the corresponding elements of <code>x</code> . Ignored if <code>x</code> is a list.
<code>formula</code>	a formula of the form <code>lhs ~ rhs</code> where <code>lhs</code> gives the data values and <code>rhs</code> the corresponding groups.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

### Details

If `x` is a list, its elements are taken as the samples or fitted linear models to be compared for homogeneity of variances. In this case, the elements must either all be numeric data vectors or fitted linear model objects, `g` is ignored, and one can simply use `bartlett.test(x)` to perform the test. If the samples are not yet contained in a list, use `bartlett.test(list(x, ...))`.

Otherwise, `x` must be a numeric data vector, and `g` must be a vector or factor object of the same length as `x` giving the group for the corresponding elements of `x`.

### Value

A list of class "htest" containing the following components:

<code>statistic</code>	Bartlett's K-squared test statistic.
------------------------	--------------------------------------

parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Bartlett test for homogeneity of variances".
data.name	a character string giving the names of the data.

## References

Bartlett, M. S. (1937). Properties of sufficiency and statistical tests. *Proceedings of the Royal Statistical Society Series A* **160**, 268–282.

## See Also

[var.test](#) for the special case of comparing variances in two samples from normal distributions; [fligner.test](#) for a rank-based (nonparametric)  $k$ -sample test for homogeneity of variances; [ansari.test](#) and [mood.test](#) for two rank based two-sample tests for difference in scale.

## Examples

```
plot(count ~ spray, data = InsectSprays)
bartlett.test(InsectSprays$count, InsectSprays$spray)
bartlett.test(count ~ spray, data = InsectSprays)
```

---

Beta

*The Beta Distribution*

---

## Description

Density, distribution function, quantile function and random generation for the Beta distribution with parameters `shape1` and `shape2` (and optional non-centrality parameter `ncp`).

## Usage

```
dbeta(x, shape1, shape2, ncp=0, log = FALSE)
pbeta(q, shape1, shape2, ncp=0, lower.tail = TRUE, log.p = FALSE)
qbeta(p, shape1, shape2, lower.tail = TRUE, log.p = FALSE)
rbeta(n, shape1, shape2)
```

## Arguments

<code>x</code> , <code>q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>shape1</code> , <code>shape2</code>	positive parameters of the Beta distribution.

<code>ncp</code>	non-centrality parameter.
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

### Details

The Beta distribution with parameters `shape1 = a` and `shape2 = b` has density

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^a (1-x)^b$$

for  $a > 0$ ,  $b > 0$  and  $0 \leq x \leq 1$  where the boundary values at  $x = 0$  or  $x = 1$  are defined as by continuity (as limits).

`pbeta` is closely related to the incomplete beta function. As defined by Abramowitz and Stegun 6.6.1

$$B_x(a, b) = \int_0^x t^{a-1} (1-t)^{b-1} dt,$$

and 6.6.2  $I_x(a, b) = B_x(a, b)/B(a, b)$  where  $B(a, b) = B_1(a, b)$  is the Beta function ([beta](#)).

$I_x(a, b)$  is `pbeta(x, a, b)`.

### Value

`dbeta` gives the density, `pbeta` the distribution function, `qbeta` the quantile function, and `rbeta` generates random deviates.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. Chapter 6: Gamma and Related Functions.

### See Also

[beta](#) for the Beta function, and [dgamma](#) for the Gamma distribution.

### Examples

```
x <- seq(0, 1, length=21)
dbeta(x, 1, 1)
pbeta(x, 1, 1)
```

---

binom.test	<i>Exact Binomial Test</i>
------------	----------------------------

---

### Description

Performs an exact test of a simple null hypothesis about the probability of success in a Bernoulli experiment.

### Usage

```
binom.test(x, n, p = 0.5,  
           alternative = c("two.sided", "less", "greater"),  
           conf.level = 0.95)
```

### Arguments

x	number of successes, or a vector of length 2 giving the numbers of successes and failures, respectively.
n	number of trials; ignored if x has length 2.
p	hypothesized probability of success.
alternative	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter.
conf.level	confidence level for the returned confidence interval.

### Details

Confidence intervals are obtained by a procedure first given in Clopper and Pearson (1934). This guarantees that the confidence level is at least `conf.level`, but in general does not give the shortest-length confidence intervals.

### Value

A list with class "htest" containing the following components:

statistic	the number of successes.
parameter	the number of trials.
p.value	the p-value of the test.
conf.int	a confidence interval for the probability of success.
estimate	the estimated probability of success.
null.value	the probability of success under the null, p.
alternative	a character string describing the alternative hypothesis.
method	the character string "Exact binomial test".
data.name	a character string giving the names of the data.

**References**

Clopper, C. J. & Pearson, E. S. (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, **26**, 404–413.

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 97–104.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 15–22.

**See Also**

[prop.test](#) for a general (approximate) test for equal or given proportions.

**Examples**

```
## Conover (1971), p. 97f.
## Under (the assumption of) simple Mendelian inheritance, a cross
## between plants of two particular genotypes produces progeny 1/4 of
## which are "dwarf" and 3/4 of which are "giant", respectively.
## In an experiment to determine if this assumption is reasonable, a
## cross results in progeny having 243 dwarf and 682 giant plants.
## If "giant" is taken as success, the null hypothesis is that p =
## 3/4 and the alternative that p != 3/4.
binom.test(c(682, 243), p = 3/4)
binom.test(682, 682 + 243, p = 3/4) # The same.
## => Data are in agreement with the null hypothesis.
```

---

 Binomial

*The Binomial Distribution*


---

**Description**

Density, distribution function, quantile function and random generation for the binomial distribution with parameters *size* and *prob*.

**Usage**

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

**Arguments**

*x*, *q*            vector of quantiles.  
*p*                vector of probabilities.  
*n*                number of observations. If `length(n) > 1`, the length is taken to be the number required.

<code>size</code>	number of trials.
<code>prob</code>	probability of success on each trial.
<code>log, log.p</code>	logical; if TRUE, probabilities $p$ are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

### Details

The binomial distribution with `size = n` and `prob = p` has density

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

for  $x = 0, \dots, n$ .

If an element of `x` is not integer, the result of `dbinom` is zero, with a warning.  $p(x)$  is computed using Loader's algorithm, see the reference below.

The quantile is defined as the smallest value  $x$  such that  $F(x) \geq p$ , where  $F$  is the distribution function.

### Value

`dbinom` gives the density, `pbinom` gives the distribution function, `qbinom` gives the quantile function and `rbinom` generates random deviates.

If `size` is not an integer, NaN is returned.

### References

Catherine Loader (2000). *Fast and Accurate Computation of Binomial Probabilities*; manuscript available from <http://cm.bell-labs.com/cm/ms/departments/sia/catherine/dbinom>

### See Also

[dnbinom](#) for the negative binomial, and [dpois](#) for the Poisson distribution.

### Examples

```
# Compute P(45 < X < 55) for X Binomial(100,0.5)
sum(dbinom(46:54, 100, 0.5))

## Using "log = TRUE" for an extended range :
n <- 2000
k <- seq(0, n, by = 20)
plot(k, dbinom(k, n, pi/10, log=TRUE), type='l', ylab="log density",
      main = "dbinom(*, log=TRUE) is better than log(dbinom(*))")
lines(k, log(dbinom(k, n, pi/10)), col='red', lwd=2)
## extreme points are omitted since dbinom gives 0.
mtext("dbinom(k, log=TRUE)", adj=0)
mtext("extended range", adj=0, line = -1, font=4)
mtext("log(dbinom(k))", col="red", adj=1)
```

---

 biplot

*Biplot of Multivariate Data*


---

**Description**

Plot a biplot on the current graphics device.

**Usage**

```
biplot(x, ...)
```

```
## Default S3 method:
```

```
biplot(x, y, var.axes = TRUE, col, cex = rep(par("cex"), 2),
       xlabs = NULL, ylabs = NULL, expand = 1,
       xlim = NULL, ylim = NULL, arrow.len = 0.1,
       main = NULL, sub = NULL, xlab = NULL, ylab = NULL, ...)
```

**Arguments**

<code>x</code>	The <code>biplot</code> , a fitted object. For <code>biplot.default</code> , the first set of points (a two-column matrix), usually associated with observations.
<code>y</code>	The second set of points (a two-column matrix), usually associated with variables.
<code>var.axes</code>	If <code>TRUE</code> the second set of points have arrows representing them as (unscaled) axes.
<code>col</code>	A vector of length 2 giving the colours for the first and second set of points respectively (and the corresponding axes). If a single colour is specified it will be used for both sets. If missing the default colour is looked for in the <a href="#">palette</a> : if there it and the next colour as used, otherwise the first two colours of the palette are used.
<code>cex</code>	The character expansion factor used for labelling the points. The labels can be of different sizes for the two sets by supplying a vector of length two.
<code>xlabs</code>	A vector of character strings to label the first set of points: the default is to use the row dimname of <code>x</code> , or <code>1:n</code> is the dimname is <code>NULL</code> .
<code>ylabs</code>	A vector of character strings to label the second set of points: the default is to use the row dimname of <code>y</code> , or <code>1:n</code> is the dimname is <code>NULL</code> .
<code>expand</code>	An expansion factor to apply when plotting the second set of points relative to the first. This can be used to tweak the scaling of the two sets to a physically comparable scale.
<code>arrow.len</code>	The length of the arrow heads on the axes plotted in <code>var.axes</code> is true. The arrow head can be suppressed by <code>arrow.len = 0</code> .
<code>xlim</code> , <code>ylim</code>	Limits for the x and y axes in the units of the first set of variables.
<code>main</code> , <code>sub</code> , <code>xlab</code> , <code>ylab</code> , ...	graphical parameters.

**Details**

A biplot is plot which aims to represent both the observations and variables of a matrix of multivariate data on the same plot. There are many variations on biplots (see the references) and perhaps the most widely used one is implemented by `biplot.princomp`. The function `biplot.default` merely provides the underlying code to plot two sets of variables on the same figure.

Graphical parameters can also be given to `biplot`.

**Side Effects**

a plot is produced on the current graphics device.

**References**

K. R. Gabriel (1971). The biplot graphical display of matrices with application to principal component analysis. *Biometrika* **58**, 453–467.

J.C. Gower and D. J. Hand (1996). *Biplots*. Chapman & Hall.

**See Also**

`biplot.princomp`, also for examples.

---

`biplot.princomp`      *Biplot for Principal Components*

---

**Description**

Produces a biplot (in the strict sense) from the output of `princomp` or `prcomp`

**Usage**

```
## S3 method for class 'prcomp':
biplot(x, choices = 1:2, scale = 1, pc.biplot = FALSE, ...)
```

```
## S3 method for class 'princomp':
biplot(x, choices = 1:2, scale = 1, pc.biplot = FALSE, ...)
```

**Arguments**

<code>x</code>	an object of class "princomp".
<code>choices</code>	length 2 vector specifying the components to plot. Only the default is a biplot in the strict sense.
<code>scale</code>	The variables are scaled by $\lambda^{\text{scale}}$ and the observations are scaled by $\lambda^{(1-\text{scale})}$ where $\lambda$ are the singular values as computed by <code>princomp</code> . Normally $0 \leq \text{scale} \leq 1$ , and a warning will be issued if the specified <code>scale</code> is outside this range.

`pc.biplot` If true, use what Gabriel (1971) refers to as a "principal component biplot", with `lambda = 1` and observations scaled up by `sqrt(n)` and variables scaled down by `sqrt(n)`. Then inner products between variables approximate covariances and distances between observations approximate Mahalanobis distance.

... optional arguments to be passed to `biplot.default`.

### Details

This is a method for the generic function `biplot`. There is considerable confusion over the precise definitions: those of the original paper, Gabriel (1971), are followed here. Gabriel and Odoroff (1990) use the same definitions, but their plots actually correspond to `pc.biplot = TRUE`.

### Side Effects

a plot is produced on the current graphics device.

### References

Gabriel, K. R. (1971). The biplot graphical display of matrices with applications to principal component analysis. *Biometrika*, **58**, 453–467.

Gabriel, K. R. and Odoroff, C. L. (1990). Biplots in biomedical research. *Statistics in Medicine*, **9**, 469–485.

### See Also

`biplot`, `princomp`.

### Examples

```
biplot(princomp(USArrests))
```

---

birthday

*Probability of coincidences*

---

### Description

Computes approximate answers to a generalised “birthday paradox” problem. `pbirthday` computes the probability of a coincidence and `qbirthday` computes the number of observations needed to have a specified probability of coincidence.

### Usage

```
qbirthday(prob = 0.5, classes = 365, coincident = 2)
pbirthday(n, classes = 365, coincident = 2)
```

**Arguments**

<code>classes</code>	How many distinct categories the people could fall into
<code>prob</code>	The desired probability of coincidence
<code>n</code>	The number of people
<code>coincident</code>	The number of people to fall in the same category

**Details**

The birthday paradox is that a very small number of people, 23, suffices to have a 50-50 chance that two of them have the same birthday. This function generalises the calculation to probabilities other than 0.5, numbers of coincident events other than 2, and numbers of classes other than 365.

This formula is approximate, as the example below shows. For `coincident=2` the exact computation is straightforward and may be preferable.

**Value**

<code>qbirthday</code>	Number of people needed for a probability <code>prob</code> that <code>k</code> of them have the same one out of <code>classes</code> equiprobable labels.
<code>pbirthday</code>	Probability of the specified coincidence

**References**

Diaconis P, Mosteller F, "Methods for studying coincidences". JASA 84:853-861

**Examples**

```
## the standard version
qbirthday()
## same 4-digit PIN number
qbirthday(classes=10^4)
## 0.9 probability of three coincident birthdays
qbirthday(coincident=3,prob=0.9)
## Chance of 4 coincident birthdays in 150 people
pbirthday(150,coincident=4)
## Accuracy compared to exact calculation
x1<- sapply(10:100, pbirthday)
x2<-1-sapply(10:100, function(n)prod((365:(365-n+1))/rep(365,n)))
par(mfrow=c(2,2))
plot(x1,x2,xlab="approximate",ylab="exact")
abline(0,1)
plot(x1,x1-x2,xlab="approximate",ylab="error")
abline(h=0)
plot(x1,x2,log="xy",xlab="approximate",ylab="exact")
abline(0,1)
plot(1-x1,1-x2,log="xy",xlab="approximate",ylab="exact")
abline(0,1)
```

---

 Box.test

*Box-Pierce and Ljung-Box Tests*


---

**Description**

Compute the Box–Pierce or Ljung–Box test statistic for examining the null hypothesis of independence in a given time series.

**Usage**

```
Box.test(x, lag = 1, type = c("Box-Pierce", "Ljung-Box"))
```

**Arguments**

<code>x</code>	a numeric vector or univariate time series.
<code>lag</code>	the statistic will be based on <code>lag</code> autocorrelation coefficients.
<code>type</code>	test to be performed: partial matching is used.

**Value**

A list with class "htest" containing the following components:

<code>statistic</code>	the value of the test statistic.
<code>parameter</code>	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>method</code>	a character string indicating which type of test was performed.
<code>data.name</code>	a character string giving the name of the data.

**Note**

Missing values are not handled.

**Author(s)**

A. Trapletti

**References**

Box, G. E. P. and Pierce, D. A. (1970), Distribution of residual correlations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, **65**, 1509–1526.

Ljung, G. M. and Box, G. E. P. (1978), On a measure of lack of fit in time series models. *Biometrika* **65**, 553–564.

Harvey, A. C. (1993) *Time Series Models*. 2nd Edition, Harvester Wheatsheaf, NY, pp. 44, 45.

**Examples**

```
x <- rnorm (100)
Box.test (x, lag = 1)
Box.test (x, lag = 1, type="Ljung")
```

---

**C***Sets Contrasts for a Factor*

---

**Description**

Sets the "contrasts" attribute for the factor.

**Usage**

```
C(object, contr, how.many, ...)
```

**Arguments**

<code>object</code>	a factor or ordered factor
<code>contr</code>	which contrasts to use. Can be a matrix with one row for each level of the factor or a suitable function like <code>contr.poly</code> or a character string giving the name of the function
<code>how.many</code>	the number of contrasts to set, by default one less than <code>nlevels(object)</code> .
<code>...</code>	additional arguments for the function <code>contr</code> .

**Details**

For compatibility with S, `contr` can be `treatment`, `helmert`, `sum` or `poly` (without quotes) as shorthand for `contr.treatment` and so on.

**Value**

The factor object with the "contrasts" attribute set.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models*. Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[contrasts](#), [contr.sum](#), etc.

**Examples**

```
## reset contrasts to defaults
options(contrasts=c("contr.treatment", "contr.poly"))
attach(warpbreaks)
tens <- C(tension, poly, 1)
attributes(tens)
detach()
## tension SHOULD be an ordered factor, but as it is not we can use
aov(breaks ~ wool + tens + tension, data=warpbreaks)

## show the use of ... The default contrast is contr.treatment here
summary(lm(breaks ~ wool + C(tension, base=2), data=warpbreaks))

# following on from help(esoph)
model3 <- glm(cbind(ncases, ncontrols) ~ agegp + C(tobgp, , 1) +
             C(alcgp, , 1), data = esoph, family = binomial())
summary(model3)
```

---

cancor

*Canonical Correlations*


---

**Description**

Compute the canonical correlations between two data matrices.

**Usage**

```
cancor(x, y, xcenter = TRUE, ycenter = TRUE)
```

**Arguments**

<code>x</code>	numeric matrix ( $n \times p_1$ ), containing the x coordinates.
<code>y</code>	numeric matrix ( $n \times p_2$ ), containing the y coordinates.
<code>xcenter</code>	logical or numeric vector of length $p_1$ , describing any centering to be done on the x values before the analysis. If <code>TRUE</code> (default), subtract the column means. If <code>FALSE</code> , do not adjust the columns. Otherwise, a vector of values to be subtracted from the columns.
<code>ycenter</code>	analogous to <code>xcenter</code> , but for the y values.

**Details**

The canonical correlation analysis seeks linear combinations of the y variables which are well explained by linear combinations of the x variables. The relationship is symmetric as 'well explained' is measured by correlations.

**Value**

A list containing the following components:

cor	correlations.
xcoef	estimated coefficients for the x variables.
ycoef	estimated coefficients for the y variables.
xcenter	the values used to adjust the x variables.
ycenter	the values used to adjust the x variables.

**References**

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
- Hotelling H. (1936). Relations between two sets of variables. *Biometrika*, **28**, 321–327.
- Seber, G. A. F. (1984). *Multivariate Observations*. New York: Wiley, p. 506f.

**See Also**

[qr](#), [svd](#).

**Examples**

```
pop <- LifeCycleSavings[, 2:3]
oec <- LifeCycleSavings[, -(2:3)]
cancor(pop, oec)

x <- matrix(rnorm(150), 50, 3)
y <- matrix(rnorm(250), 50, 5)
(cxy <- cancor(x, y))
all(abs(cor(x %>% cxy$xcoef,
            y %>% cxy$ycoef)[,1:3] - diag(cxy $ cor)) < 1e-15)
all(abs(cor(x %>% cxy$xcoef) - diag(3)) < 1e-15)
all(abs(cor(y %>% cxy$ycoef) - diag(5)) < 1e-15)
```

---

case/variable.names

*Case and Variable Names of Fitted Models*

---

**Description**

Simple utilities returning (non-missing) case names, and (non-eliminated) variable names.

**Usage**

```

case.names(object, ...)
## S3 method for class 'lm':
case.names(object, full = FALSE, ...)

variable.names(object, ...)
## S3 method for class 'lm':
variable.names(object, full = FALSE, ...)

```

**Arguments**

`object` an R object, typically a fitted model.

`full` logical; if TRUE, all names (including zero weights, ...) are returned.

`...` further arguments passed to or from other methods.

**Value**

A character vector.

**See Also**

[lm](#)

**Examples**

```

x <- 1:20
y <- x + (x/4 - 2)^3 + rnorm(20, s=3)
names(y) <- paste("0",x,sep=".")
ww <- rep(1,20); ww[13] <- 0
summary(lmxy <- lm(y ~ x + I(x^2)+I(x^3) + I((x-10)^2),
                 weights = ww), cor = TRUE)

variable.names(lmxy)
variable.names(lmxy, full= TRUE)# includes the last
case.names(lmxy)
case.names(lmxy, full = TRUE)# includes the 0-weight case

```

**Description**

Density, distribution function, quantile function and random generation for the Cauchy distribution with location parameter `location` and scale parameter `scale`.

**Usage**

```
dcauchy(x, location = 0, scale = 1, log = FALSE)
pcauchy(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qcauchy(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rcauchy(n, location = 0, scale = 1)
```

**Arguments**

`x`, `q`            vector of quantiles.  
`p`                    vector of probabilities.  
`n`                    number of observations. If `length(n) > 1`, the length is taken to be the number required.  
`location`, `scale`        location and scale parameters.  
`log`, `log.p`        logical; if TRUE, probabilities `p` are given as `log(p)`.  
`lower.tail`        logical; if TRUE (default), probabilities are  $P[X \leq x]$ , otherwise,  $P[X > x]$ .

**Details**

If `location` or `scale` are not specified, they assume the default values of 0 and 1 respectively.

The Cauchy distribution with location  $l$  and scale  $s$  has density

$$f(x) = \frac{1}{\pi s} \left( 1 + \left( \frac{x-l}{s} \right)^2 \right)^{-1}$$

for all  $x$ .

**Value**

`dcauchy`, `pcauchy`, and `qcauchy` are respectively the density, distribution function and quantile function of the Cauchy distribution. `rcauchy` generates random deviates from the Cauchy.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[dt](#) for the t distribution which generalizes `dcauchy(*, l = 0, s = 1)`.

**Examples**

```
dcauchy(-1:4)
```

---

chisq.test

*Pearson's Chi-squared Test for Count Data*


---

### Description

chisq.test performs chi-squared tests on contingency tables.

### Usage

```
chisq.test(x, y = NULL, correct = TRUE,
           p = rep(1/length(x), length(x)),
           simulate.p.value = FALSE, B = 2000)
```

### Arguments

x	a vector or matrix.
y	a vector; ignored if x is a matrix.
correct	a logical indicating whether to apply continuity correction when computing the test statistic.
p	a vector of probabilities of the same length of x.
simulate.p.value	a logical indicating whether to compute p-values by Monte Carlo simulation.
B	an integer specifying the number of replicates used in the Monte Carlo simulation.

### Details

If `x` is a matrix with one row or column, or if `x` is a vector and `y` is not given, `x` is treated as a one-dimensional contingency table. In this case, the hypothesis tested is whether the population probabilities equal those in `p`, or are all equal if `p` is not given.

If `x` is a matrix with at least two rows and columns, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, `x` and `y` must be vectors or factors of the same length; incomplete cases are removed, the objects are coerced into factor objects, and the contingency table is computed from these. Then, Pearson's chi-squared test of the null that the joint distribution of the cell counts in a 2-dimensional contingency table is the product of the row and column marginals is performed. If `simulate.p.value` is `FALSE`, the p-value is computed from the asymptotic chi-squared distribution of the test statistic; continuity correction is only used in the 2-by-2 case if `correct` is `TRUE`. Otherwise, if `simulate.p.value` is `TRUE`, the p-value is computed by Monte Carlo simulation with `B` replicates. This is done by random sampling from the set of all contingency tables with given marginals, and works only if the marginals are positive. (A C translation of the algorithm of Patefield (1981) is used.)

**Value**

A list with class "htest" containing the following components:

statistic	the value the chi-squared test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic, NA if the p-value is computed by Monte Carlo simulation.
p.value	the p-value for the test.
method	a character string indicating the type of test performed, and whether Monte Carlo simulation or continuity correction was used.
data.name	a character string giving the name(s) of the data.
observed	the observed counts.
expected	the expected counts under the null hypothesis.
residuals	the Pearson residuals, $(\text{observed} - \text{expected}) / \sqrt{\text{expected}}$ .

**References**

Patefield, W. M. (1981) Algorithm AS159. An efficient method of generating  $r \times c$  tables with given row and column totals. *Applied Statistics* **30**, 91–97.

**Examples**

```
## Not really a good example
chisq.test(InsectSprays$count > 7, InsectSprays$spray)
# Prints test summary
chisq.test(InsectSprays$count > 7, InsectSprays$spray)$obs
# Counts observed
chisq.test(InsectSprays$count > 7, InsectSprays$spray)$exp
# Counts expected under the null

## Effect of simulating p-values
x <- matrix(c(12, 5, 7, 7), nc = 2)
chisq.test(x)$p.value      # 0.4233
chisq.test(x, simulate.p.value = TRUE, B = 10000)$p.value
# around 0.29!

## Testing for population probabilities
## Case A. Tabulated data
x <- c(A = 20, B = 15, C = 25)
chisq.test(x)
chisq.test(as.table(x))    # the same
## Case B. Raw data
x <- trunc(5 * runif(100))
chisq.test(table(x))       # NOT 'chisq.test(x)'
```

Chisquare

*The (non-central) Chi-Squared Distribution***Description**

Density, distribution function, quantile function and random generation for the chi-squared ( $\chi^2$ ) distribution with `df` degrees of freedom and optional non-centrality parameter `ncp`.

**Usage**

```
dchisq(x, df, ncp=0, log = FALSE)
pchisq(q, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
qchisq(p, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
rchisq(n, df, ncp=0)
```

**Arguments**

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>df</code>	degrees of freedom (non-negative, but can be non-integer).
<code>ncp</code>	non-centrality parameter (non-negative). Note that <code>ncp</code> values larger than about 1417 are not allowed currently for <code>pchisq</code> and <code>qchisq</code> .
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

The chi-squared distribution with `df` =  $n$  degrees of freedom has density

$$f_n(x) = \frac{1}{2^{n/2}\Gamma(n/2)} x^{n/2-1} e^{-x/2}$$

for  $x > 0$ . The mean and variance are  $n$  and  $2n$ .

The non-central chi-squared distribution with `df` =  $n$  degrees of freedom and non-centrality parameter `ncp` =  $\lambda$  has density

$$f(x) = e^{-\lambda/2} \sum_{r=0}^{\infty} \frac{(\lambda/2)^r}{r!} f_{n+2r}(x)$$

for  $x \geq 0$ . For integer  $n$ , this is the distribution of the sum of squares of  $n$  normals each with variance one,  $\lambda$  being the sum of squares of the normal means; further,  $E(X) = n + \lambda$ ,  $Var(X) = 2(n + 2 * \lambda)$ , and  $E((X - E(X))^3) = 8(n + 3 * \lambda)$ .

Note that the degrees of freedom `df` =  $n$ , can be non-integer, and for non-centrality  $\lambda > 0$ , even  $n = 0$ ; see the reference, chapter 29.

**Value**

`dchisq` gives the density, `pchisq` gives the distribution function, `qchisq` gives the quantile function, and `rchisq` generates random deviates.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Johnson, Kotz and Balakrishnan (1995). *Continuous Univariate Distributions*, Vol 2; Wiley NY;

**See Also**

A central chi-squared distribution with  $n$  degrees of freedom is the same as a Gamma distribution with shape  $\alpha = n/2$  and scale  $\sigma = 2$ . Hence, see [dgamma](#) for the Gamma distribution.

**Examples**

```
dchisq(1, df=1:3)
pchisq(1, df= 3)
pchisq(1, df= 3, ncp = 0:4)# includes the above

x <- 1:10
## Chi-squared(df = 2) is a special exponential distribution
all.equal(dchisq(x, df=2), dexp(x, 1/2))
all.equal(pchisq(x, df=2), pexp(x, 1/2))

## non-central RNG -- df=0 is ok for ncp > 0: Z0 has point mass at 0!
Z0 <- rchisq(100, df = 0, ncp = 2.)
graphics::stem(Z0)

## Not run:
## visual testing
## do P-P plots for 1000 points at various degrees of freedom
L <- 1.2; n <- 1000; pp <- ppoints(n)
op <- par(mfrow = c(3,3), mar= c(3,3,1,1)+.1, mgp= c(1.5,.6,0),
         oma = c(0,0,3,0))
for(df in 2^(4*rnorm(9))) {
  plot(pp, sort(pchisq(rr <- rchisq(n,df=df, ncp=L), df=df, ncp=L)),
       ylab="pchisq(rchisq(.),.)", pch=".")
  mtext(paste("df = ",formatC(df, digits = 4)), line= -2, adj=0.05)
  abline(0,1,col=2)
}
mtext(expression("P-P plots : Noncentral  " *
                chi^2 *(n=1000, df=X, ncp= 1.2)"),
       cex = 1.5, font = 2, outer=TRUE)
par(op)
## End(Not run)
```

---

clearNames	<i>Remove the Names from an Object</i>
------------	--

---

**Description**

This function sets the `names` attribute of `object` to `NULL` and returns the object.

**Usage**

```
clearNames(object)
```

**Arguments**

`object` an object that may have a `names` attribute

**Value**

An object similar to `object` but without names.

**Author(s)**

Douglas Bates and Saikat DebRoy

**See Also**

[setNames](#)

**Examples**

```
lapply( women, mean ) # has a names attribute
clearNames( lapply( women, mean ) ) # removes the names
```

---

cmdscale	<i>Classical (Metric) Multidimensional Scaling</i>
----------	--

---

**Description**

Classical multidimensional scaling of a data matrix. Also known as *principal coordinates analysis* (Gower, 1966).

**Usage**

```
cmdscale(d, k = 2, eig = FALSE, add = FALSE, x.ret = FALSE)
```

**Arguments**

<code>d</code>	a distance structure such as that returned by <code>dist</code> or a full symmetric matrix containing the dissimilarities.
<code>k</code>	the dimension of the space which the data are to be represented in; must be in $\{1, 2, \dots, n - 1\}$ .
<code>eig</code>	indicates whether eigenvalues should be returned.
<code>add</code>	logical indicating if an additive constant $c^*$ should be computed, and added to the non-diagonal dissimilarities such that all $n - 1$ eigenvalues are non-negative.
<code>x.ret</code>	indicates whether the doubly centered symmetric distance matrix should be returned.

**Details**

Multidimensional scaling takes a set of dissimilarities and returns a set of points such that the distances between the points are approximately equal to the dissimilarities.

The functions `isoMDS` and `sammon` in package **MASS** provide alternative ordination techniques.

When `add = TRUE`, an additive constant  $c^*$  is computed, and the dissimilarities  $d_{ij} + c^*$  are used instead of the original  $d_{ij}$ 's.

Whereas S (Becker *et al.*, 1988) computes this constant using an approximation suggested by Torgerson, R uses the analytical solution of Cailliez (1983), see also Cox and Cox (1994).

**Value**

If `eig = FALSE` and `x.ret = FALSE` (default), a matrix with `k` columns whose rows give the coordinates of the points chosen to represent the dissimilarities.

Otherwise, a list containing the following components.

<code>points</code>	a matrix with <code>k</code> columns whose rows give the coordinates of the points chosen to represent the dissimilarities.
<code>eig</code>	the $n - 1$ eigenvalues computed during the scaling process if <code>eig</code> is true.
<code>x</code>	the doubly centered distance matrix if <code>x.ret</code> is true.
<code>GOF</code>	a numeric vector of length 2, equal to say $(g_1, g_2)$ , where $g_i = (\sum_{j=1}^k \lambda_j) / (\sum_{j=1}^n T_i(\lambda_j))$ , where $\lambda_j$ are the eigenvalues (sorted decreasingly), $T_1(v) =  v $ , and $T_2(v) = \max(v, 0)$ .

**References**

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
- Cailliez, F. (1983) The analytical solution of the additive constant problem. *Psychometrika* **48**, 343–349.
- Cox, T. F. and Cox, M. A. A. (1994) *Multidimensional Scaling*. Chapman and Hall.
- Gower, J. C. (1966) Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika* **53**, 325–328.

Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979). Chapter 14 of *Multivariate Analysis*, London: Academic Press.

Seber, G. A. F. (1984). *Multivariate Observations*. New York: Wiley.

Torgerson, W. S. (1958). *Theory and Methods of Scaling*. New York: Wiley.

### See Also

[dist](#). Also [isoMDS](#) and [sammon](#) in package **MASS**.

### Examples

```
loc <- cmdscale(eurodist)
x <- loc[,1]
y <- -loc[,2]
plot(x, y, type="n", xlab="", ylab="", main="cmdscale(eurodist)")
text(x, y, names(eurodist), cex=0.8)

cmdsE <- cmdscale(eurodist, k=20, add = TRUE, eig = TRUE, x.ret = TRUE)
str(cmdsE)
```

---

coef

*Extract Model Coefficients*

---

### Description

`coef` is a generic function which extracts model coefficients from objects returned by modeling functions. `coefficients` is an *alias* for it.

### Usage

```
coef(object, ...)
coefficients(object, ...)
```

### Arguments

<code>object</code>	an object for which the extraction of model coefficients is meaningful.
<code>...</code>	other arguments.

### Details

All object classes which are returned by model fitting functions should provide a `coef` method or use the default one. (Note that the method is for `coef` and not `coefficients`.)

Class "aov" has a `coef` method that does not report aliased coefficients (see [alias](#)).

### Value

Coefficients extracted from the model object `object`.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

[fitted.values](#) and [residuals](#) for related methods; [glm](#), [lm](#) for model fitting.

**Examples**

```
x <- 1:5; coef(lm(c(1:3,7,6) ~ x))
```

---

complete.cases      *Find Complete Cases*

---

**Description**

Return a logical vector indicating which cases are complete, i.e., have no missing values.

**Usage**

```
complete.cases(...)
```

**Arguments**

...                    a sequence of vectors, matrices and data frames.

**Value**

A logical vector specifying which observations/rows have no missing values across the entire sequence.

**See Also**

[is.na](#), [na.omit](#), [na.fail](#).

**Examples**

```
x <- airquality[, -1] # x is a regression design matrix
y <- airquality[, 1] # y is the corresponding response

stopifnot(complete.cases(y) != is.na(y))
ok <- complete.cases(x,y)
sum(!ok) # how many are not "ok" ?
x <- x[ok,]
y <- y[ok]
```

---

`confint`*Confidence Intervals for Model Parameters*

---

**Description**

Computes confidence intervals for one or more parameters in a fitted model. Base has a method for objects inheriting from class "lm".

**Usage**

```
confint(object, parm, level = 0.95, ...)
```

**Arguments**

<code>object</code>	a fitted model object.
<code>parm</code>	a specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
<code>level</code>	the confidence level required.
<code>...</code>	additional argument(s) for methods

**Details**

`confint` is a generic function with no default method. For objects of class "lm" the direct formulae based on t values are used.

Package **MASS** contains methods for "glm" and "nls" objects.

**Value**

A matrix (or vector) with columns giving lower and upper confidence limits for each parameter. These will be labelled as  $(1-\text{level})/2$  and  $1 - (1-\text{level})/2$  in % (by default 2.5% and 97.5%).

**See Also**

[confint.nls](#) in package **MASS**.

**Examples**

```
fit <- lm(100/mpg ~ disp + hp + wt + am, data=mtcars)
confint(fit)
confint(fit, "wt")
```

---

constrOptim                      *Linearly constrained optimisation*

---

### Description

Minimise a function subject to linear inequality constraints using an adaptive barrier algorithm.

### Usage

```
constrOptim(theta, f, grad, ui, ci, mu = 1e-04, control = list(),
            method = if(is.null(grad)) "Nelder-Mead" else "BFGS",
            outer.iterations = 100, outer.eps = 1e-05, ...)
```

### Arguments

theta	Starting value: must be in the feasible region.
f	Function to minimise.
grad	Gradient of f.
ui	Constraints (see below).
ci	Constraints (see below).
mu	(Small) tuning parameter.
control	Passed to <code>optim</code> .
method	Passed to <code>optim</code> .
outer.iterations	Iterations of the barrier algorithm.
outer.eps	Criterion for relative convergence of the barrier algorithm.
...	Other arguments passed to <code>optim</code> , which will pass them to f and grad if it does not used them.

### Details

The feasible region is defined by `ui %*% theta - ci >= 0`. The starting value must be in the interior of the feasible region, but the minimum may be on the boundary.

A logarithmic barrier is added to enforce the constraints and then `optim` is called. The barrier function is chosen so that the objective function should decrease at each outer iteration. Minima in the interior of the feasible region are typically found quite quickly, but a substantial number of outer iterations may be needed for a minimum on the boundary.

The tuning parameter `mu` multiplies the barrier term. Its precise value is often relatively unimportant. As `mu` increases the augmented objective function becomes closer to the original objective function but also less smooth near the boundary of the feasible region.

Any `optim` method that permits infinite values for the objective function may be used (currently all but "L-BFGS-B"). The gradient function must be supplied except with `method="Nelder-Mead"`.

As with `optim`, the default is to minimise and maximisation can be performed by setting `control$fnscale` to a negative value.

**Value**

As for `optim`, but with two extra components: `barrier.value` giving the value of the barrier function at the optimum and `outer.iterations` gives the number of outer iterations (calls to `optim`)

**References**

K. Lange *Numerical Analysis for Statisticians*. Springer 2001, p185ff

**See Also**

`optim`, especially `method="L-BGFS-B"` which does box-constrained optimisation.

**Examples**

```
## from optim
fr <- function(x) { ## Rosenbrock Banana function
  x1 <- x[1]
  x2 <- x[2]
  100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
  x1 <- x[1]
  x2 <- x[2]
  c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
    200 * (x2 - x1 * x1))
}

optim(c(-1.2,1), fr, grr)
#Box-constraint, optimum on the boundary
constrOptim(c(-1.2,0.9), fr, grr, ui=rbind(c(-1,0),c(0,-1)), ci=c(-1,-1))
# x<=0.9, y-x>0.1
constrOptim(c(.5,0), fr, grr, ui=rbind(c(-1,0),c(1,-1)), ci=c(-0.9,0.1))

## Solves linear and quadratic programming problems
## but needs a feasible starting value
#
# from example(solve.QP) in 'quadprog'
# no derivative
fQP <- function(b) {-sum(c(0,5,0)*b)+0.5*sum(b*b)}
Amat <- matrix(c(-4,-3,0,2,1,0,0,-2,1),3,3)
bvec <- c(-8,2,0)
constrOptim(c(2,-1,-1), fQP, NULL, ui=t(Amat),ci=bvec)
# derivative
gQP <- function(b) {-c(0,5,0)+b}
constrOptim(c(2,-1,-1), fQP, gQP, ui=t(Amat), ci=bvec)

## Now with maximisation instead of minimisation
hQP <- function(b) {sum(c(0,5,0)*b)-0.5*sum(b*b)}
constrOptim(c(2,-1,-1), hQP, NULL, ui=t(Amat), ci=bvec,
  control=list(fnscale=-1))
```

---

 contrast

*Contrast Matrices*


---

### Description

Return a matrix of contrasts.

### Usage

```

contr.helmert(n, contrasts = TRUE)
contr.poly(n, scores = 1:n, contrasts = TRUE)
contr.sum(n, contrasts = TRUE)
contr.treatment(n, base = 1, contrasts = TRUE)

```

### Arguments

<code>n</code>	a vector of levels for a factor, or the number of levels.
<code>contrasts</code>	a logical indicating whether contrasts should be computed.
<code>scores</code>	the set of values over which orthogonal polynomials are to be computed.
<code>base</code>	an integer specifying which group is considered the baseline group. Ignored if <code>contrasts</code> is <code>FALSE</code> .

### Details

These functions are used for creating contrast matrices for use in fitting analysis of variance and regression models. The columns of the resulting matrices contain contrasts which can be used for coding a factor with  $n$  levels. The returned value contains the computed contrasts. If the argument `contrasts` is `FALSE` then a square indicator matrix is returned.

`contr.helmert` returns Helmert contrasts, which contrast the second level with the first, the third with the average of the first two, and so on. `contr.poly` returns contrasts based on orthogonal polynomials. `contr.sum` uses “sum to zero contrasts”.

`contr.treatment` contrasts each level with the baseline level (specified by `base`): the baseline level is omitted. Note that this does not produce “contrasts” as defined in the standard theory for linear models as they are not orthogonal to the constant.

### Value

A matrix with  $n$  rows and  $k$  columns, with  $k=n-1$  if `contrasts` is `TRUE` and  $k=n$  if `contrasts` is `FALSE`.

### References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models*. Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[contrasts](#), [C](#), and [aov](#), [glm](#), [lm](#).

**Examples**

```
(cH <- contr.helmert(4))
apply(cH, 2,sum) # column sums are 0!
crossprod(cH) # diagonal -- columns are orthogonal
contr.helmert(4, contrasts = FALSE) # just the 4 x 4 identity matrix

(cT <- contr.treatment(5))
all(crossprod(cT) == diag(4)) # TRUE: even orthonormal

(cP <- contr.poly(3)) # Linear and Quadratic
zapsmall(crossprod(cP), dig=15) # orthonormal up to fuzz
```

---

contrasts

*Get and Set Contrast Matrices*

---

**Description**

Set and view the contrasts associated with a factor.

**Usage**

```
contrasts(x, contrasts = TRUE)
contrasts(x, how.many) <- value
```

**Arguments**

<code>x</code>	a factor or a logical variable.
<code>contrasts</code>	logical. See Details.
<code>how.many</code>	How many contrasts should be made. Defaults to one less than the number of levels of <code>x</code> . This need not be the same as the number of columns of <code>ctr</code> .
<code>value</code>	either a matrix whose columns give coefficients for contrasts in the levels of <code>x</code> , or the (quoted) name of a function which computes such matrices.

**Details**

If contrasts are not set for a factor the default functions from `options("contrasts")` are used.

A logical vector `x` is converted into a two-level factor with levels `c(FALSE, TRUE)` (regardless of which levels occur in the variable).

The argument `contrasts` is ignored if `x` has a matrix `contrasts` attribute set. Otherwise if `contrasts = TRUE` it is passed to a contrasts function such as `contr.treatment` and if `contrasts = FALSE` an identity matrix is returned.

## References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models*. Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

`C`, `contr.helmert`, `contr.poly`, `contr.sum`, `contr.treatment`; `glm`, `aov`, `lm`.

## Examples

```
example(factor)
fff <- ff[, drop=TRUE] # reduce to 5 levels.
contrasts(fff) # treatment contrasts by default
contrasts(C(fff, sum))
contrasts(fff, contrasts = FALSE) # the 5x5 identity matrix

contrasts(fff) <- contr.sum(5); contrasts(fff) # set sum contrasts
contrasts(fff, 2) <- contr.sum(5); contrasts(fff) # set 2 contrasts
# supply 2 contrasts, compute 2 more to make full set of 4.
contrasts(fff) <- contr.sum(5)[,1:2]; contrasts(fff)
```

---

convolve

*Fast Convolution*

---

## Description

Use the Fast Fourier Transform to compute the several kinds of convolutions of two sequences.

## Usage

```
convolve(x, y, conj = TRUE, type = c("circular", "open", "filter"))
```

## Arguments

<code>x, y</code>	numeric sequences <i>of the same length</i> to be convolved.
<code>conj</code>	logical; if TRUE, take the complex <i>conjugate</i> before back-transforming (default, and used for usual convolution).
<code>type</code>	character; one of "circular", "open", "filter" (beginning of word is ok). For <code>circular</code> , the two sequences are treated as <i>circular</i> , i.e., periodic. For <code>open</code> and <code>filter</code> , the sequences are padded with 0s (from left and right) first; "filter" returns the middle sub-vector of "open", namely, the result of running a weighted mean of <code>x</code> with weights <code>y</code> .

## Details

The Fast Fourier Transform, `fft`, is used for efficiency.

The input sequences `x` and `y` must have the same length if `circular` is true.

Note that the usual definition of convolution of two sequences `x` and `y` is given by `convolve(x, rev(y), type = "o")`.

**Value**

If `r <- convolve(x,y, type = "open")` and `n <- length(x)`, `m <- length(y)`, then

$$r_k = \sum_i x_{k-m+i} y_i$$

where the sum is over all valid indices  $i$ , for  $k = 1, \dots, n + m - 1$

If `type == "circular"`,  $n = m$  is required, and the above is true for  $i, k = 1, \dots, n$  when  $x_j := x_{n+j}$  for  $j < 1$ .

**References**

Brillinger, D. R. (1981) *Time Series: Data Analysis and Theory*, Second Edition. San Francisco: Holden-Day.

**See Also**

[fft](#), [nextn](#), and particularly [filter](#) (from the `stats` package) which may be more appropriate.

**Examples**

```
x <- c(0,0,0,100,0,0,0)
y <- c(0,0,1, 2 ,1,0,0)/4
zapsmall(convolve(x,y))          # *NOT* what you first thought.
zapsmall(convolve(x, y[3:5], type="f")) # rather
x <- rnorm(50)
y <- rnorm(50)
# Circular convolution *has* this symmetry:
all.equal(convolve(x,y, conj = FALSE), rev(convolve(rev(y),x)))

n <- length(x <- -20:24)
y <- (x-10)^2/1000 + rnorm(x)/8

Han <- function(y) # Hanning
  convolve(y, c(1,2,1)/4, type = "filter")

plot(x,y, main="Using convolve(.) for Hanning filters")
lines(x[-c(1, n) ], Han(y), col="red")
lines(x[-c(1:2, (n-1):n)], Han(Han(y)), lwd=2, col="dark blue")
```

---

cophenetic

*Cophenetic Distances for a Hierarchical Clustering*

---

**Description**

Computes the cophenetic distances for a hierarchical clustering.

**Usage**

```
cophenetic(x)
```

**Arguments**

`x` an object of class `hclust` or with a method for `as.hclust()` such as `agnes`.

**Details**

The cophenetic distance between two observations that have been clustered is defined to be the intergroup dissimilarity at which the two observations are first combined into a single cluster. Note that this distance has many ties and restrictions.

It can be argued that a dendrogram is an appropriate summary of some data if the correlation between the original distances and the cophenetic distances is high. Otherwise, it should simply be viewed as the description of the output of the clustering algorithm.

**Value**

An object of class `dist`.

**Author(s)**

Robert Gentleman

**References**

Sneath, P.H.A. and Sokal, R.R. (1973) *Numerical Taxonomy: The Principles and Practice of Numerical Classification*, p. 278 ff; Freeman, San Francisco.

**See Also**

`dist`, `hclust`

**Examples**

```
d1 <- dist(USArrests)
hc <- hclust(d1, "ave")
d2 <- cophenetic(hc)
cor(d1,d2) # 0.7659

## Example from Sneath & Sokal, Fig. 5-29, p.279
d0 <- c(1,3.8,4.4,5.1, 4,4.2,5, 2.6,5.3, 5.4)
attributes(d0) <- list(Size = 5, diag=TRUE)
class(d0) <- "dist"
names(d0) <- letters[1:5]
d0
str(upgma <- hclust(d0, method = "average"))
plot(upgma, hang = -1)
#
(d.coph <- cophenetic(upgma))
cor(d0, d.coph) # 0.9911
```

---

cor

*Correlation, Variance and Covariance (Matrices)*


---

### Description

`var`, `cov` and `cor` compute the variance of `x` and the covariance or correlation of `x` and `y` if these are vectors. If `x` and `y` are matrices then the covariances (or correlations) between the columns of `x` and the columns of `y` are computed.

`cov2cor` scales a covariance matrix into the corresponding correlation matrix *efficiently*.

### Usage

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "all.obs",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "all.obs",
    method = c("pearson", "kendall", "spearman"))

cov2cor(V)
```

### Arguments

<code>x</code>	a numeric vector, matrix or data frame.
<code>y</code>	NULL (default) or a vector, matrix or data frame with compatible dimensions to <code>x</code> . The default is equivalent to <code>y = x</code> (but more efficient).
<code>na.rm</code>	logical. Should missing values be removed?
<code>use</code>	an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings "all.obs", "complete.obs" or "pairwise.complete.obs".
<code>method</code>	a character string indicating which correlation coefficient (or covariance) is to be computed. One of "pearson" (default), "kendall", or "spearman", can be abbreviated.
<code>V</code>	symmetric numeric matrix, usually positive definite such as a covariance matrix.

### Details

For `cov` and `cor` one must *either* give a matrix or data frame for `x` *or* give both `x` and `y`.

`var` is just another interface to `cov`, where `na.rm` is used to determine the default for `use` when that is unspecified. If `na.rm` is TRUE then the complete observations (rows) are used (`use = "complete"`) to compute the variance. Otherwise (`use = "all"`), `var` will give an error if there are missing values.

If `use` is "all.obs", then the presence of missing observations will produce an error. If `use` is "complete.obs" then missing values are handled by casewise deletion. Finally, if `use` has

the value "pairwise.complete.obs" then the correlation between each pair of variables is computed using all complete pairs of observations on those variables. This can result in covariance or correlation matrices which are not positive semidefinite.

The denominator  $n - 1$  is used which gives an unbiased estimator of the (co)variance for i.i.d. observations. These functions return `NA` when there is only one observation (whereas S-PLUS has been returning `NaN`), and fail if `x` has length zero.

For `cor()`, if method is "kendall" or "spearman", Kendall's  $\tau$  or Spearman's  $\rho$  statistic is used to estimate a rank-based measure of association. These are more robust and have been recommended if the data do not necessarily come from a bivariate normal distribution.

For `cov()`, a non-Pearson method is unusual but available for the sake of completeness. Note that "spearman" basically computes `cor(R(x), R(y))` (or `cov(.,.)`) where `R(u) := rank(u, na.last="keep")`. Notice also that the ranking is (currently) done removing only cases that are missing on the variable itself, which may not be what you expect if you let use be "complete.obs" or "pairwise.complete.obs".

Scaling a covariance matrix into a correlation one can be achieved in many ways, mathematically most appealing by multiplication with a diagonal matrix from left and right, or more efficiently by using `sweep(., FUN = "/" )` twice. The `cov2cor` function is even a bit more efficient, and provided mostly for didactical reasons.

### Value

For `r <- cor(*, use = "all.obs")`, it is now guaranteed that `all(r <= 1)`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`cor.test` for confidence intervals (and tests).

`cov.wt` for *weighted* covariance computation.

`sd` for standard deviation (vectors).

### Examples

```
var(1:10)# 9.166667

var(1:5,1:5)# 2.5

## Two simple vectors
cor(1:10,2:11)# == 1

## Correlation Matrix of Multivariate sample:
(Cl <- cor(longley))
## Graphical Correlation Matrix:
symnum(Cl) # highly correlated

## Spearman's rho and Kendall's tau
```

```

symnum(c1S <- cor(longley, method = "spearman"))
symnum(c1K <- cor(longley, method = "kendall"))
## How much do they differ?
i <- lower.tri(C1)
cor(cbind(P = C1[i], S = c1S[i], K = c1K[i]))

## cov2cor() scales a covariance matrix by its diagonal
##           to become the correlation matrix.
cov2cor # see the function definition {and learn ..}
stopifnot(all.equal(C1, cov2cor(cov(longley))),
           all.equal(cor(longley, method="kendall"),
                     cov2cor(cov(longley, method="kendall"))))

##--- Missing value treatment:
C1 <- cov(swiss)
range(eigen(C1, only=TRUE)$val) # 6.19 1921
swM <- swiss
swM[1,2] <- swM[7,3] <- swM[25,5] <- NA # create 3 "missing"
try(cov(swM)) # Error: missing obs...
C2 <- cov(swM, use = "complete")
range(eigen(C2, only=TRUE)$val) # 6.46 1930
C3 <- cov(swM, use = "pairwise")
range(eigen(C3, only=TRUE)$val) # 6.19 1938

(scM <- symnum(cor(swM, method = "kendall", use = "complete")))
## Kendall's tau doesn't change much: identical symnum codings!
identical(scM, symnum(cor(swiss, method = "kendall")))

all.equal(cov2cor(cov(swM, method = "kendall", use = "pairwise")),
          cor(swM, method = "kendall", use = "pairwise"))

```

---

cor.test

*Test for Association/Correlation Between Paired Samples*


---

## Description

Test for association between paired samples, using one of Pearson's product moment correlation coefficient, Kendall's  $\tau$  or Spearman's  $\rho$ .

## Usage

```

cor.test(x, ...)

## Default S3 method:
cor.test(x, y,
         alternative = c("two.sided", "less", "greater"),
         method = c("pearson", "kendall", "spearman"),
         exact = NULL, conf.level = 0.95, ...)

## S3 method for class 'formula':
cor.test(formula, data, subset, na.action, ...)

```

**Arguments**

<code>x, y</code>	numeric vectors of data values. <code>x</code> and <code>y</code> must have the same length.
<code>alternative</code>	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter. "greater" corresponds to positive association, "less" to negative association.
<code>method</code>	a character string indicating which correlation coefficient is to be used for the test. One of "pearson", "kendall", or "spearman", can be abbreviated.
<code>exact</code>	a logical indicating whether an exact p-value should be computed. Only used for Kendall's $\tau$ . See the Details for the meaning of NULL (the default).
<code>conf.level</code>	confidence level for the returned confidence interval. Currently only used for the Pearson product moment correlation coefficient if there are at least 4 complete pairs of observations.
<code>formula</code>	a formula of the form $\sim u + v$ , where each of <code>u</code> and <code>v</code> are numeric variables giving the data values for one sample. The samples must be of the same length.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

**Details**

The three methods each estimate the association between paired samples and compute a test of the value being zero. They use different measures of association, all in the range  $[-1, 1]$  with 0 indicating no association. These are sometimes referred to as tests of no *correlation*, but that term is often confined to the default method.

If `method` is "pearson", the test statistic is based on Pearson's product moment correlation coefficient `cor(x, y)` and follows a t distribution with `length(x) - 2` degrees of freedom if the samples follow independent normal distributions. If there are at least 4 complete pairs of observation, an asymptotic confidence interval is given based on Fisher's Z transform.

If `method` is "kendall" or "spearman", Kendall's  $\tau$  or Spearman's  $\rho$  statistic is used to estimate a rank-based measure of association. These tests may be used if the data do not necessarily come from a bivariate normal distribution.

For Kendall's test, by default (if `exact` is NULL), an exact p-value is computed if there are less than 50 paired samples containing finite values and there are no ties. Otherwise, the test statistic is the estimate scaled to zero mean and unit variance, and is approximately normally distributed.

For Spearman's test, p-values are computed using algorithm AS 89.

**Value**

A list with class "htest" containing the following components:

<code>statistic</code>	the value of the test statistic.
<code>parameter</code>	the degrees of freedom of the test statistic in the case that it follows a t distribution.

p.value	the p-value of the test.
estimate	the estimated measure of association, with name "cor", "tau", or "rho" corresponding to the method employed.
null.value	the value of the association measure under the null hypothesis, always 0.
alternative	a character string describing the alternative hypothesis.
method	a character string indicating how the association was measured.
data.name	a character string giving the names of the data.
conf.int	a confidence interval for the measure of association. Currently only given for Pearson's product moment correlation coefficient in case of at least 4 complete pairs of observations.

## References

D. J. Best & D. E. Roberts (1975), Algorithm AS 89: The Upper Tail Probabilities of Spearman's  $\rho$ . *Applied Statistics*, **24**, 377–379.

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 185–194 (Kendall and Spearman tests).

## Examples

```
## Hollander & Wolfe (1973), p. 187f.
## Assessment of tuna quality. We compare the Hunter L measure of
## lightness to the averages of consumer panel scores (recoded as
## integer values from 1 to 6 and averaged over 80 such values) in
## 9 lots of canned tuna.

x <- c(44.4, 45.9, 41.9, 53.3, 44.7, 44.1, 50.7, 45.2, 60.1)
y <- c( 2.6,  3.1,  2.5,  5.0,  3.6,  4.0,  5.2,  2.8,  3.8)

## The alternative hypothesis of interest is that the
## Hunter L value is positively associated with the panel score.

cor.test(x, y, method = "kendall", alternative = "greater")
## => p=0.05972

cor.test(x, y, method = "kendall", alternative = "greater",
         exact = FALSE) # using large sample approximation
## => p=0.04765

## Compare this to
cor.test(x, y, method = "spearm", alternative = "g")
cor.test(x, y,                alternative = "g")

## Formula interface.
pairs(USJudgeRatings)
cor.test(~ CONT + INTG, data = USJudgeRatings)
```

cov.wt

*Weighted Covariance Matrices***Description**

Returns a list containing estimates of the weighted covariance matrix and the mean of the data, and optionally of the (weighted) correlation matrix.

**Usage**

```
cov.wt(x, wt = rep(1/nrow(x), nrow(x)), cor = FALSE, center = TRUE)
```

**Arguments**

<code>x</code>	a matrix or data frame. As usual, rows are observations and columns are variables.
<code>wt</code>	a non-negative and non-zero vector of weights for each observation. Its length must equal the number of rows of <code>x</code> .
<code>cor</code>	A logical indicating whether the estimated correlation weighted matrix will be returned as well.
<code>center</code>	Either a logical or a numeric vector specifying the centers to be used when computing covariances. If <code>TRUE</code> , the (weighted) mean of each variable is used, if <code>FALSE</code> , zero is used. If <code>center</code> is numeric, its length must equal the number of columns of <code>x</code> .

**Details**

The covariance matrix is divided by one minus the sum of squares of the weights, so if the weights are the default ( $1/n$ ) the conventional unbiased estimate of the covariance matrix with divisor  $(n-1)$  is obtained. This differs from the behaviour in S-PLUS.

**Value**

A list containing the following named components:

<code>cov</code>	the estimated (weighted) covariance matrix
<code>center</code>	an estimate for the center (mean) of the data.
<code>n.obs</code>	the number of observations (rows) in <code>x</code> .
<code>wt</code>	the weights used in the estimation. Only returned if given as an argument.
<code>cor</code>	the estimated correlation matrix. Only returned if <code>cor</code> is <code>TRUE</code> .

**See Also**

[cov](#) and [var](#).

---

`cpgram`*Plot Cumulative Periodogram*

---

**Description**

Plots a cumulative periodogram.

**Usage**

```
cpgram(ts, taper = 0.1,  
       main = paste("Series: ", deparse(substitute(ts))),  
       ci.col = "blue")
```

**Arguments**

<code>ts</code>	a univariate time series
<code>taper</code>	proportion tapered in forming the periodogram
<code>main</code>	main title
<code>ci.col</code>	colour for confidence band.

**Value**

None.

**Side Effects**

Plots the cumulative periodogram in a square plot.

**Note**

From package **MASS**.

**Author(s)**

B.D. Ripley

**Examples**

```
par(pty = "s", mfrow = c(1,2))  
cpgram(lh)  
lh.ar <- ar(lh, order.max = 9)  
cpgram(lh.ar$resid, main = "AR(3) fit to lh")  
  
cpgram(ldeaths)
```

---

`cutree`*Cut a tree into groups of data*

---

**Description**

Cuts a tree, e.g., as resulting from `hclust`, into several groups either by specifying the desired number(s) of groups or the cut height(s).

**Usage**

```
cutree(tree, k = NULL, h = NULL)
```

**Arguments**

`tree` a tree as produced by `hclust`. `cutree()` only expects a list with components `merge`, `height`, and `labels`, of appropriate content each.

`k` an integer scalar or vector with the desired number of groups

`h` numeric scalar or vector with heights where the tree should be cut.

At least one of `k` or `h` must be specified, `k` overrides `h` if both are given.

**Value**

`cutree` returns a vector with group memberships if `k` or `h` are scalar, otherwise a matrix with group memberships is returned where each column corresponds to the elements of `k` or `h`, respectively (which are also used as column names).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[hclust](#), [dendrogram](#) for cutting trees themselves.

**Examples**

```
hc <- hclust(dist(USArrests))

cutree(hc, k=1:5)#k = 1 is trivial
cutree(hc, h=250)

## Compare the 2 and 3 grouping:
g24 <- cutree(hc, k = c(2,4))
table(g24[, "2"], g24[, "4"])
```

decompose

*Classical Seasonal Decomposition by Moving Averages***Description**

Decompose a time series into seasonal, trend and irregular components using moving averages. Deals with additive or multiplicative seasonal component.

**Usage**

```
decompose(x, type = c("additive", "multiplicative"), filter = NULL)
```

**Arguments**

<code>x</code>	A time series.
<code>type</code>	The type of seasonal component.
<code>filter</code>	A vector of filter coefficients in reverse time order (as for AR or MA coefficients), used for filtering out the seasonal component. If <code>NULL</code> , a moving average with symmetric window is performed.

**Details**

The additive model used is:

$$Y[t] = T[t] + S[t] + e[t]$$

The multiplicative model used is:

$$Y[t] = T[t] * S[t] + e[t]$$

**Value**

An object of class "decomposed.ts" with following components:

<code>seasonal</code>	The seasonal component (i.e., the repeated seasonal figure)
<code>figure</code>	The estimated seasonal figure only
<code>trend</code>	The trend component
<code>random</code>	The remainder part
<code>type</code>	The value of <code>type</code>

**Note**

The function `stl` provides a much more sophisticated decomposition.

**Author(s)**

David Meyer (David.Meyer@wu-wien.ac.at)

**See Also**[stl](#)**Examples**

```
m <- decompose(co2)
m$figure
plot(m)
```

---

delete.response      *Modify Terms Objects*

---

**Description**

delete.response returns a terms object for the same model but with no response variable.

drop.terms removes variables from the right-hand side of the model. There is also a "[.terms" method to perform the same function (with keep.response=TRUE).

reformulate creates a formula from a character vector.

**Usage**

```
delete.response(termobj)
```

```
reformulate(termlabels, response = NULL)
```

```
drop.terms(termobj, dropx = NULL, keep.response = FALSE)
```

**Arguments**

termobj      A terms object

termlabels    character vector giving the right-hand side of a model formula.

response      character string, symbol or call giving the left-hand side of a model formula.

dropx          vector of positions of variables to drop from the right-hand side of the model.

keep.response      Keep the response in the resulting object?

**Value**

delete.response and drop.terms return a terms object.

reformulate returns a formula.

**See Also**[terms](#)

**Examples**

```
ff <- y ~ z + x + w
tt <- terms(ff)
tt
delete.response(tt)
drop.terms(tt, 2:3, keep.response = TRUE)
tt[-1]
tt[2:3]
reformulate(attr(tt, "term.labels"))

## keep LHS :
reformulate("x*w", ff[[2]])
fS <- surv(ft, case) ~ a + b
reformulate(c("a", "b*f"), fS[[2]])

stopifnot(identical(      ~ var, reformulate("var")),
           identical(~ a + b + c, reformulate(letters[1:3])),
           identical( y ~ a + b, reformulate(letters[1:2], "y"))
          )
```

---

**dendrapply***Apply a Function to All Nodes of a Dendrogram*

---

**Description**

Apply function FUN to each node of a [dendrogram](#) recursively. When `y <- dendrapply(x, fn)`, then `y` is a dendrogram of the same graph structure as `x` and each for each node, `y.node[j] <- FUN( x.node[j], ...)` (where `y.node[j]` is an (invalid!) notation for the `j`-th node of `y`).

**Usage**

```
dendrapply(X, FUN, ...)
```

**Arguments**

<code>X</code>	an object of class " <a href="#">dendrogram</a> ".
<code>FUN</code>	an R function to be applied to each dendrogram node, typically working on its <a href="#">attributes</a> alone, returning an altered version of the same node.
<code>...</code>	potential further arguments passed to FUN.

**Value**

Usually a dendrogram of the same (graph) structure as `X`. For that, the function must be conceptually of the form `FUN <- function(X) { attributes(X) <- .....; X }`, i.e. returning the node with some attributes added or changed.

**Note**

this is still somewhat experimental, and suggestions for enhancements (or nice examples of usage) are very welcome.

**Author(s)**

Martin Maechler

**See Also**

[as.dendrogram](#), [lapply](#) for applying a function to each component of a list.

**Examples**

```
## a smallish simple dendrogram
dhc <- as.dendrogram(hc <- hclust(dist(USArrests), "ave"))
(dhc21 <- dhc[[2]][[1]])

## too simple:
dendrapply(dhc21, function(n) str(attributes(n)))

## toy example to set colored leaf labels :
local({
  colLab <- function(n) {
    if(is.leaf(n)) {
      a <- attributes(n)
      i <- i+1
      attr(n, "nodePar") <-
        c(a$nodePar, list(lab.col = mycols[i], lab.font= i%%3))
    }
    n
  }
  mycols <- grDevices::rainbow(attr(dhc21,"members"))
  i <- 0
})
dL <- dendrapply(dhc21, colLab)
op <- par(mfrow=2:1)
plot(dhc21)
plot(dL) ## --> colored labels!
par(op)
```

**Description**

Class "dendrogram" provides general functions for handling tree-like structures. It is intended as a replacement for similar functions in hierarchical clustering and classification/regression trees, such that all of these can use the same engine for plotting or cutting trees.

The code is still in testing stage and the API may change in the future.

**Usage**

```

as.dendrogram(object, ...)
## S3 method for class 'hclust':
as.dendrogram(object, hang = -1, ...)

## S3 method for class 'dendrogram':
plot(x, type = c("rectangle", "triangle"),
     center = FALSE,
     edge.root = is.leaf(x) || !is.null(attr(x,"edgetext")),
     nodePar = NULL, edgePar = list(),
     leaflab = c("perpendicular", "textlike", "none"),
     dLeaf = NULL, xlab = "", ylab = "", xaxt = "n", yaxt = "s",
     horiz = FALSE, frame.plot = FALSE, ...)

## S3 method for class 'dendrogram':
cut(x, h, ...)

## S3 method for class 'dendrogram':
print(x, digits, ...)

## S3 method for class 'dendrogram':
rev(x)

## S3 method for class 'dendrogram':
str(object, max.level = 0, digits.d = 3,
     give.attr = FALSE, wid = getOption("width"),
     nest.lev = 0, indent.str = "", stem = "--", ...)

is.leaf(object)

```

**Arguments**

object	any R object that can be made into one of class "dendrogram".
x	object of class "dendrogram".
hang	numeric scalar indicating how the <i>height</i> of leaves should be computed from the heights of their parents; see <a href="#">plot.hclust</a> .
type	type of plot.
center	logical; if TRUE, nodes are plotted centered with respect to the leaves in the branch. Otherwise (default), plot them in the middle of all direct child nodes.
edge.root	logical; if true, draw an edge to the root node.
nodePar	a list of plotting parameters to use for the nodes (see <a href="#">points</a> ) or NULL by default which does not draw symbols at the nodes. The list may contain components named pch, cex, col, and/or bg each of which can have length two for specifying separate attributes for <i>inner</i> nodes and <i>leaves</i> .
edgePar	a list of plotting parameters to use for the edge <a href="#">segments</a> and labels (if there's an edgetext). The list may contain components named col, lty and

	lwd (for the segments), p.col, p.lwd, and p.lty (for the <code>polygon</code> around the text) and t.col for the text color. As with <code>nodePar</code> , each can have length two for differentiating leaves and inner nodes.
leaflab	a string specifying how leaves are labeled. The default "perpendicular" write text vertically (by default). "textlike" writes text horizontally (in a rectangle), and "none" suppresses leaf labels.
dLeaf	a number specifying the distance in user coordinates between the tip of a leaf and its label. If NULL as per default, 3/4 of a letter width or height is used.
horiz	logical indicating if the dendrogram should be drawn <i>horizontally</i> or not.
frame.plot	logical indicating if a box around the plot should be drawn, see <code>plot.default</code> .
h	height at which the tree is cut.
..., xlab, ylab, xaxt, yaxt	graphical parameters, or arguments for other methods.
digits	integer specifying the precision for printing, see <code>print.default</code> .
max.level, digits.d, give.attr, wid, nest.lev, indent.str	arguments to <code>str</code> , see <code>str.default()</code> . Note that <code>give.attr = FALSE</code> still shows <code>height</code> and <code>members</code> attributes for each node.
stem	a string used for <code>str()</code> specifying the <i>stem</i> to use for each dendrogram branch.

## Details

Warning: This documentation is preliminary.

The dendrogram is directly represented as a nested list where each component corresponds to a branch of the tree. Hence, the first branch of tree `z` is `z[[1]]`, the second branch of the corresponding subtree is `z[[1]][[2]]` etc.. Each node of the tree carries some information needed for efficient plotting or cutting as attributes, of which only `members`, `height` and `leaf` for leaves are compulsory:

**members** total number of leaves in the branch

**height** numeric non-negative height at which the node is plotted.

**midpoint** numeric horizontal distance of the node from the left border (the leftmost leaf) of the branch (unit 1 between all leaves). This is used for `plot(*, center=FALSE)`.

**label** character; the label of the node

**x.member** for `cut()` \$upper, the number of *former* members; more generally a substitute for the `members` component used for "horizontal" (when `horiz = FALSE`, else "vertical") alignment.

**edgetext** character; the label for the edge leading to the node

**nodePar** a named list (of length-1 components) specifying node-specific attributes for `points` plotting, see the `nodePar` argument above.

**edgePar** a named list (of length-1 components) specifying attributes for `segments` plotting of the edge leading to the node, and drawing of the `edgetext` if available, see the `edgePar` argument above.

**leaf** logical, if TRUE, the node is a leaf of the tree.

`cut.dendrogram()` returns a list with components `$upper` and `$lower`, the first is a truncated version of the original tree, also of class `dendrogram`, the latter a list with the branches obtained from cutting the tree, each a `dendrogram`.

There are `[]`, `print`, and `str` methods for "dendrogram" objects where the first one (extraction) ensures that selecting sub-branches keeps the class.

Objects of class "hclust" can be converted to class "dendrogram" using method `as.dendrogram`.

`rev.dendrogram` simply returns the dendrogram `x` with reversed nodes, see also `reorder.dendrogram`.

`is.leaf(object)` is logical indicating if `object` is a leaf (the most simple dendrogram). `plotNode()` and `plotNodeLimit()` are helper functions.

### Note

When using `type = "triangle"`, `center = TRUE` often looks better.

### See Also

`order.dendrogram` also on the `labels` method for dendrograms.

### Examples

```
hc <- hclust(dist(USArrests), "ave")
(dend1 <- as.dendrogram(hc)) # "print()" method
str(dend1)                   # "str()" method
str(dend1, max = 2) # only the first two sub-levels

op <- par(mfrow= c(2,2), mar = c(5,2,1,4))
plot(dend1)
## "triangle" type and show inner nodes:
plot(dend1, nodePar=list(pch = c(1,NA), cex=0.8, lab.cex = 0.8),
      type = "t", center=TRUE)
plot(dend1, edgePar=list(col = 1:2, lty = 2:3), dLeaf=1, edge.root = TRUE)
plot(dend1, nodePar=list(pch = 2:1,cex=.4*2:1, col = 2:3), horiz=TRUE)

dend2 <- cut(dend1, h=70)
plot(dend2$upper)
## leaves are wrong horizontally:
plot(dend2$upper, nodePar=list(pch = c(1,7), col = 2:1))
## dend2$lower is *NOT* a dendrogram, but a list of .. :
plot(dend2$lower[[3]], nodePar=list(col=4), horiz = TRUE, type = "tr")
## "inner" and "leaf" edges in different type & color :
plot(dend2$lower[[2]], nodePar=list(col=1),# non empty list
      edgePar = list(lty=1:2, col=2:1), edge.root=TRUE)
par(op)
str(d3 <- dend2$lower[[2]][[2]][[1]])

nP <- list(col=3:2, cex=c(2.0, 0.75), pch= 21:22, bg= c("light blue", "pink"),
```

```

        lab.cex = 0.75, lab.col = "tomato")
plot(d3, nodePar= nP, edgePar = list(col="gray", lwd=2), horiz = TRUE)
addE <- function(n) {
  if(!is.leaf(n)) {
    attr(n, "edgePar") <- list(p.col="plum")
    attr(n, "edgetext") <- paste(attr(n,"members"),"members")
  }
  n
}
d3e <- dendrapply(d3, addE)
plot(d3e, nodePar= nP)
plot(d3e, nodePar= nP, leaflab = "textlike")

```

density

*Kernel Density Estimation***Description**

The function `density` computes kernel density estimates with the given kernel and bandwidth.

**Usage**

```

density(x, bw = "nrd0", adjust = 1,
        kernel = c("gaussian", "epanechnikov", "rectangular",
                   "triangular", "biweight", "cosine", "optcosine"),
        window = kernel, width,
        give.Rkern = FALSE,
        n = 512, from, to, cut = 3, na.rm = FALSE)

```

**Arguments**

<code>x</code>	the data from which the estimate is to be computed.
<code>bw</code>	the smoothing bandwidth to be used. The kernels are scaled such that this is the standard deviation of the smoothing kernel. (Note this differs from the reference books cited below, and from S-PLUS.) <code>bw</code> can also be a character string giving a rule to choose the bandwidth. See <a href="#">bw.nrd</a> . The specified (or computed) value of <code>bw</code> is multiplied by <code>adjust</code> .
<code>adjust</code>	the bandwidth used is actually <code>adjust*bw</code> . This makes it easy to specify values like “half the default” bandwidth.
<code>kernel</code> , <code>window</code>	a character string giving the smoothing kernel to be used. This must be one of "gaussian", "rectangular", "triangular", "epanechnikov", "biweight", "cosine" or "optcosine", with default "gaussian", and may be abbreviated to a unique prefix (single letter). "cosine" is smoother than "optcosine", which is the usual “cosine” kernel in the literature and almost MSE-efficient. However, "cosine" is the version used by S.

<code>width</code>	this exists for compatibility with <code>S</code> ; if given, and <code>bw</code> is not, will set <code>bw</code> to <code>width</code> if this is a character string, or to a kernel-dependent multiple of <code>width</code> if this is numeric.
<code>give.Rkern</code>	logical; if true, <i>no</i> density is estimated, and the “canonical bandwidth” of the chosen kernel is returned instead.
<code>n</code>	the number of equally spaced points at which the density is to be estimated. When <code>n &gt; 512</code> , it is rounded up to the next power of 2 for efficiency reasons ( <code>fft</code> ).
<code>from, to</code>	the left and right-most points of the grid at which the density is to be estimated.
<code>cut</code>	by default, the values of <code>left</code> and <code>right</code> are <code>cut</code> bandwidths beyond the extremes of the data. This allows the estimated density to drop to approximately zero at the extremes.
<code>na.rm</code>	logical; if TRUE, missing values are removed from <code>x</code> . If FALSE any missing values cause an error.

### Details

The algorithm used in `density` disperses the mass of the empirical distribution function over a regular grid of at least 512 points and then uses the fast Fourier transform to convolve this approximation with a discretized version of the kernel and then uses linear approximation to evaluate the density at the specified points.

The statistical properties of a kernel are determined by  $\sigma_K^2 = \int t^2 K(t) dt$  which is always  $= 1$  for our kernels (and hence the bandwidth `bw` is the standard deviation of the kernel) and  $R(K) = \int K^2(t) dt$ .

MSE-equivalent bandwidths (for different kernels) are proportional to  $\sigma_K R(K)$  which is scale invariant and for our kernels equal to  $R(K)$ . This value is returned when `give.Rkern = TRUE`. See the examples for using exact equivalent bandwidths.

Infinite values in `x` are assumed to correspond to a point mass at  $+/-\text{Inf}$  and the density estimate is of the sub-density on  $(-\text{Inf}, +\text{Inf})$ .

### Value

If `give.Rkern` is true, the number  $R(K)$ , otherwise an object with class "density" whose underlying structure is a list containing the following components.

<code>x</code>	the <code>n</code> coordinates of the points where the density is estimated.
<code>y</code>	the estimated density values.
<code>bw</code>	the bandwidth used.
<code>N</code>	the sample size after elimination of missing values.
<code>call</code>	the call which produced the result.
<code>data.name</code>	the deparsed name of the <code>x</code> argument.
<code>has.na</code>	logical, for compatibility (always FALSE).

## References

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole (for S version).
- Scott, D. W. (1992) *Multivariate Density Estimation. Theory, Practice and Visualization*. New York: Wiley.
- Sheather, S. J. and Jones M. C. (1991) A reliable data-based bandwidth selection method for kernel density estimation. *J. Roy. Statist. Soc. B*, 683–690.
- Silverman, B. W. (1986) *Density Estimation*. London: Chapman and Hall.
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer.

## See Also

[bw.nrd](#), [plot.density](#), [hist](#).

## Examples

```
plot(density(c(-20,rep(0,98),20)), xlim = c(-4,4))# IQR = 0

# The Old Faithful geyser data
d <- density(faithful$eruptions, bw = "sj")
d
plot(d)

plot(d, type = "n")
polygon(d, col = "wheat")

## Missing values:
x <- xx <- faithful$eruptions
x[i.out <- sample(length(x), 10)] <- NA
doR <- density(x, bw = 0.15, na.rm = TRUE)
lines(doR, col = "blue")
points(xx[i.out], rep(0.01, 10))

(kernels <- eval(formals(density)$kernel))

## show the kernels in the R parametrization
plot (density(0, bw = 1), xlab = "",
      main="R's density() kernels with bw = 1")
for(i in 2:length(kernels))
  lines(density(0, bw = 1, kern = kernels[i]), col = i)
legend(1.5,.4, legend = kernels, col = seq(kernels),
      lty = 1, cex = .8, y.int = 1)

## show the kernels in the S parametrization
plot(density(0, from=-1.2, to=1.2, width=2, kern="gaussian"), type="l",
      ylim = c(0, 1), xlab="", main="R's density() kernels with width = 1")
for(i in 2:length(kernels))
  lines(density(0, width=2, kern = kernels[i]), col = i)
legend(0.6, 1.0, legend = kernels, col = seq(kernels), lty = 1)
```

```
(RKs <- cbind(sapply(kernels, function(k)density(kern = k, give.Rkern = TRUE))))
100*round(RKs["epanechnikov",]/RKs, 4) ## Efficiencies

if(interactive()) {
bw <- bw.SJ(precip) ## sensible automatic choice
plot(density(precip, bw = bw, n = 2^13),
     main = "same sd bandwidths, 7 different kernels")
for(i in 2:length(kernels))
  lines(density(precip, bw = bw, kern = kernels[i], n = 2^13), col = i)

## Bandwidth Adjustment for "Exactly Equivalent Kernels"
h.f <- sapply(kernels, function(k)density(kern = k, give.Rkern = TRUE))
(h.f <- (h.f["gaussian"] / h.f)^ .2)
## -> 1, 1.01, .995, 1.007,... close to 1 => adjustment barely visible..

plot(density(precip, bw = bw, n = 2^13),
     main = "equivalent bandwidths, 7 different kernels")
for(i in 2:length(kernels))
  lines(density(precip, bw = bw, adjust = h.f[i], kern = kernels[i],
              n = 2^13), col = i)
legend(55, 0.035, legend = kernels, col = seq(kernels), lty = 1)
}
```

---

deriv

*Symbolic and Algorithmic Derivatives of Simple Expressions*


---

## Description

Compute derivatives of simple expressions, symbolically.

## Usage

```
D (expr, name)
deriv(expr, namevec, function.arg, tag = ".expr", hessian = FALSE)
deriv3(expr, namevec, function.arg, tag = ".expr", hessian = TRUE)
```

## Arguments

expr	expression or call to be differentiated.
name, namevec	character vector, giving the variable names (only one for D( )) with respect to which derivatives will be computed.
function.arg	If specified, a character vector of arguments for a function return, or a function (with empty body) or TRUE, the latter indicating that a function with argument names namevec should be used.
tag	character; the prefix to be used for the locally created variables in result.
hessian	a logical value indicating whether the second derivatives should be calculated and incorporated in the return value.

## Details

D is modelled after its S namesake for taking simple symbolic derivatives.

`deriv` is a *generic* function with a default and a `formula` method. It returns a `call` for computing the `expr` and its (partial) derivatives, simultaneously. It uses so-called “*algorithmic derivatives*”. If `function.arg` is a function, its arguments can have default values, see the `fx` example below.

Currently, `deriv.formula` just calls `deriv.default` after extracting the expression to the right of `~`.

`deriv3` and its methods are equivalent to `deriv` and its methods except that `hessian` defaults to `TRUE` for `deriv3`.

## Value

D returns a `call` and therefore can easily be iterated for higher derivatives.

`deriv` and `deriv3` normally return an `expression` object whose evaluation returns the function values with a `"gradient"` attribute containing the gradient matrix. If `hessian` is `TRUE` the evaluation also returns a `"hessian"` attribute containing the Hessian array.

If `function.arg` is specified, `deriv` and `deriv3` return a function with those arguments rather than an expression.

## References

Griewank, A. and Corliss, G. F. (1991) *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM proceedings, Philadelphia.

Bates, D. M. and Chambers, J. M. (1992) *Nonlinear models*. Chapter 10 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

`nlm` and `optim` for numeric minimization which could make use of derivatives,

## Examples

```
## formula argument :
dx2x <- deriv(~ x^2, "x") ; dx2x
## Not run:
expression({
  .value <- x^2
  .grad <- array(0, c(length(.value), 1), list(NULL, c("x")))
  .grad[, "x"] <- 2 * x
  attr(.value, "gradient") <- .grad
  .value
})
## End(Not run)
mode(dx2x)
x <- -1:2
eval(dx2x)
```

```

## Something 'tougher':
trig.exp <- expression(sin(cos(x + y^2)))
( D.sc <- D(trig.exp, "x") )
all.equal(D(trig.exp[[1]], "x"), D.sc)

( dxy <- deriv(trig.exp, c("x", "y")) )
y <- 1
eval(dxy)
eval(D.sc)

## function returned:
deriv((y ~ sin(cos(x) * y)), c("x","y"), func = TRUE)

## function with defaulted arguments:
(fx <- deriv(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
            function(b0, b1, th, x = 1:7){} ) )
fx(2,3,4)

## Higher derivatives
deriv3(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
      c("b0", "b1", "th", "x") )

## Higher derivatives:
DD <- function(expr,name, order = 1) {
  if(order < 1) stop("'order' must be >= 1")
  if(order == 1) D(expr,name)
  else DD(D(expr, name), name, order - 1)
}
DD(expression(sin(x^2)), "x", 3)
## showing the limits of the internal "simplify()" :
## Not run:
-sin(x^2) * (2 * x) * 2 + ((cos(x^2) * (2 * x) * (2 * x) + sin(x^2) *
  2) * (2 * x) + sin(x^2) * (2 * x) * 2)
## End(Not run)

```

---

deviance

*Model Deviance*


---

### Description

Returns the deviance of a fitted model object.

### Usage

```
deviance(object, ...)
```

### Arguments

**object**            an object for which the deviance is desired.  
**...**             additional optional argument.

**Details**

This is a generic function which can be used to extract deviances for fitted models. Consult the individual modeling functions for details on how to use this function.

**Value**

The value of the deviance extracted from the object `object`.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

[df.residual](#), [extractAIC](#), [glm](#), [lm](#).

---

<code>df.residual</code>	<i>Residual Degrees-of-Freedom</i>
--------------------------	------------------------------------

---

**Description**

Returns the residual degrees-of-freedom extracted from a fitted model object.

**Usage**

```
df.residual(object, ...)
```

**Arguments**

`object` an object for which the degrees-of-freedom are desired.  
`...` additional optional arguments.

**Details**

This is a generic function which can be used to extract residual degrees-of-freedom for fitted models. Consult the individual modeling functions for details on how to use this function.

The default method just extracts the `df.residual` component.

**Value**

The value of the residual degrees-of-freedom extracted from the object `x`.

**See Also**

[deviance](#), [glm](#), [lm](#).

---

`diffinv`*Discrete Integration: Inverse of Differencing*

---

**Description**

Computes the inverse function of the lagged differences function `diff`.

**Usage**

```
diffinv(x, lag = 1, differences = 1,  
        xi = rep(0.0, lag*differences*NCOL(x)), ...)
```

**Arguments**

<code>x</code>	a numeric vector, matrix, or time series.
<code>lag</code>	a scalar lag parameter.
<code>differences</code>	an integer representing the order of the difference.
<code>xi</code>	a numeric vector, matrix, or time series containing the initial values for the integrals.
<code>...</code>	arguments passed to or from other methods.

**Details**

`diffinv` is a generic function with methods for class "ts" and default for vectors and matrices.

Missing values are not handled.

**Value**

A numeric vector, matrix, or time series representing the discrete integral of `x`.

**Author(s)**

A. Trapletti

**See Also**

`diff`

**Examples**

```
s <- 1:10  
d <- diff(s)  
diffinv(d, xi = 1)
```

---

 dist *Distance Matrix Computation*


---

**Description**

This function computes and returns the distance matrix computed by using the specified distance measure to compute the distances between the rows of a data matrix.

**Usage**

```
dist(x, method = "euclidean", diag = FALSE, upper = FALSE, p = 2)

as.dist(m, diag = FALSE, upper = FALSE)

## S3 method for class 'dist':
print(x, diag = NULL, upper = NULL,
      digits = getOption("digits"), justify = "none", right = TRUE, ...)

## S3 method for class 'dist':
as.matrix(x)
```

**Arguments**

x	a numeric matrix, data frame or "dist" object.
method	the distance measure to be used. This must be one of "euclidean", "maximum", "manhattan", "canberra", "binary" or "minkowski". Any unambiguous substring can be given.
diag	logical value indicating whether the diagonal of the distance matrix should be printed by <code>print.dist</code> .
upper	logical value indicating whether the upper triangle of the distance matrix should be printed by <code>print.dist</code> .
p	The power of the Minkowski distance.
m	A matrix of distances to be converted to a "dist" object (only the lower triangle is used, the rest is ignored).
digits, justify	passed to <code>format</code> inside of <code>print()</code> .
right, ...	further arguments, passed to the (next) <code>print</code> method.

**Details**

Available distance measures are (written for two vectors  $x$  and  $y$ ):

**euclidean:** Usual square distance between the two vectors (2 norm).

**maximum:** Maximum distance between two components of  $x$  and  $y$  (supremum norm)

**manhattan:** Absolute distance between the two vectors (1 norm).

**canberra:**  $\sum_i |x_i - y_i| / |x_i + y_i|$ . Terms with zero numerator and denominator are omitted from the sum and treated as if the values were missing.

**binary:** (aka *asymmetric binary*): The vectors are regarded as binary bits, so non-zero elements are 'on' and zero elements are 'off'. The distance is the *proportion* of bits in which only one is on amongst those in which at least one is on.

**minkowski:** The  $p$  norm, the  $p$ th root of the sum of the  $p$ th powers of the differences of the components.

Missing values are allowed, and are excluded from all computations involving the rows within which they occur. Further, when `Inf` values are involved, all pairs of values are excluded when their contribution to the distance gave `NaN` or `NA`.

If some columns are excluded in calculating a Euclidean, Manhattan, Canberra or Minkowski distance, the sum is scaled up proportionally to the number of columns used. If all pairs are excluded when calculating a particular distance, the value is `NA`.

The "dist" method of `as.matrix()` and `as.dist()` can be used for conversion between objects of class "dist" and conventional distance matrices.

## Value

An object of class "dist".

The lower triangle of the distance matrix stored by columns in a vector, say `do`. If `n` is the number of observations, i.e., `n <- attr(do, "Size")`, then for  $i < j \leq n$ , the dissimilarity between (row)  $i$  and  $j$  is `do[n*(i-1) - i*(i-1)/2 + j-i]`. The length of the vector is  $n*(n-1)/2$ , i.e., of order  $n^2$ .

The object has the following attributes (besides "class" equal to "dist"):

Size	integer, the number of observations in the dataset.
Labels	optionally, contains the labels, if any, of the observations of the dataset.
Diag, Upper	logicals corresponding to the arguments <code>diag</code> and <code>upper</code> above, specifying how the object should be printed.
call	optionally, the <code>call</code> used to create the object.
method	optionally, the distance method used; resulting from <code>dist()</code> , the <code>(match.arg())</code> method argument.

## References

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
- Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979) *Multivariate Analysis*. Academic Press.
- Borg, I. and Groenen, P. (1997) *Modern Multidimensional Scaling. Theory and Applications*. Springer.

## See Also

`daisy` in the `cluster` package with more possibilities in the case of *mixed* (continuous / categorical) variables. `hclust`.

**Examples**

```

x <- matrix(rnorm(100), nrow=5)
dist(x)
dist(x, diag = TRUE)
dist(x, upper = TRUE)
m <- as.matrix(dist(x))
d <- as.dist(m)
stopifnot(d == dist(x))
names(d) <- LETTERS[1:5]
print(d, digits = 3)

## example of binary and canberra distances.
x <- c(0, 0, 1, 1, 1, 1)
y <- c(1, 0, 1, 1, 0, 1)
dist(rbind(x,y), method="binary")
## answer 0.4 = 2/5
dist(rbind(x,y), method="canberra")
## answer 2 * (6/5)

## Examples involving "Inf" :
## 1)
x[6] <- Inf
(m2 <- rbind(x,y))
dist(m2, method="binary")# warning, answer 0.5 = 2/4
## These all give "Inf":
stopifnot(Inf == dist(m2, method= "euclidean"),
          Inf == dist(m2, method= "maximum"),
          Inf == dist(m2, method= "manhattan"))
## "Inf" is same as very large number:
x1 <- x; x1[6] <- 1e100
stopifnot(dist(cbind(x ,y), method="canberra") ==
          print(dist(cbind(x1,y), method="canberra")))

## 2)
y[6] <- Inf #-> 6-th pair is excluded
dist(rbind(x,y), method="binary") # warning; 0.5
dist(rbind(x,y), method="canberra") # 3
dist(rbind(x,y), method="maximum") # 1
dist(rbind(x,y), method="manhattan")# 2.4

```

**Description**

This extracts coefficients in terms of the original levels of the coefficients rather than the coded variables.

**Usage**

```
dummy.coef(object, ...)

## S3 method for class 'lm':
dummy.coef(object, use.na = FALSE, ...)

## S3 method for class 'aovlist':
dummy.coef(object, use.na = FALSE, ...)
```

**Arguments**

<code>object</code>	a linear model fit.
<code>use.na</code>	logical flag for coefficients in a singular model. If <code>use.na</code> is true, undetermined coefficients will be missing; if false they will get one possible value.
<code>...</code>	arguments passed to or from other methods.

**Details**

A fitted linear model has coefficients for the contrasts of the factor terms, usually one less in number than the number of levels. This function re-expresses the coefficients in the original coding; as the coefficients will have been fitted in the reduced basis, any implied constraints (e.g., zero sum for `contr.helmert` or `contr.sum`) will be respected. There will be little point in using `dummy.coef` for `contr.treatment` contrasts, as the missing coefficients are by definition zero.

The method used has some limitations, and will give incomplete results for terms such as `poly(x, 2)`. However, it is adequate for its main purpose, `aov` models.

**Value**

A list giving for each term the values of the coefficients. For a multistratum `aov` model, such a list for each stratum.

**Warning**

This function is intended for human inspection of the output: it should not be used for calculations. Use coded variables for all calculations.

The results differ from `S` for singular values, where `S` can be incorrect.

**See Also**

[aov, model.tables](#)

**Examples**

```
options(contrasts=c("contr.helmert", "contr.poly"))
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
```

```

K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
npk.aov <- aov(yield ~ block + N*P*K, npk)
dummy.coef(npk.aov)

npk.aovE <- aov(yield ~ N*P*K + Error(block), npk)
dummy.coef(npk.aovE)

```

---

ecdf

*Empirical Cumulative Distribution Function*


---

## Description

Compute or plot an empirical cumulative distribution function.

## Usage

```

ecdf(x)

## S3 method for class 'ecdf':
plot(x, ..., ylab="Fn(x)", verticals = FALSE,
      col.01line = "gray70")

## S3 method for class 'ecdf':
print(x, digits= getOption("digits") - 2, ...)

```

## Arguments

<code>x</code>	numeric vector of “observations” in <code>ecdf</code> ; for the methods, an object of class “ <code>ecdf</code> ”, typically.
<code>...</code>	arguments to be passed to subsequent methods, i.e., <code>plot.stepfun</code> for the <code>plot</code> method.
<code>ylab</code>	label for the y-axis.
<code>verticals</code>	see <code>plot.stepfun</code> .
<code>col.01line</code>	numeric or character specifying the color of the horizontal lines at $y = 0$ and $1$ , see <code>colors</code> .
<code>digits</code>	number of significant digits to use, see <code>print</code> .

**Details**

The e.c.d.f. (empirical cumulative distribution function)  $F_n$  is a step function with jump  $1/n$  at each observation (possibly with multiple jumps at one place if there are ties).

For observations  $\mathbf{x} = (x_1, x_2, \dots, x_n)$ ,  $F_n$  is the fraction of observations less or equal to  $t$ , i.e.,

$$F_n(t) = \#\{x_i \leq t\} / n = \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{[x_i \leq t]}.$$

The function `plot.ecdf` which implements the `plot` method for `ecdf` objects, is implemented via a call to `plot.stepfun`; see its documentation.

**Value**

For `ecdf`, a function of class "ecdf", inheriting from the "stepfun" class.

**Author(s)**

Martin Maechler, <maechler@stat.math.ethz.ch>.

**See Also**

`stepfun`, the more general class of step functions, `approxfun` and `splinefun`.

**Examples**

```
##-- Simple didactical ecdf example:
Fn <- ecdf(rnorm(12))
Fn; summary(Fn)
12*Fn(knots(Fn)) == 1:12 ## == 1:12 if and only if there are no ties !

y <- round(rnorm(12),1); y[3] <- y[1]
Fn12 <- ecdf(y)
Fn12
print(knots(Fn12), dig=2)
12*Fn12(knots(Fn12)) ## ~= 1:12 if there where no ties

summary(Fn12)
summary.stepfun(Fn12)
print(ls.Fn12 <- ls(env= environment(Fn12)))
##[1] "f" "method" "n" "ties" "x" "y" "yleft" "yright"

12 * Fn12((-20:20)/10)

###----- Plotting -----

op <- par(mfrow=c(3,1), mgp=c(1.5, 0.8,0), mar= .1+c(3,3,2,1))

F10 <- ecdf(rnorm(10))
summary(F10)
```

```

plot(F10)
plot(F10, verticals= TRUE, do.p = FALSE)

plot(Fn12)# , lwd=2) dis-regarded
xx <- unique(sort(c(seq(-3,2, length=201), knots(Fn12))))
lines(xx, Fn12(xx), col='blue')
abline(v=knots(Fn12),lty=2,col='gray70')

plot(xx, Fn12(xx), type='b', cex=.1)#- plot.default
plot(Fn12, col.h='red', add= TRUE) #- plot method
abline(v=knots(Fn12),lty=2,col='gray70')
plot(Fn12, verticals=TRUE, col.p='blue', col.h='red',col.v='bisque')
par(op)

##-- this works too (automatic call to ecdf(.)):
plot.ecdf(rnorm(24))

```

---

eff.aovlist

*Compute Efficiencies of Multistratum Analysis of Variance*


---

### Description

Computes the efficiencies of fixed-effect terms in an analysis of variance model with multiple strata.

### Usage

```
eff.aovlist(aovlist)
```

### Arguments

`aovlist`            The result of a call to `aov` with an Error term.

### Details

Fixed-effect terms in an analysis of variance model with multiple strata may be estimable in more than one stratum, in which case there is less than complete information in each. The efficiency is the fraction of the maximum possible precision (inverse variance) obtainable by estimating in just that stratum.

This is used to pick strata in which to estimate terms in `model.tables.aovlist` and elsewhere.

### Value

A matrix giving for each non-pure-error stratum (row) the efficiencies for each fixed-effect term in the model.

### See Also

[aov](#), [model.tables.aovlist](#), [se.contrast.aovlist](#)

**Examples**

```
## for balanced designs all efficiencies are zero or one.
## so as a statistically meaningless test:
options(contrasts=c("contr.helmert", "contr.poly"))
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
npk.aovE <- aov(yield ~ N*P*K + Error(block), npk)
eff.aovlist(npk.aovE)
```

---

effects

*Effects from Fitted Model*


---

**Description**

Returns (orthogonal) effects from a fitted model, usually a linear model. This is a generic function, but currently only has a methods for objects inheriting from classes "lm" and "glm".

**Usage**

```
effects(object, ...)

## S3 method for class 'lm':
effects(object, set.sign = FALSE, ...)
```

**Arguments**

object	an R object; typically, the result of a model fitting function such as <a href="#">lm</a> .
set.sign	logical. If TRUE, the sign of the effects corresponding to coefficients in the model will be set to agree with the signs of the corresponding coefficients, otherwise the sign is arbitrary.
...	arguments passed to or from other methods.

**Details**

For a linear model fitted by [lm](#) or [aov](#), the effects are the uncorrelated single-degree-of-freedom values obtained by projecting the data onto the successive orthogonal subspaces generated by the QR decomposition during the fitting process. The first  $r$  (the rank of the model) are associated with coefficients and the remainder span the space of residuals (but are not associated with particular residuals).

Empty models do not have effects.

**Value**

A (named) numeric vector of the same length as `residuals`, or a matrix if there were multiple responses in the fitted model, in either case of class "coef".

The first  $r$  rows are labelled by the corresponding coefficients, and the remaining rows are unlabelled. Note that in rank-deficient models the “corresponding” coefficients will be in a different order if pivoting occurred.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

`coef`

**Examples**

```
y <- c(1:3,7,5)
x <- c(1:3,6:7)
( ee <- effects(lm(y ~ x)) )
c(round(ee - effects(lm(y+10 ~ I(x-3.8))),3))# just the first is different
```

---

embed

*Embedding a Time Series*

---

**Description**

Embeds the time series `x` into a low-dimensional Euclidean space.

**Usage**

```
embed(x, dimension = 1)
```

**Arguments**

`x` a numeric vector, matrix, or time series.  
`dimension` a scalar representing the embedding dimension.

**Details**

Each row of the resulting matrix consists of sequences  $x[t]$ ,  $x[t-1]$ , ...,  $x[t-dimension+1]$ , where  $t$  is the original index of `x`. If `x` is a matrix, i.e., `x` contains more than one variable, then  $x[t]$  consists of the  $t$ th observation on each variable.

**Value**

A matrix containing the embedded time series `x`.

**Author(s)**

A. Trapletti, B.D. Ripley

**Examples**

```
x <- 1:10
embed(x, 3)
```

---

`expand.model.frame` *Add new variables to a model frame*

---

**Description**

Evaluates new variables as if they had been part of the formula of the specified model. This ensures that the same `na.action` and `subset` arguments are applied and allows, for example, `x` to be recovered for a model using `sin(x)` as a predictor.

**Usage**

```
expand.model.frame(model, extras,
                   envir = environment(formula(model)),
                   na.expand = FALSE)
```

**Arguments**

<code>model</code>	a fitted model
<code>extras</code>	one-sided formula or vector of character strings describing new variables to be added
<code>envir</code>	an environment to evaluate things in
<code>na.expand</code>	logical; see below

**Details**

If `na.expand=FALSE` then NA values in the extra variables will be passed to the `na.action` function used in `model`. This may result in a shorter data frame (with `na.omit`) or an error (with `na.fail`). If `na.expand=TRUE` the returned data frame will have precisely the same rows as `model.frame(model)`, but the columns corresponding to the extra variables may contain NA.

**Value**

A data frame.

**See Also**

[model.frame](#), [predict](#)

**Examples**

```

model <- lm(log(Volume) ~ log(Girth) + log(Height), data=trees)
expand.model.frame(model, ~ Girth) # prints data.frame like

dd <- data.frame(x=1:5, y=rnorm(5), z=c(1,2,NA,4,5))
model <- glm(y ~ x, data=dd, subset=1:4, na.action=na.omit)
expand.model.frame(model, "z", na.expand=FALSE) # = default
expand.model.frame(model, "z", na.expand=TRUE)

```

Exponential

*The Exponential Distribution***Description**

Density, distribution function, quantile function and random generation for the exponential distribution with rate `rate` (i.e., mean  $1/\text{rate}$ ).

**Usage**

```

dexp(x, rate = 1, log = FALSE)
pexp(q, rate = 1, lower.tail = TRUE, log.p = FALSE)
qexp(p, rate = 1, lower.tail = TRUE, log.p = FALSE)
rexp(n, rate = 1)

```

**Arguments**

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If $\text{length}(n) > 1$ , the length is taken to be the number required.
<code>rate</code>	vector of rates.
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

If `rate` is not specified, it assumes the default value of 1.

The exponential distribution with rate  $\lambda$  has density

$$f(x) = \lambda e^{-\lambda x}$$

for  $x \geq 0$ .

**Value**

`dexp` gives the density, `pexp` gives the distribution function, `qexp` gives the quantile function, and `rexp` generates random deviates.

**Note**

The cumulative hazard  $H(t) = -\log(1 - F(t))$  is `-pexp(t, r, lower = FALSE, log = TRUE)`.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`exp` for the exponential function, `dgamma` for the gamma distribution and `dweibull` for the Weibull distribution, both of which generalize the exponential.

**Examples**

```
dexp(1) - exp(-1) #-> 0
```

---

extractAIC	<i>Extract AIC from a Fitted Model</i>
------------	--

---

**Description**

Computes the (generalized) Akaike **A**n **I**nformation **C**riterion for a fitted parametric model.

**Usage**

```
extractAIC(fit, scale, k = 2, ...)
```

**Arguments**

<code>fit</code>	fitted model, usually the result of a fitter like <code>lm</code> .
<code>scale</code>	optional numeric specifying the scale parameter of the model, see <code>scale</code> in <code>step</code> .
<code>k</code>	numeric specifying the “weight” of the <i>equivalent degrees of freedom</i> ( $\equiv$ <code>edf</code> ), part in the AIC formula.
<code>...</code>	further arguments (currently unused in base R).

**Details**

This is a generic function, with methods in base R for "aov", "coxph", "glm", "lm", "negbin" and "survreg" classes.

The criterion used is

$$AIC = -2 \log L + k \times \text{edf},$$

where  $L$  is the likelihood and `edf` the equivalent degrees of freedom (i.e., the number of parameters for usual parametric models) of `fit`.

For linear models with unknown scale (i.e., for `lm` and `ao`),  $-2 \log L$  is computed from the *deviance* and uses a different additive constant to `AIC`.

`k = 2` corresponds to the traditional AIC, using `k = log(n)` provides the BIC (Bayes IC) instead.

For further information, particularly about `scale`, see `step`.

### Value

A numeric vector of length 2, giving

`edf`                    the “equivalent **d**egrees of **f**reedom” of the fitted model `fit`.  
`AIC`                    the (generalized) Akaike Information Criterion for `fit`.

### Note

These functions are used in `add1`, `drop1` and `step` and that may be their main use.

### Author(s)

B. D. Ripley

### References

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer (4th ed).

### See Also

`AIC`, `deviance`, `add1`, `step`

### Examples

```
example(glm)
extractAIC(glm.D93)#>> 5 15.129
```

---

factanal

*Factor Analysis*

---

### Description

Perform maximum-likelihood factor analysis on a covariance matrix or data matrix.

### Usage

```
factanal(x, factors, data = NULL, covmat = NULL, n.obs = NA,
         subset, na.action,
         start = NULL, scores = c("none", "regression", "Bartlett"),
         rotation = "varimax", control = NULL, ...)
```

**Arguments**

<code>x</code>	A formula or a numeric matrix or an object that can be coerced to a numeric matrix.
<code>factors</code>	The number of factors to be fitted.
<code>data</code>	A data frame, used only if <code>x</code> is a formula.
<code>covmat</code>	A covariance matrix, or a covariance list as returned by <code>cov.wt</code> . Of course, correlation matrices are covariance matrices.
<code>n.obs</code>	The number of observations, used if <code>covmat</code> is a covariance matrix.
<code>subset</code>	A specification of the cases to be used, if <code>x</code> is used as a matrix or formula.
<code>na.action</code>	The <code>na.action</code> to be used if <code>x</code> is used as a formula.
<code>start</code>	NULL or a matrix of starting values, each column giving an initial set of uniquenesses.
<code>scores</code>	Type of scores to produce, if any. The default is none, "regression" gives Thompson's scores, "Bartlett" gives Bartlett's weighted least-squares scores. Partial matching allows these names to be abbreviated.
<code>rotation</code>	character. "none" or the name of a function to be used to rotate the factors: it will be called with first argument the loadings matrix, and should return a list with component <code>loadings</code> giving the rotated loadings, or just the rotated loadings.
<code>control</code>	A list of control values, <b>nstart</b> The number of starting values to be tried if <code>start = NULL</code> . Default 1. <b>trace</b> logical. Output tracing information? Default FALSE. <b>lower</b> The lower bound for uniquenesses during optimization. Should be > 0. Default 0.005. <b>opt</b> A list of control values to be passed to <code>optim</code> 's <code>control</code> argument. <b>rotate</b> a list of additional arguments for the rotation function.
<code>...</code>	Components of <code>control</code> can also be supplied as named arguments to <code>factanal</code> .

**Details**

The factor analysis model is

$$x = \Lambda f + e$$

for a  $p$ -element row-vector  $x$ , a  $p \times k$  matrix of *loadings*, a  $k$ -element vector of *scores* and a  $p$ -element vector of errors. None of the components other than  $x$  is observed, but the major restriction is that the scores be uncorrelated and of unit variance, and that the errors be independent with variances  $\Phi$ , the *uniquenesses*. Thus factor analysis is in essence a model for the covariance matrix of  $x$ ,

$$\Sigma = \Lambda' \Lambda + \Psi$$

There is still some indeterminacy in the model for it is unchanged if  $\Lambda$  is replaced by  $G\Lambda$  for any orthogonal matrix  $G$ . Such matrices  $G$  are known as *rotations* (although the term is applied also to non-orthogonal invertible matrices).

If `covmat` is supplied it is used. Otherwise `x` is used if it is a matrix, or a formula `x` is used with data to construct a model matrix, and that is used to construct a covariance matrix. (It makes no sense for the formula to have a response, and all the variables must be numeric.) Once a covariance matrix is found or calculated from `x`, it is converted to a correlation matrix for analysis. The correlation matrix is returned as component `correlation` of the result.

The fit is done by optimizing the log likelihood assuming multivariate normality over the uniquenesses. (The maximizing loadings for given uniquenesses can be found analytically: Lawley & Maxwell (1971, p. 27).) All the starting values supplied in `start` are tried in turn and the best fit obtained is used. If `start = NULL` then the first fit is started at the value suggested by Jöreskog (1963) and given by Lawley & Maxwell (1971, p. 31), and then `control$nstart - 1` other values are tried, randomly selected as equal values of the uniquenesses.

The uniquenesses are technically constrained to lie in  $[0, 1]$ , but near-zero values are problematical, and the optimization is done with a lower bound of `control$lower`, default 0.005 (Lawley & Maxwell, 1971, p. 32).

Scores can only be produced if a data matrix is supplied and used. The first method is the regression method of Thomson (1951), the second the weighted least squares method of Bartlett (1937, 8). Both are estimates of the unobserved scores  $f$ . Thomson's method regresses (in the population) the unknown  $f$  on  $x$  to yield

$$\hat{f} = \Lambda' \Sigma^{-1} x$$

and then substitutes the sample estimates of the quantities on the right-hand side. Bartlett's method minimizes the sum of squares of standardized errors over the choice of  $f$ , given (the fitted)  $\Lambda$ .

If `x` is a formula then the standard NA-handling is applied to the scores (if requested): see [napredict](#).

## Value

An object of class "factanal" with components

<code>loadings</code>	A matrix of loadings, one column for each factor. The factors are ordered in decreasing order of sums of squares of loadings, and given the sign that will make the sum of the loadings positive.
<code>uniquenesses</code>	The uniquenesses computed.
<code>correlation</code>	The correlation matrix used.
<code>criteria</code>	The results of the optimization: the value of the negative log-likelihood and information on the iterations used.
<code>factors</code>	The argument <code>factors</code> .
<code>dof</code>	The number of degrees of freedom of the factor analysis model.
<code>method</code>	The method: always "mle".
<code>scores</code>	If requested, a matrix of scores.
<code>n.obs</code>	The number of observations if available, or NA.
<code>call</code>	The matched call.
<code>na.action</code>	If relevant.
<code>STATISTIC, PVAL</code>	The significance-test statistic and P value, if it can be computed.

**Note**

There are so many variations on factor analysis that it is hard to compare output from different programs. Further, the optimization in maximum likelihood factor analysis is hard, and many other examples we compared had less good fits than produced by this function. In particular, solutions which are Heywood cases (with one or more uniquenesses essentially zero) are much often common than most texts and some other programs would lead one to believe.

**References**

- Bartlett, M. S. (1937) The statistical conception of mental factors. *British Journal of Psychology*, **28**, 97–104.
- Bartlett, M. S. (1938) Methods of estimating mental factors. *Nature*, **141**, 609–610.
- Jöreskog, K. G. (1963) *Statistical Estimation in Factor Analysis*. Almqvist and Wicksell.
- Lawley, D. N. and Maxwell, A. E. (1971) *Factor Analysis as a Statistical Method*. Second edition. Butterworths.
- Thomson, G. H. (1951) *The Factorial Analysis of Human Ability*. London University Press.

**See Also**

[print.loadings](#), [varimax](#), [princomp](#), [ability.cov](#), [Harman23.cor](#), [Harman74.cor](#)

**Examples**

```
# A little demonstration, v2 is just v1 with noise,
# and same for v4 vs. v3 and v6 vs. v5
# Last four cases are there to add noise
# and introduce a positive manifold (g factor)
v1 <- c(1,1,1,1,1,1,1,1,1,1,3,3,3,3,3,4,5,6)
v2 <- c(1,2,1,1,1,1,1,2,1,2,1,3,4,3,3,3,4,6,5)
v3 <- c(3,3,3,3,3,1,1,1,1,1,1,1,1,1,1,5,4,6)
v4 <- c(3,3,4,3,3,1,1,2,1,1,1,1,1,2,1,1,5,6,4)
v5 <- c(1,1,1,1,1,3,3,3,3,3,1,1,1,1,1,6,4,5)
v6 <- c(1,1,1,2,1,3,3,3,4,3,1,1,1,2,1,6,5,4)
m1 <- cbind(v1,v2,v3,v4,v5,v6)
cor(m1)
factanal(m1, factors=3) # varimax is the default
factanal(m1, factors=3, rotation="promax")
# The following shows the g factor as PC1
prcomp(m1)

## formula interface
factanal(~v1+v2+v3+v4+v5+v6, factors = 3,
         scores = "Bartlett")$scores

## a realistic example from Barthlomew (1987, pp. 61-65)
example(ability.cov)
```

---

<code>factor.scope</code>	<i>Compute Allowed Changes in Adding to or Dropping from a Formula</i>
---------------------------	--

---

**Description**

`add.scope` and `drop.scope` compute those terms that can be individually added to or dropped from a model while respecting the hierarchy of terms.

**Usage**

```
add.scope(terms1, terms2)

drop.scope(terms1, terms2)

factor.scope(factor, scope)
```

**Arguments**

<code>terms1</code>	the terms or formula for the base model.
<code>terms2</code>	the terms or formula for the upper ( <code>add.scope</code> ) or lower ( <code>drop.scope</code> ) scope. If missing for <code>drop.scope</code> it is taken to be the null formula, so all terms (except any intercept) are candidates to be dropped.
<code>factor</code>	the "factor" attribute of the terms of the base object.
<code>scope</code>	a list with one or both components <code>drop</code> and <code>add</code> giving the "factor" attribute of the lower and upper scopes respectively.

**Details**

`factor.scope` is not intended to be called directly by users.

**Value**

For `add.scope` and `drop.scope` a character vector of terms labels. For `factor.scope`, a list with components `drop` and `add`, character vectors of terms labels.

**See Also**

[add1](#), [drop1](#), [aov](#), [lm](#)

**Examples**

```
add.scope(~ a + b + c + a:b, ~ (a + b + c)^3)
# [1] "a:c" "b:c"
drop.scope(~ a + b + c + a:b)
# [1] "c" "a:b"
```

family

*Family Objects for Models***Description**

Family objects provide a convenient way to specify the details of the models used by functions such as `glm`. See the documentation for `glm` for the details on how such model fitting takes place.

**Usage**

```
family(object, ...)

binomial(link = "logit")
gaussian(link = "identity")
Gamma(link = "inverse")
inverse.gaussian(link = "1/mu^2")
poisson(link = "log")
quasi(link = "identity", variance = "constant")
quasibinomial(link = "logit")
quasipoisson(link = "log")
```

**Arguments**

link	a specification for the model link function. The <code>gaussian</code> family accepts the links "identity", "log" and "inverse"; the <code>binomial</code> family the links "logit", "probit", "log" and "cloglog" (complementary log-log); the <code>Gamma</code> family the links "inverse", "identity" and "log"; the <code>poisson</code> family the links "log", "identity", and "sqrt" and the <code>inverse.gaussian</code> family the links "1/mu^2", "inverse", "inverse" and "log". The <code>quasi</code> family allows the links "logit", "probit", "cloglog", "identity", "inverse", "log", "1/mu^2" and "sqrt". The function <code>power</code> can also be used to create a power link function for the <code>quasi</code> family.
variance	for all families, other than <code>quasi</code> , the variance function is determined by the family. The <code>quasi</code> family will accept the specifications "constant", " $\mu(1-\mu)$ ", " $\mu$ ", " $\mu^2$ " and " $\mu^3$ " for the variance function.
object	the function <code>family</code> accesses the family objects which are stored within objects created by modelling functions (e.g., <code>glm</code> ).
...	further arguments passed to methods.

**Details**

The `quasibinomial` and `quasipoisson` families differ from the `binomial` and `poisson` families only in that the dispersion parameter is not fixed at one, so they can “model” overdispersion. For the `binomial` case see McCullagh and Nelder (1989, pp. 124–8). Although they

show that there is (under some restrictions) a model with variance proportional to mean as in the quasi-binomial model, note that `glm` does not compute maximum-likelihood estimates in that model. The behaviour of `S` is closer to the quasi-variants.

### Author(s)

The design was inspired by `S` functions of the same names described in Hastie & Pregibon (1992).

### References

- McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models*. London: Chapman and Hall.
- Dobson, A. J. (1983) *An Introduction to Statistical Modelling*. London: Chapman and Hall.
- Cox, D. R. and Snell, E. J. (1981). *Applied Statistics; Principles and Examples*. London: Chapman and Hall.
- Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models*. Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

### See Also

[glm](#), [power](#).

### Examples

```
nf <- gaussian()# Normal family
nf
str(nf)# internal STRucture

gf <- Gamma()
gf
str(gf)
gf$linkinv
gf$variance(-3:4) #- == (.)^2

## quasipoisson. compare with example(glm)
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
d.AD <- data.frame(treatment, outcome, counts)
glm.qD93 <- glm(counts ~ outcome + treatment, family=quasipoisson())
glm.qD93
anova(glm.qD93, test="F")
summary(glm.qD93)
## for Poisson results use
anova(glm.qD93, dispersion = 1, test="Chisq")
summary(glm.qD93, dispersion = 1)

## tests of quasi
x <- rnorm(100)
y <- rpois(100, exp(1+x))
glm(y ~x, family=quasi(var="mu", link="log"))
# which is the same as
```

```

glm(y ~x, family=poisson)
glm(y ~x, family=quasi(var="mu^2", link="log"))
## Not run: glm(y ~x, family=quasi(var="mu^3", link="log")) # should fail
y <- rbinom(100, 1, plogis(x))
# needs to set a starting value for the next fit
glm(y ~x, family=quasi(var="mu(1-mu)", link="logit"), start=c(0,1))

```

FDist

*The F Distribution***Description**

Density, distribution function, quantile function and random generation for the F distribution with `df1` and `df2` degrees of freedom (and optional non-centrality parameter `ncp`).

**Usage**

```

df(x, df1, df2, log = FALSE)
pf(q, df1, df2, ncp=0, lower.tail = TRUE, log.p = FALSE)
qf(p, df1, df2, lower.tail = TRUE, log.p = FALSE)
rf(n, df1, df2)

```

**Arguments**

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>df1, df2</code>	degrees of freedom.
<code>ncp</code>	non-centrality parameter.
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

The F distribution with `df1 =  $n_1$`  and `df2 =  $n_2$`  degrees of freedom has density

$$f(x) = \frac{\Gamma(n_1/2 + n_2/2)}{\Gamma(n_1/2)\Gamma(n_2/2)} \left(\frac{n_1}{n_2}\right)^{n_1/2} x^{n_1/2-1} \left(1 + \frac{n_1 x}{n_2}\right)^{-(n_1+n_2)/2}$$

for  $x > 0$ .

It is the distribution of the ratio of the mean squares of  $n_1$  and  $n_2$  independent standard normals, and hence of the ratio of two independent chi-squared variates each divided by its degrees of freedom. Since the ratio of a normal and the root mean-square of  $m$  independent normals has a Student's  $t_m$  distribution, the square of a  $t_m$  variate has a F distribution on 1 and  $m$  degrees of freedom.

The non-central F distribution is again the ratio of mean squares of independent normals of unit variance, but those in the numerator are allowed to have non-zero means and `ncp` is the sum of squares of the means. See [Chisquare](#) for further details on non-central distributions.

**Value**

df gives the density, pf gives the distribution function qf gives the quantile function, and rf generates random deviates.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[dchisq](#) for chi-squared and [dt](#) for Student's t distributions.

**Examples**

```
## the density of the square of a t_m is 2*dt(x, m)/(2*x)
# check this is the same as the density of F_{1,m}
x <- seq(0.001, 5, len=100)
all.equal(df(x^2, 1, 5), dt(x, 5)/x)

## Identity: qf(2*p - 1, 1, df) == qt(p, df)^2 for p >= 1/2
p <- seq(1/2, .99, length=50); df <- 10
rel.err <- function(x,y) ifelse(x==y,0, abs(x-y)/mean(abs(c(x,y))))
quantile(rel.err(qf(2*p - 1, df1=1, df2=df), qt(p, df)^2), .90) # ~ 7e-9
```

---

 fft

*Fast Discrete Fourier Transform*


---

**Description**

Performs the Fast Fourier Transform of an array.

**Usage**

```
fft(z, inverse = FALSE)
mvfft(z, inverse = FALSE)
```

**Arguments**

z a real or complex array containing the values to be transformed.

inverse if TRUE, the unnormalized inverse transform is computed (the inverse has a + in the exponent of  $e$ , but here, we do *not* divide by  $1/\text{length}(x)$ ).

**Value**

When `z` is a vector, the value computed and returned by `fft` is the unnormalized univariate Fourier transform of the sequence of values in `z`. When `z` contains an array, `fft` computes and returns the multivariate (spatial) transform. If `inverse` is `TRUE`, the (unnormalized) inverse Fourier transform is returned, i.e., if `y <- fft(z)`, then `z` is `fft(y, inverse = TRUE) / length(y)`.

By contrast, `mvfft` takes a real or complex matrix as argument, and returns a similar shaped matrix, but with each column replaced by its discrete Fourier transform. This is useful for analyzing vector-valued series.

The FFT is fastest when the length of the series being transformed is highly composite (i.e., has many factors). If this is not the case, the transform may take a long time to compute and will use a large amount of memory.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Singleton, R. C. (1979) Mixed Radix Fast Fourier Transforms, in *Programs for Digital Signal Processing*, IEEE Digital Signal Processing Committee eds. IEEE Press.

**See Also**

[convolve](#), [nextn](#).

**Examples**

```
x <- 1:4
fft(x)
fft(fft(x), inverse = TRUE)/length(x)
```

---

filter

*Linear Filtering on a Time Series*

---

**Description**

Applies linear filtering to a univariate time series or to each series separately of a multivariate time series.

**Usage**

```
filter(x, filter, method = c("convolution", "recursive"),
       sides = 2, circular = FALSE, init)
```

**Arguments**

<code>x</code>	a univariate or multivariate time series.
<code>filter</code>	a vector of filter coefficients in reverse time order (as for AR or MA coefficients).
<code>method</code>	Either "convolution" or "recursive" (and can be abbreviated). If "convolution" a moving average is used: if "recursive" an autoregression is used.
<code>sides</code>	for convolution filters only. If <code>sides=1</code> the filter coefficients are for past values only; if <code>sides=2</code> they are centred around lag 0. In this case the length of the filter should be odd, but if it is even, more of the filter is forward in time than backward.
<code>circular</code>	for convolution filters only. If <code>TRUE</code> , wrap the filter around the ends of the series, otherwise assume external values are missing (NA).
<code>init</code>	for recursive filters only. Specifies the initial values of the time series just prior to the start value, in reverse time order. The default is a set of zeros.

**Details**

Missing values are allowed in `x` but not in `filter` (where they would lead to missing values everywhere in the output).

Note that there is an implied coefficient 1 at lag 0 in the recursive filter, which gives

$$y_i = x_i + f_1 y_{i-1} + \dots + f_p y_{i-p}$$

No check is made to see if recursive filter is invertible: the output may diverge if it is not.

The convolution filter is

$$y_i = f_1 x_{i+o} + \dots + f_p x_{i+o-p-1}$$

where `o` is the offset: see `sides` for how it is determined.

**Value**

A time series object.

**Note**

`convolve(, type="filter")` uses the FFT for computations and so *may* be faster for long filters on univariate series, but it does not return a time series (and so the time alignment is unclear), nor does it handle missing values. `filter` is faster for a filter of length 100 on a series of length 1000, for example.

**See Also**

[convolve](#), [arima.sim](#)

**Examples**

```
x <- 1:100
filter(x, rep(1, 3))
filter(x, rep(1, 3), sides = 1)
filter(x, rep(1, 3), sides = 1, circular = TRUE)

filter(presidents, rep(1,3))
```

---

fisher.test

*Fisher's Exact Test for Count Data*


---

**Description**

Performs Fisher's exact test for testing the null of independence of rows and columns in a contingency table with fixed marginals.

**Usage**

```
fisher.test(x, y = NULL, workspace = 200000, hybrid = FALSE,
            control = list(), or = 1, alternative = "two.sided",
            conf.int = TRUE, conf.level = 0.95)
```

**Arguments**

x	either a two-dimensional contingency table in matrix form, or a factor object.
y	a factor object; ignored if x is a matrix.
workspace	an integer specifying the size of the workspace used in the network algorithm.
hybrid	a logical indicating whether the exact probabilities (default) or a hybrid approximation thereof should be computed. In the hybrid case, asymptotic chi-squared probabilities are only used provided that the "Cochran" conditions are satisfied.
control	a list with named components for low level algorithm control.
or	the hypothesized odds ratio. Only used in the 2 by 2 case.
alternative	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter. Only used in the 2 by 2 case.
conf.int	logical indicating if a confidence interval should be computed (and returned).
conf.level	confidence level for the returned confidence interval. Only used in the 2 by 2 case.

## Details

If  $x$  is a matrix, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, both  $x$  and  $y$  must be vectors of the same length. Incomplete cases are removed, the vectors are coerced into factor objects, and the contingency table is computed from these.

In the one-sided 2 by 2 cases, p-values are obtained directly using the hypergeometric distribution. Otherwise, computations are based on a C version of the FORTRAN subroutine FEXACT which implements the network developed by Mehta and Patel (1986) and improved by Clarkson, Fan & Joe (1993). The FORTRAN code can be obtained from <http://www.netlib.org/toms/643>. Note this fails (with an error message) when the entries of the table are too large.

In the 2 by 2 case, the null of conditional independence is equivalent to the hypothesis that the odds ratio equals one. Exact inference can be based on observing that in general, given all marginal totals fixed, the first element of the contingency table has a non-central hypergeometric distribution with non-centrality parameter given by the odds ratio (Fisher, 1935).

## Value

A list with class "htest" containing the following components:

p.value	the p-value of the test.
conf.int	a confidence interval for the odds ratio. Only present in the 2 by 2 case.
estimate	an estimate of the odds ratio. Note that the <i>conditional</i> Maximum Likelihood Estimate (MLE) rather than the unconditional MLE (the sample odds ratio) is used. Only present in the 2 by 2 case.
null.value	the odds ratio under the null, or. Only present in the 2 by 2 case.
alternative	a character string describing the alternative hypothesis.
method	the character string "Fisher's Exact Test for Count Data".
data.name	a character string giving the names of the data.

## References

- Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 59–66.
- Fisher, R. A. (1935). The logic of inductive inference. *Journal of the Royal Statistical Society Series A* **98**, 39–54.
- Fisher, R. A. (1962). Confidence limits for a cross-product ratio. *Australian Journal of Statistics* **4**, 41.
- Cyrus R. Mehta & Nitin R. Patel (1986). Algorithm 643. FEXACT: A Fortran subroutine for Fisher's exact test on unordered  $r * c$  contingency tables. *ACM Transactions on Mathematical Software*, **12**, 154–161.
- Douglas B. Clarkson, Yuan-an Fan & Harry Joe (1993). A Remark on Algorithm 643: FEXACT: An Algorithm for Performing Fisher's Exact Test in  $r \times c$  Contingency Tables. *ACM Transactions on Mathematical Software*, **19**, 484–488.

## See Also

[chisq.test](#)

**Examples**

```
## Agresti (1990), p. 61f, Fisher's Tea Drinker
## A British woman claimed to be able to distinguish whether milk or
## tea was added to the cup first. To test, she was given 8 cups of
## tea, in four of which milk was added first. The null hypothesis
## is that there is no association between the true order of pouring
## and the women's guess, the alternative that there is a positive
## association (that the odds ratio is greater than 1).
TeaTasting <-
matrix(c(3, 1, 1, 3),
      nr = 2,
      dimnames = list(Guess = c("Milk", "Tea"),
                      Truth = c("Milk", "Tea")))
fisher.test(TeaTasting, alternative = "greater")
## => p=0.2429, association could not be established

## Fisher (1962), Convictions of like-sex twins in criminals
Convictions <-
matrix(c(2, 10, 15, 3),
      nr = 2,
      dimnames =
      list(c("Dizygotic", "Monozygotic"),
          c("Convicted", "Not convicted")))
Convictions
fisher.test(Convictions, alternative = "less")
fisher.test(Convictions, conf.int = FALSE)
fisher.test(Convictions, conf.level = 0.95)$conf.int
fisher.test(Convictions, conf.level = 0.99)$conf.int
```

---

fitted

*Extract Model Fitted Values*


---

**Description**

`fitted` is a generic function which extracts fitted values from objects returned by modeling functions. `fitted.values` is an alias for it.

All object classes which are returned by model fitting functions should provide a `fitted` method. (Note that the generic is `fitted` and not `fitted.values`.)

Methods can make use of `napredict` methods to compensate for the omission of missing values. The default, `lm` and `glm` methods do.

**Usage**

```
fitted(object, ...)
fitted.values(object, ...)
```

**Arguments**

object            an object for which the extraction of model fitted values is meaningful.  
...                other arguments.

**Value**

Fitted values extracted from the object `x`.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

[coefficients](#), [glm](#), [lm](#), [residuals](#).

---

fivenum

*Tukey Five-Number Summaries*

---

**Description**

Returns Tukey's five number summary (minimum, lower-hinge, median, upper-hinge, maximum) for the input data.

**Usage**

```
fivenum(x, na.rm = TRUE)
```

**Arguments**

`x`                numeric, maybe including `NA`s and `±Infs`.  
`na.rm`            logical; if `TRUE`, all `NA` and `NaN`s are dropped, before the statistics are computed.

**Value**

A numeric vector of length 5 containing the summary information. See [boxplot.stats](#) for more details.

**See Also**

[IQR](#), [boxplot.stats](#), [median](#), [quantile](#), [range](#).

**Examples**

```
fivenum(c(rnorm(100), -1:1/0))
```

---

fligner.test                      *Fligner-Killeen Test for Homogeneity of Variances*

---

### Description

Performs a Fligner-Killeen (median) test of the null that the variances in each of the groups (samples) are the same.

### Usage

```
fligner.test(x, ...)

## Default S3 method:
fligner.test(x, g, ...)

## S3 method for class 'formula':
fligner.test(formula, data, subset, na.action, ...)
```

### Arguments

<code>x</code>	a numeric vector of data values, or a list of numeric data vectors.
<code>g</code>	a vector or factor object giving the group for the corresponding elements of <code>x</code> . Ignored if <code>x</code> is a list.
<code>formula</code>	a formula of the form <code>lhs ~ rhs</code> where <code>lhs</code> gives the data values and <code>rhs</code> the corresponding groups.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

### Details

If `x` is a list, its elements are taken as the samples to be compared for homogeneity of variances, and hence have to be numeric data vectors. In this case, `g` is ignored, and one can simply use `fligner.test(x)` to perform the test. If the samples are not yet contained in a list, use `fligner.test(list(x, ...))`.

Otherwise, `x` must be a numeric data vector, and `g` must be a vector or factor object of the same length as `x` giving the group for the corresponding elements of `x`.

The Fligner-Killeen (median) test has been determined in a simulation study as one of the many tests for homogeneity of variances which is most robust against departures from normality, see Conover, Johnson & Johnson (1981). It is a  $k$ -sample simple linear rank which uses the ranks of the absolute values of the centered samples and weights  $a(i) = \text{qnorm}((1+i/(n+1))/2)$ . The version implemented here uses median centering in each of the samples (F-K:med  $X^2$  in the reference).

**Value**

A list of class "htest" containing the following components:

statistic	the Fligner-Killeen:med $X^2$ test statistic.
parameter	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
p.value	the p-value of the test.
method	the character string "Fligner-Killeen test for homogeneity of variances".
data.name	a character string giving the names of the data.

**References**

William J. Conover & Mark E. Johnson & Myrle M. Johnson (1981). A comparative study of tests for homogeneity of variances, with applications to the outer continental shelf bidding data. *Technometrics* **23**, 351–361.

**See Also**

[ansari.test](#) and [mood.test](#) for rank-based two-sample test for a difference in scale parameters; [var.test](#) and [bartlett.test](#) for parametric tests for the homogeneity of variances.

**Examples**

```
plot(count ~ spray, data = InsectSprays)
fligner.test(InsectSprays$count, InsectSprays$spray)
fligner.test(count ~ spray, data = InsectSprays)
## Compare this to bartlett.test()
```

---

formula

*Model Formulae*

---

**Description**

The generic function `formula` and its specific methods provide a way of extracting formulae which have been included in other objects.

`as.formula` is almost identical, additionally preserving attributes when `object` already inherits from "formula". The default value of the `env` argument is used only when the formula would otherwise lack an environment.

**Usage**

```
formula(x, ...)
as.formula(object, env = parent.frame())
```

## Arguments

<code>x</code> , object	R object.
<code>...</code>	further arguments passed to or from other methods.
<code>env</code>	the environment to associate with the result.

## Details

The models fit by, e.g., the `lm` and `glm` functions are specified in a compact symbolic form. The `~` operator is basic in the formation of such models. An expression of the form `y ~ model` is interpreted as a specification that the response `y` is modelled by a linear predictor specified symbolically by `model`. Such a model consists of a series of terms separated by `+` operators. The terms themselves consist of variable and factor names separated by `:` operators. Such a term is interpreted as the interaction of all the variables and factors appearing in the term.

In addition to `+` and `:`, a number of other operators are useful in model formulae. The `*` operator denotes factor crossing: `a*b` interpreted as `a+b+a:b`. The `^` operator indicates crossing to the specified degree. For example `(a+b+c)^2` is identical to `(a+b+c)*(a+b+c)` which in turn expands to a formula containing the main effects for `a`, `b` and `c` together with their second-order interactions. The `%in%` operator indicates that the terms on its left are nested within those on the right. For example `a + b %in% a` expands to the formula `a + a:b`. The `-` operator removes the specified terms, so that `(a+b+c)^2 - a:b` is identical to `a + b + c + b:c + a:c`. It can also be used to remove the intercept term: `y ~ x - 1` is a line through the origin. A model with no intercept can be also specified as `y ~ x + 0` or `y ~ 0 + x`.

While formulae usually involve just variable and factor names, they can also involve arithmetic expressions. The formula `log(y) ~ a + log(x)` is quite legal. When such arithmetic expressions involve operators which are also used symbolically in model formulae, there can be confusion between arithmetic and symbolic operator use.

To avoid this confusion, the function `I()` can be used to bracket those portions of a model formula where the operators are used in their arithmetic sense. For example, in the formula `y ~ a + I(b+c)`, the term `b+c` is to be interpreted as the sum of `b` and `c`.

As from R 1.8.0 variable names can be quoted by backticks ``like this`` in formulae, although there is no guarantee that all code using formulae will accept such non-syntactic names.

When `formula` is called on a fitted model object, either a specific method is used (such as that for class `"nls"`) or the default method. The default first looks for a `"formula"` component of the object (and evaluates it), then a `"terms"` component, then a `formula` parameter of the call (and evaluates its value) and finally a `"formula"` attribute.

## Value

All the functions above produce an object of class `"formula"` which contains a symbolic model formula.

## Environments

A formula object has an associated environment, and this environment (rather than the parent environment) is used by `model.frame` to evaluate variables that are not found in the supplied `data` argument.

Formulas created with the `~` operator use the environment in which they were created. Formulas created with `as.formula` will use the `env` argument for their environment. Pre-existing formulas extracted with `as.formula` will only have their environment changed if `env` is given explicitly.

## References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models*. Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

I.

For formula manipulation: [terms](#), and [all.vars](#); for typical use: [lm](#), [glm](#), and [coplot](#).

## Examples

```
class(fo <- y ~ x1*x2) # "formula"
fo
typeof(fo)# R internal : "language"
terms(fo)

environment(fo)
environment(as.formula("y ~ x"))
environment(as.formula("y ~ x",env=new.env()))

## Create a formula for a model with a large number of variables:
xnam <- paste("x", 1:25, sep="")
(fmla <- as.formula(paste("y ~ ", paste(xnam, collapse= "+"))))
```

---

formula.nls

*Extract Model Formula from nls Object*

---

## Description

Returns the model used to fit object.

## Usage

```
## S3 method for class 'nls':
formula(x, ...)
```

## Arguments

`x` an object inheriting from class "nls", representing a nonlinear least squares fit.

`...` further arguments passed to or from other methods.

**Value**

a formula representing the model used to obtain object.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [formula](#)

**Examples**

```
fml <- nls(circumference ~ A/(1+exp((B-age)/C)), Orange,
          start = list(A=160, B=700, C = 350))
formula(fml)
```

---

<code>friedman.test</code>	<i>Friedman Rank Sum Test</i>
----------------------------	-------------------------------

---

**Description**

Performs a Friedman rank sum test with unreplicated blocked data.

**Usage**

```
friedman.test(y, ...)

## Default S3 method:
friedman.test(y, groups, blocks, ...)

## S3 method for class 'formula':
friedman.test(formula, data, subset, na.action, ...)
```

**Arguments**

<code>y</code>	either a numeric vector of data values, or a data matrix.
<code>groups</code>	a vector giving the group for the corresponding elements of <code>y</code> if this is a vector; ignored if <code>y</code> is a matrix. If not a factor object, it is coerced to one.
<code>blocks</code>	a vector giving the block for the corresponding elements of <code>y</code> if this is a vector; ignored if <code>y</code> is a matrix. If not a factor object, it is coerced to one.
<code>formula</code>	a formula of the form $a \sim b \mid c$ , where <code>a</code> , <code>b</code> and <code>c</code> give the data values and corresponding groups and blocks, respectively.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

**Details**

`friedman.test` can be used for analyzing unreplicated complete block designs (i.e., there is exactly one observation in `y` for each combination of levels of `groups` and `blocks`) where the normality assumption may be violated.

The null hypothesis is that apart from an effect of `blocks`, the location parameter of `y` is the same in each of the `groups`.

If `y` is a matrix, `groups` and `blocks` are obtained from the column and row indices, respectively. NA's are not allowed in `groups` or `blocks`; if `y` contains NA's, corresponding blocks are removed.

**Value**

A list with class "htest" containing the following components:

<code>statistic</code>	the value of Friedman's chi-squared statistic.
<code>parameter</code>	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>method</code>	the character string "Friedman rank sum test".
<code>data.name</code>	a character string giving the names of the data.

**References**

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 139–146.

**See Also**

[quade.test](#).

**Examples**

```
## Hollander & Wolfe (1973), p. 140ff.
## Comparison of three methods ("round out", "narrow angle", and
## "wide angle") for rounding first base. For each of 18 players
## and the three method, the average time of two runs from a point on
## the first base line 35ft from home plate to a point 15ft short of
## second base is recorded.
RoundingTimes <-
matrix(c(5.40, 5.50, 5.55,
        5.85, 5.70, 5.75,
        5.20, 5.60, 5.50,
        5.55, 5.50, 5.40,
        5.90, 5.85, 5.70,
        5.45, 5.55, 5.60,
        5.40, 5.40, 5.35,
        5.45, 5.50, 5.35,
        5.25, 5.15, 5.00,
        5.85, 5.80, 5.70,
        5.25, 5.20, 5.10,
```

```

      5.65, 5.55, 5.45,
      5.60, 5.35, 5.45,
      5.05, 5.00, 4.95,
      5.50, 5.50, 5.40,
      5.45, 5.55, 5.50,
      5.55, 5.55, 5.35,
      5.45, 5.50, 5.55,
      5.50, 5.45, 5.25,
      5.65, 5.60, 5.40,
      5.70, 5.65, 5.55,
      6.30, 6.30, 6.25),
  nr = 22,
  byrow = TRUE,
  dimnames = list(1 : 22,
                  c("Round Out", "Narrow Angle", "Wide Angle"))
friedman.test(RoundingTimes)
## => strong evidence against the null that the methods are equivalent
##   with respect to speed

wb <- aggregate(warpbreaks$breaks,
                by = list(w = warpbreaks$wool,
                          t = warpbreaks$tension),
                FUN = mean)
wb
friedman.test(wb$x, wb$w, wb$t)
friedman.test(x ~ w | t, data = wb)

```

---

ftable

*Flat Contingency Tables*


---

## Description

Create “flat” contingency tables.

## Usage

```
ftable(x, ...)
```

```
## Default S3 method:
```

```
ftable(..., exclude = c(NA, NaN), row.vars = NULL, col.vars = NULL)
```

## Arguments

<code>x, ...</code>	R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, or a contingency table object of class "table" or "ftable".
<code>exclude</code>	values to use in the exclude argument of factor when interpreting non-factor objects.

<code>row.vars</code>	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the rows of the flat contingency table.
<code>col.vars</code>	a vector of integers giving the numbers of the variables, or a character vector giving the names of the variables to be used for the columns of the flat contingency table.

## Details

`ftable` creates “flat” contingency tables. Similar to the usual contingency tables, these contain the counts of each combination of the levels of the variables (factors) involved. This information is then re-arranged as a matrix whose rows and columns correspond to unique combinations of the levels of the row and column variables (as specified by `row.vars` and `col.vars`, respectively). The combinations are created by looping over the variables in reverse order (so that the levels of the “left-most” variable vary the slowest). Displaying a contingency table in this flat matrix form (via `print.ftable`, the print method for objects of class “`ftable`”) is often preferable to showing it as a higher-dimensional array.

`ftable` is a generic function. Its default method, `ftable.default`, first creates a contingency table in array form from all arguments except `row.vars` and `col.vars`. If the first argument is of class “`table`”, it represents a contingency table and is used as is; if it is a flat table of class “`ftable`”, the information it contains is converted to the usual array representation using `as.ftable`. Otherwise, the arguments should be R objects which can be interpreted as factors (including character strings), or a list (or data frame) whose components can be so interpreted, which are cross-tabulated using `table`. Then, the arguments `row.vars` and `col.vars` are used to collapse the contingency table into flat form. If neither of these two is given, the last variable is used for the columns. If both are given and their union is a proper subset of all variables involved, the other variables are summed out.

When the arguments are R expressions interpreted as factors, additional arguments will be passed to `table` to control how the variable names are displayed; see the last example below.

Function `ftable.formula` provides a formula method for creating flat contingency tables.

## Value

`ftable` returns an object of class “`ftable`”, which is a matrix with counts of each combination of the levels of variables with information on the names and levels of the (row and columns) variables stored as attributes “`row.vars`” and “`col.vars`”.

## See Also

`ftable.formula` for the formula interface (which allows a `data = .` argument); `read.ftable` for information on reading, writing and coercing flat contingency tables; `table` for “ordinary” cross-tabulation; `xtabs` for formula-based cross-tabulation.

## Examples

```
## Start with a contingency table.
ftable(Titanic, row.vars = 1:3)
ftable(Titanic, row.vars = 1:2, col.vars = "Survived")
```

```
ftable(Titanic, row.vars = 2:1, col.vars = "Survived")

## Start with a data frame.
x <- ftable(mtcars[c("cyl", "vs", "am", "gear")])
x
ftable(x, row.vars = c(2, 4))

## Start with expressions, use table()'s "dnn" to change labels
ftable(mtcars$cyl, mtcars$vs, mtcars$am, mtcars$gear, row.vars = c(2, 4),
       dnn = c("Cylinders", "V/S", "Transmission", "Gears"))
```

---

<code>ftable.formula</code>	<i>Formula Notation for Flat Contingency Tables</i>
-----------------------------	---

---

## Description

Produce or manipulate a flat contingency table using formula notation.

## Usage

```
## S3 method for class 'formula':
ftable(formula, data = NULL, subset, na.action, ...)
```

## Arguments

<code>formula</code>	a formula object with both left and right hand sides specifying the column and row variables of the flat table.
<code>data</code>	a data frame, list or environment containing the variables to be cross-tabulated, or a contingency table (see below).
<code>subset</code>	an optional vector specifying a subset of observations to be used. Ignored if <code>data</code> is a contingency table.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Ignored if <code>data</code> is a contingency table.
<code>...</code>	further arguments to the default <code>ftable</code> method may also be passed as arguments, see <a href="#">ftable.default</a> .

## Details

This is a method of the generic function [ftable](#).

The left and right hand side of `formula` specify the column and row variables, respectively, of the flat contingency table to be created. Only the `+` operator is allowed for combining the variables. A `.` may be used once in the formula to indicate inclusion of all the “remaining” variables.

If `data` is an object of class `"table"` or an array with more than 2 dimensions, it is taken as a contingency table, and hence all entries should be nonnegative. Otherwise, if it is not a flat contingency table (i.e., an object of class `"ftable"`), it should be a data frame or matrix, list or

environment containing the variables to be cross-tabulated. In this case, `na.action` is applied to the data to handle missing values, and, after possibly selecting a subset of the data as specified by the `subset` argument, a contingency table is computed from the variables.

The contingency table is then collapsed to a flat table, according to the row and column variables specified by `formula`.

### Value

A flat contingency table which contains the counts of each combination of the levels of the variables, collapsed into a matrix for suitably displaying the counts.

### See Also

`ftable`, `ftable.default`; `table`.

### Examples

```
Titanic
x <- ftable(Survived ~ ., data = Titanic)
x
ftable(Sex ~ Class + Age, data = x)
```

---

GammaDist

*The Gamma Distribution*

---

### Description

Density, distribution function, quantile function and random generation for the Gamma distribution with parameters `shape` and `scale`.

### Usage

```
dgamma(x, shape, rate = 1, scale = 1/rate, log = FALSE)
pgamma(q, shape, rate = 1, scale = 1/rate, lower.tail = TRUE,
       log.p = FALSE)
qgamma(p, shape, rate = 1, scale = 1/rate, lower.tail = TRUE,
       log.p = FALSE)
rgamma(n, shape, rate = 1, scale = 1/rate)
```

### Arguments

`x`, `q`            vector of quantiles.  
`p`                 vector of probabilities.  
`n`                 number of observations. If `length(n) > 1`, the length is taken to be the number required.  
`rate`             an alternative way to specify the scale.  
`shape`, `scale`    shape and scale parameters.

`log`, `log.p` logical; if TRUE, probabilities `p` are given as `log(p)`.  
`lower.tail` logical; if TRUE (default), probabilities are  $P[X \leq x]$ , otherwise,  $P[X > x]$ .

### Details

If `scale` is omitted, it assumes the default value of 1.

The Gamma distribution with parameters `shape` =  $\alpha$  and `scale` =  $\sigma$  has density

$$f(x) = \frac{1}{\sigma^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\sigma}$$

for  $x > 0$ ,  $\alpha > 0$  and  $\sigma > 0$ . The mean and variance are  $E(X) = \alpha\sigma$  and  $Var(X) = \alpha\sigma^2$ .

`pgamma()` uses algorithm AS 239, see the references.

### Value

`dgamma` gives the density, `pgamma` gives the distribution function `qgamma` gives the quantile function, and `rgamma` generates random deviates.

### Note

The S parametrization is via `shape` and `rate`: S has no `scale` parameter.

The cumulative hazard  $H(t) = -\log(1 - F(t))$  is `-pgamma(t, ..., lower = FALSE, log = TRUE)`.

`pgamma` is closely related to the incomplete gamma function. As defined by Abramowitz and Stegun 6.5.1

$$P(a, x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt$$

$P(a, x)$  is `pgamma(x, a)`. Other authors (for example Karl Pearson in his 1922 tables) omit the normalizing factor, defining the incomplete gamma function as `pgamma(x, a) * gamma(a)`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Shea, B. L. (1988) Algorithm AS 239, Chi-squared and Incomplete Gamma Integral, *Applied Statistics (JRSS C)* **37**, 466–473.

Abramowitz, M. and Stegun, I. A. (1972) *Handbook of Mathematical Functions*. New York: Dover. Chapter 6: Gamma and Related Functions.

### See Also

[gamma](#) for the Gamma function, [dbeta](#) for the Beta distribution and [dchisq](#) for the chi-squared distribution which is a special case of the Gamma distribution.

**Examples**

```
-log(dgamma(1:4, shape=1))
p <- (1:9)/10
pgamma(qgamma(p, shape=2), shape=2)
1 - 1/exp(qgamma(p, shape=1))
```

Geometric

*The Geometric Distribution***Description**

Density, distribution function, quantile function and random generation for the geometric distribution with parameter `prob`.

**Usage**

```
dgeom(x, prob, log = FALSE)
pgeom(q, prob, lower.tail = TRUE, log.p = FALSE)
qgeom(p, prob, lower.tail = TRUE, log.p = FALSE)
rgeom(n, prob)
```

**Arguments**

<code>x</code> , <code>q</code>	vector of quantiles representing the number of failures in a sequence of Bernoulli trials before success occurs.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>prob</code>	probability of success in each trial.
<code>log</code> , <code>log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

The geometric distribution with `prob = p` has density

$$p(x) = p(1 - p)^x$$

for  $x = 0, 1, 2, \dots$

If an element of `x` is not integer, the result of `pgeom` is zero, with a warning.

The quantile is defined as the smallest value  $x$  such that  $F(x) \geq p$ , where  $F$  is the distribution function.

**Value**

`dgeom` gives the density, `pgeom` gives the distribution function, `qgeom` gives the quantile function, and `rgeom` generates random deviates.

**See Also**

[dnbinom](#) for the negative binomial which generalizes the geometric distribution.

**Examples**

```
qgeom((1:9)/10, prob = .2)
Ni <- rgeom(20, prob = 1/4); table(factor(Ni, 0:max(Ni)))
```

---

getInitial

*Get Initial Parameter Estimates*


---

**Description**

This function evaluates initial parameter estimates for a nonlinear regression model. If `data` is a parameterized data frame or `pframe` object, its `parameters` attribute is returned. Otherwise the object is examined to see if it contains a call to a `selfStart` object whose `initial` attribute can be evaluated.

**Usage**

```
getInitial(object, data, ...)
```

**Arguments**

<code>object</code>	a formula or a <code>selfStart</code> model that defines a nonlinear regression model
<code>data</code>	a data frame in which the expressions in the formula or arguments to the <code>selfStart</code> model can be evaluated
<code>...</code>	optional additional arguments

**Value**

A named numeric vector or list of starting estimates for the parameters. The construction of many `selfStart` models is such that these "starting" estimates are, in fact, the converged parameter estimates.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [selfStart](#), [selfStart.default](#), [selfStart.formula](#)

**Examples**

```
PurTrt <- Puromycin[ Puromycin$state == "treated", ]
getInitial( rate ~ SSmicmen( conc, Vm, K ), PurTrt )
```

**Description**

glm is used to fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution.

**Usage**

```
glm(formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart,
    offset, control = glm.control(...), model = TRUE,
    method = "glm.fit", x = FALSE, y = TRUE, contrasts = NULL, ...)

glm.fit(x, y, weights = rep(1, nobs),
        start = NULL, etastart = NULL, mustart = NULL,
        offset = rep(0, nobs), family = gaussian(),
        control = glm.control(), intercept = TRUE)

## S3 method for class 'glm':
weights(object, type = c("prior", "working"), ...)
```

**Arguments**

formula	a symbolic description of the model to be fit. The details of model specification are given below.
family	a description of the error distribution and link function to be used in the model. This can be a character string naming a family function, a family function or the result of a call to a family function. (See <a href="#">family</a> for details of family functions.)
data	an optional data frame containing the variables in the model. If not found in data, the variables are taken from <code>environment(formula)</code> , typically the environment from which <code>glm</code> is called.
weights	an optional vector of weights to be used in the fitting process.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the <code>na.action</code> setting of <a href="#">options</a> , and is <code>na.fail</code> if that is unset. The “factory-fresh” default is <code>na.omit</code> .
start	starting values for the parameters in the linear predictor.
etastart	starting values for the linear predictor.
mustart	starting values for the vector of means.
offset	this can be used to specify an <i>a priori</i> known component to be included in the linear predictor during fitting.

<code>control</code>	a list of parameters for controlling the fitting process. See the documentation for <a href="#">glm.control</a> for details.
<code>model</code>	a logical value indicating whether <i>model.frame</i> should be included as a component of the returned value.
<code>method</code>	the method to be used in fitting the model. The default method <code>"glm.fit"</code> uses iteratively reweighted least squares (IWLS). The only current alternative is <code>"model.frame"</code> which returns the model frame and does no fitting.
<code>x, y</code>	For <code>glm</code> : logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value. For <code>glm.fit</code> : <code>x</code> is a design matrix of dimension $n * p$ , and <code>y</code> is a vector of observations of length <code>n</code> .
<code>contrasts</code>	an optional list. See the <code>contrasts.arg</code> of <code>model.matrix.default</code> .
<code>object</code>	an object inheriting from class <code>"glm"</code> .
<code>type</code>	character, partial matching allowed. Type of weights to extract from the fitted model object.
<code>intercept</code>	logical. Should an intercept be included?
<code>...</code>	further arguments passed to or from other methods.

### Details

A typical predictor has the form `response ~ terms` where `response` is the (numeric) response vector and `terms` is a series of terms which specifies a linear predictor for `response`. For binomial models the response can also be specified as a `factor` (when the first level denotes failure and all others success) or as a two-column matrix with the columns giving the numbers of successes and failures. A terms specification of the form `first + second` indicates all the terms in `first` together with all the terms in `second` with duplicates removed. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a `terms` object as the formula.

A specification of the form `first:second` indicates the the set of terms obtained by taking the interactions of all terms in `first` with all terms in `second`. The specification `first*second` indicates the *cross* of `first` and `second`. This is the same as `first + second + first:second`.

`glm.fit` and `glm.fit.null` are the workhorse functions: the former calls the latter for a null model (with no intercept).

If more than one of `etastart`, `start` and `mustart` is specified, the first in the list will be used. All of `weights`, `subset`, `offset`, `etastart` and `mustart` are evaluated in the same way as variables in `formula`, that is first in data and then in the environment of `formula`.

### Value

`glm` returns an object of class inheriting from `"glm"` which inherits from the class `"lm"`. See later in this section.

The function `summary` (i.e., `summary.glm`) can be used to obtain or print a summary of the results and the function `anova` (i.e., `anova.glm`) to produce an analysis of variance table.

The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` can be used to extract various useful features of the value returned by `glm`.

`weights` extracts a vector of weights, one for each case in the fit (after subsetting and `na.action`).

An object of class "glm" is a list containing at least the following components:

<code>coefficients</code>	a named vector of coefficients
<code>residuals</code>	the <i>working</i> residuals, that is the residuals in the final iteration of the IWLS fit.
<code>fitted.values</code>	the fitted mean values, obtained by transforming the linear predictors by the inverse of the link function.
<code>rank</code>	the numeric rank of the fitted linear model.
<code>family</code>	the <code>family</code> object used.
<code>linear.predictors</code>	the linear fit on link scale.
<code>deviance</code>	up to a constant, minus twice the maximized log-likelihood. Where sensible, the constant is chosen so that a saturated model has deviance zero.
<code>aic</code>	Akaike's <i>An Information Criterion</i> , minus twice the maximized log-likelihood plus twice the number of coefficients (so assuming that the dispersion is known).
<code>null.deviance</code>	The deviance for the null model, comparable with <code>deviance</code> . The null model will include the offset, and an intercept if there is one in the model
<code>iter</code>	the number of iterations of IWLS used.
<code>weights</code>	the <i>working</i> weights, that is the weights in the final iteration of the IWLS fit.
<code>prior.weights</code>	the case weights initially supplied.
<code>df.residual</code>	the residual degrees of freedom.
<code>df.null</code>	the residual degrees of freedom for the null model.
<code>y</code>	the $y$ vector used. (It is a vector even for a binomial model.)
<code>converged</code>	logical. Was the IWLS algorithm judged to have converged?
<code>boundary</code>	logical. Is the fitted value on the boundary of the attainable values?
<code>call</code>	the matched call.
<code>formula</code>	the formula supplied.
<code>terms</code>	the <code>terms</code> object used.
<code>data</code>	the data argument.
<code>offset</code>	the offset vector used.
<code>control</code>	the value of the <code>control</code> argument used.
<code>method</code>	the name of the fitter function used, in R always "glm.fit".
<code>contrasts</code>	(where relevant) the contrasts used.
<code>xlevels</code>	(where relevant) a record of the levels of the factors used in fitting.

In addition, non-empty fits will have components `qr`, `R` and `effects` relating to the final weighted linear fit.

Objects of class `"glm"` are normally of class `c("glm", "lm")`, that is inherit from class `"lm"`, and well-designed methods for class `"lm"` will be applied to the weighted linear model at the final iteration of IWLS. However, care is needed, as extractor functions for class `"glm"` such as `residuals` and `weights` do **not** just pick out the component of the fit with the same name.

If a `binomial` `glm` model is specified by giving a two-column response, the weights returned by `prior.weights` are the total numbers of cases (factored by the supplied case weights) and the component `y` of the result is the proportion of successes.

### Author(s)

The original R implementation of `glm` was written by Simon Davies working for Ross Ihaka at the University of Auckland, but has since been extensively re-written by members of the R Core team.

The design was inspired by the S function of the same name described in Hastie & Pregibon (1992).

### References

- Dobson, A. J. (1990) *An Introduction to Generalized Linear Models*. London: Chapman and Hall.
- Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models*. Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.
- McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models*. London: Chapman and Hall.
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer.

### See Also

`anova.glm`, `summary.glm`, etc. for `glm` methods, and the generic functions `anova`, `summary`, `effects`, `fitted.values`, and `residuals`. Further, `lm` for non-generalized *linear* models.

`esoph`, `infert` and `predict.glm` have examples of fitting binomial `glms`.

### Examples

```
## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
print(d.AD <- data.frame(treatment, outcome, counts))
glm.D93 <- glm(counts ~ outcome + treatment, family=poisson())
anova(glm.D93)
summary(glm.D93)

## an example with offsets from Venables & Ripley (2002, p.189)

## Not run:
## Need the anorexia data from a recent version of the package 'MASS':
library(MASS)
## End(Not run)
```

```

anorex.1 <- glm(Postwt ~ Prewt + Treat + offset(Prewt),
               family = gaussian, data = anorexia)
summary(anorex.1)

# A Gamma example, from McCullagh & Nelder (1989, pp. 300-2)
clotting <- data.frame(
  u = c(5,10,15,20,30,40,60,80,100),
  lot1 = c(118,58,42,35,27,25,21,19,18),
  lot2 = c(69,35,26,21,18,16,13,12,12))
summary(glm(lot1 ~ log(u), data=clotting, family=Gamma))
summary(glm(lot2 ~ log(u), data=clotting, family=Gamma))

## Not run:
## for an example of the use of a terms object as a formula
demo(glm.vr)
## End(Not run)

```

---

glm.control

*Auxiliary for Controlling GLM Fitting*


---

## Description

Auxiliary function as user interface for `glm` fitting. Typically only used when calling `glm` or `glm.fit`.

## Usage

```
glm.control(epsilon = 1e-8, maxit = 25, trace = FALSE)
```

## Arguments

<code>epsilon</code>	positive convergence tolerance $\epsilon$ ; the iterations converge when $ dev - dev_{old} /( dev  + 0.1) < \epsilon$ .
<code>maxit</code>	integer giving the maximal number of IWLS iterations.
<code>trace</code>	logical indicating if output should be produced for each iteration.

## Details

If `epsilon` is small, it is also used as the tolerance for the least squares solution.

When `trace` is true, calls to `cat` produce the output for each IWLS iteration. Hence, `options(digits = *)` can be used to increase the precision, see the example.

## Value

A list with the arguments as components.

**References**

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models*. Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[glm.fit](#), the fitting procedure used by [glm](#).

**Examples**

```
### A variation on example(glm) :

## Annette Dobson's example ...
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
oo <- options(digits = 12) # to see more when tracing :
glm.D93X <- glm(counts ~ outcome + treatment, family=poisson(),
               trace = TRUE, epsilon = 1e-14)
options(oo)
coef(glm.D93X) # the last two are closer to 0 than in ?glm's glm.D93
# put less so than in R < 1.8.0 when the default was 1e-4
```

---

 glm.summaries

 Accessing Generalized Linear Model Fits
 

---

**Description**

These functions are all [methods](#) for class `glm` or `summary.glm` objects.

**Usage**

```
## S3 method for class 'glm':
family(object, ...)

## S3 method for class 'glm':
residuals(object, type = c("deviance", "pearson", "working",
                          "response", "partial"), ...)
```

**Arguments**

<code>object</code>	an object of class <code>glm</code> , typically the result of a call to <a href="#">glm</a> .
<code>type</code>	the type of residuals which should be returned. The alternatives are: "deviance" (default), "pearson", "working", "response", and "partial".
<code>...</code>	further arguments passed to or from other methods.

## Details

The references define the types of residuals: Davison & Snell is a good reference for the usages of each.

The partial residuals are a matrix of working residuals, with each column formed by omitting a term from the model.

## References

Davison, A. C. and Snell, E. J. (1991) *Residuals and diagnostics*. In: Statistical Theory and Modelling. In Honour of Sir David Cox, FRS, eds. Hinkley, D. V., Reid, N. and Snell, E. J., Chapman & Hall.

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models*. Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

McCullagh P. and Nelder, J. A. (1989) *Generalized Linear Models*. London: Chapman and Hall.

## See Also

[glm](#) for computing `glm.obj`, [anova.glm](#); the corresponding *generic* functions, [summary.glm](#), [coef](#), [deviance](#), [df.residual](#), [effects](#), [fitted](#), [residuals](#).

---

hclust

*Hierarchical Clustering*

---

## Description

Hierarchical cluster analysis on a set of dissimilarities and methods for analyzing it.

## Usage

```
hclust(d, method = "complete", members=NULL)

## S3 method for class 'hclust':
plot(x, labels = NULL, hang = 0.1,
     axes = TRUE, frame.plot = FALSE, ann = TRUE,
     main = "Cluster Dendrogram",
     sub = NULL, xlab = NULL, ylab = "Height", ...)

plclust(tree, hang = 0.1, unit = FALSE, level = FALSE, hmin = 0,
        square = TRUE, labels = NULL, plot. = TRUE,
        axes = TRUE, frame.plot = FALSE, ann = TRUE,
        main = "", sub = NULL, xlab = NULL, ylab = "Height")
```

**Arguments**

<code>d</code>	a dissimilarity structure as produced by <code>dist</code> .
<code>method</code>	the agglomeration method to be used. This should be (an unambiguous abbreviation of) one of "ward", "single", "complete", "average", "mcquitty", "median" or "centroid".
<code>members</code>	NULL or a vector with length size of <code>d</code> . See the Details section.
<code>x, tree</code>	an object of the type produced by <code>hclust</code> .
<code>hang</code>	The fraction of the plot height by which labels should hang below the rest of the plot. A negative value will cause the labels to hang down from 0.
<code>labels</code>	A character vector of labels for the leaves of the tree. By default the row names or row numbers of the original data are used. If <code>labels=FALSE</code> no labels at all are plotted.
<code>axes, frame.plot, ann</code>	logical flags as in <code>plot.default</code> .
<code>main, sub, xlab, ylab</code>	character strings for <code>title</code> . <code>sub</code> and <code>xlab</code> have a non-NULL default when there's a <code>tree\$call</code> .
<code>...</code>	Further graphical arguments.
<code>unit</code>	logical. If true, the splits are plotted at equally-spaced heights rather than at the height in the object.
<code>hmin</code>	numeric. All heights less than <code>hmin</code> are regarded as being <code>hmin</code> : this can be used to suppress detail at the bottom of the tree.
<code>level, square, plot.</code>	as yet unimplemented arguments of <code>plclust</code> for S-PLUS compatibility.

**Details**

This function performs a hierarchical cluster analysis using a set of dissimilarities for the  $n$  objects being clustered. Initially, each object is assigned to its own cluster and then the algorithm proceeds iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster. At each stage distances between clusters are recomputed by the Lance-Williams dissimilarity update formula according to the particular clustering method being used.

A number of different clustering methods are provided. *Ward's* minimum variance method aims at finding compact, spherical clusters. The *complete linkage* method finds similar clusters. The *single linkage* method (which is closely related to the minimal spanning tree) adopts a 'friends of friends' clustering strategy. The other methods can be regarded as aiming for clusters with characteristics somewhere between the single and complete link methods.

If `members != NULL`, then `d` is taken to be a dissimilarity matrix between clusters instead of dissimilarities between singletons and `members` gives the number of observations per cluster. This way the hierarchical cluster algorithm can be "started in the middle of the dendrogram", e.g., in order to reconstruct the part of the tree above a cut (see examples). Dissimilarities between clusters can be efficiently computed (i.e., without `hclust` itself) only for a limited number of distance/linkage combinations, the simplest one being squared Euclidean distance and centroid linkage. In this case the dissimilarities between the clusters are the squared Euclidean distances between cluster means.

In hierarchical cluster displays, a decision is needed at each merge to specify which subtree should go on the left and which on the right. Since, for  $n$  observations there are  $n - 1$  merges, there are  $2^{(n-1)}$  possible orderings for the leaves in a cluster tree, or dendrogram. The algorithm used in `hclust` is to order the subtree so that the tighter cluster is on the left (the last, i.e., most recent, merge of the left subtree is at a lower value than the last merge of the right subtree). Single observations are the tightest clusters possible, and merges involving two observations place them in order by their observation sequence number.

### Value

An object of class **hclust** which describes the tree produced by the clustering process. The object is a list with components:

<code>merge</code>	an $n - 1$ by 2 matrix. Row $i$ of <code>merge</code> describes the merging of clusters at step $i$ of the clustering. If an element $j$ in the row is negative, then observation $-j$ was merged at this stage. If $j$ is positive then the merge was with the cluster formed at the (earlier) stage $j$ of the algorithm. Thus negative entries in <code>merge</code> indicate agglomerations of singletons, and positive entries indicate agglomerations of non-singletons.
<code>height</code>	a set of $n - 1$ non-decreasing real values. The clustering <i>height</i> : that is, the value of the criterion associated with the clustering <code>method</code> for the particular agglomeration.
<code>order</code>	a vector giving the permutation of the original observations suitable for plotting, in the sense that a cluster plot using this ordering and matrix <code>merge</code> will not have crossings of the branches.
<code>labels</code>	labels for each of the objects being clustered.
<code>call</code>	the call which produced the result.
<code>method</code>	the cluster method that has been used.
<code>dist.method</code>	the distance that has been used to create <code>d</code> (only returned if the distance object has a "method" attribute).

There are `print`, `plot` and `identify` (see `identify.hclust`) methods and the `rect.hclust()` function for `hclust` objects. The `plclust()` function is basically the same as the `plot` method, `plot.hclust`, primarily for back compatibility with S-plus. Its extra arguments are not yet implemented.

### Author(s)

The `hclust` function is based on Fortran code contributed to STATLIB by F. Murtagh.

### References

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (S version.)
- Everitt, B. (1974). *Cluster Analysis*. London: Heinemann Educ. Books.
- Hartigan, J. A. (1975). *Clustering Algorithms*. New York: Wiley.
- Sneath, P. H. A. and R. R. Sokal (1973). *Numerical Taxonomy*. San Francisco: Freeman.

- Anderberg, M. R. (1973). *Cluster Analysis for Applications*. Academic Press: New York.
- Gordon, A. D. (1999). *Classification*. Second Edition. London: Chapman and Hall / CRC
- Murtagh, F. (1985). "Multidimensional Clustering Algorithms", in *COMPSTAT Lectures 4*. Wuerzburg: Physica-Verlag (for algorithmic details of algorithms used).

### See Also

[identify.hclust](#), [rect.hclust](#), [cutree](#), [dendrogram](#), [kmeans](#).

### Examples

```
hc <- hclust(dist(USArrests), "ave")
plot(hc)
plot(hc, hang = -1)

## Do the same with centroid clustering and squared Euclidean distance,
## cut the tree into ten clusters and reconstruct the upper part of the
## tree from the cluster centers.
hc <- hclust(dist(USArrests)^2, "cen")
memb <- cutree(hc, k = 10)
cent <- NULL
for(k in 1:10){
  cent <- rbind(cent, colMeans(USArrests[memb == k, , drop = FALSE]))
}
hc1 <- hclust(dist(cent)^2, method = "cen", members = table(memb))
opar <- par(mfrow = c(1, 2))
plot(hc, labels = FALSE, hang = -1, main = "Original Tree")
plot(hc1, labels = FALSE, hang = -1, main = "Re-start from 10 clusters")
par(opar)
```

---

heatmap

*Draw a Heat Map*

---

### Description

A heat map is a false color image (basically `image(t(x))`) with a dendrogram added to the left side and to the top. Typically, reordering of the rows and columns according to some set of values (row or column means) within the restrictions imposed by the dendrogram is carried out.

### Usage

```
heatmap(x, Rowv=NULL, Colv=if(symm)"Rowv" else NULL,
        distfun = dist, hclustfun = hclust,
        reorderfun = function(d,w) reorder(d,w),
        add.expr, symm = FALSE, revC = identical(Colv, "Rowv"),
        scale=c("row", "column", "none"), na.rm = TRUE,
        margins = c(5, 5), ColSideColors, RowSideColors,
        cexRow = 0.2 + 1/log10(nr), cexCol = 0.2 + 1/log10(nc),
```

```
labRow = NULL, labCol = NULL, main = NULL,
xlab = NULL, ylab = NULL,
keep.dendro = FALSE, verbose = getOption("verbose"), ...)
```

### Arguments

<code>x</code>	numeric matrix of the values to be plotted.
<code>Rowv</code>	determines if and how the <i>row</i> dendrogram should be computed and reordered. Either a <a href="#">dendrogram</a> or a vector of values used to reorder the row dendrogram or <code>NA</code> to suppress any row dendrogram (and reordering) or by default, <code>NULL</code> , see <i>Details</i> below.
<code>Colv</code>	determines if and how the <i>column</i> dendrogram should be reordered. Has the same options as the <code>Rowv</code> argument above and <i>additionally</i> when <code>x</code> is a square matrix, <code>Colv = "Rowv"</code> means that columns should be treated identically to the rows.
<code>distfun</code>	function used to compute the distance (dissimilarity) between both rows and columns. Defaults to <a href="#">dist</a> .
<code>hclustfun</code>	function used to compute the hierarchical clustering when <code>Rowv</code> or <code>Colv</code> are not dendrograms. Defaults to <a href="#">hclust</a> .
<code>reorderfun</code>	function( <code>d,w</code> ) of dendrogram and weights for reordering the row and column dendrograms. The default uses <a href="#">reorder.dendrogram</a> .
<code>add.expr</code>	expression that will be evaluated after the call to <code>image</code> . Can be used to add components to the plot.
<code>symm</code>	logical indicating if <code>x</code> should be treated <b>symmetrically</b> ; can only be true when <code>x</code> is a square matrix.
<code>revC</code>	logical indicating if the column order should be <b>reversed</b> for plotting, such that e.g., for the symmetric case, the symmetry axis is as usual.
<code>scale</code>	character indicating if the values should be centered and scaled in either the row direction or the column direction, or none. The default is <code>"row"</code> if <code>symm</code> false, and <code>"none"</code> otherwise.
<code>na.rm</code>	logical indicating whether <code>NA</code> 's should be removed.
<code>margins</code>	numeric vector of length 2 containing the margins (see <a href="#">par(mar=*)</a> ) for column and row names, respectively.
<code>ColSideColors</code>	(optional) character vector of length <code>ncol(x)</code> containing the color names for a horizontal side bar that may be used to annotate the columns of <code>x</code> .
<code>RowSideColors</code>	(optional) character vector of length <code>nrow(x)</code> containing the color names for a vertical side bar that may be used to annotate the rows of <code>x</code> .
<code>cexRow, cexCol</code>	positive numbers, used as <code>cex.axis</code> in for the row or column axis labeling. The defaults currently only use number of rows or columns, respectively.
<code>labRow, labCol</code>	character vectors with row and column labels to use; these default to <code>rownames(x)</code> or <code>colnames(x)</code> , respectively.

<code>main, xlab, ylab</code>	main, x- and y-axis titles; defaults to none.
<code>keep.dendro</code>	logical indicating if the dendrogram(s) should be kept as part of the result (when <code>Rowv</code> and/or <code>Colv</code> are not NA).
<code>verbose</code>	logical indicating if information should be printed.
<code>...</code>	additional arguments passed on to <code>image</code> , e.g., <code>col</code> specifying the colors.

### Details

If either `Rowv` or `Colv` are dendrograms they are honored (and not reordered). Otherwise, dendrograms are computed as `dd <- as.dendrogram(hclustfun(distfun(X)))` where `X` is either `x` or `t(x)`.

If either is a vector (of “weights”) then the appropriate dendrogram is reordered according to the supplied values subject to the constraints imposed by the dendrogram, by `reorder(dd, Rowv)`, in the row case. If either is missing, as by default, then the ordering of the corresponding dendrogram is by the mean value of the rows/columns, i.e., in the case of rows, `Rowv <- rowMeans(x, na.rm=na.rm)`. If either is `NULL`, *no reordering* will be done for the corresponding side.

By default (`scale = "row"`) the rows are scaled to have mean zero and standard deviation one. There is some empirical evidence from genomic plotting that this is useful.

The default colors are not pretty. Consider using enhancements such as the **RColorBrewer** package, <http://cran.r-project.org/src/contrib/PACKAGES.html#RColorBrewer>.

### Value

Invisibly, a list with components

<code>rowInd</code>	row index permutation vector as returned by <code>order.dendrogram</code> .
<code>colInd</code>	column index permutation vector.
<code>Rowv</code>	the row dendrogram; only if input <code>Rowv</code> was not NA and <code>keep.dendro</code> is true.
<code>Colv</code>	the column dendrogram; only if input <code>Colv</code> was not NA and <code>keep.dendro</code> is true.

### Note

Unless `Rowv = NA` (or `Colv = NA`), the original rows and columns are reordered *in any case* to match the dendrogram, e.g., the rows by `order.dendrogram(Rowv)` where `Rowv` is the (possibly `reorder()`ed) row dendrogram.

`heatmap()` uses `layout` and draws the `image` in the lower right corner of a 2x2 layout. Consequentially, it can **not** be used in a multi column/row layout, i.e., when `par(mfrow= *)` or `(mfcol= *)` has been called.

### Author(s)

Andy Liaw, original; R. Gentleman, M. Maechler, W. Huber, revisions.

**See Also**

[image](#), [hclust](#)

**Examples**

```
require(graphics)
x <- as.matrix(mtcars)
rc <- rainbow(nrow(x), start=0, end=.3)
cc <- rainbow(ncol(x), start=0, end=.3)
hv <- heatmap(x, col = cm.colors(256), scale="column",
              RowSideColors = rc, ColSideColors = cc, margin=c(5,10),
              xlab = "specification variables", ylab= "Car Models",
              main = "heatmap(<Mtcars data>, ..., scale = \"column\")")
str(hv) # the two re-ordering index vectors

## no column dendrogram (nor reordering) at all:
heatmap(x, Colv = NA, col = cm.colors(256), scale="column",
        RowSideColors = rc, margin=c(5,10),
        xlab = "specification variables", ylab= "Car Models",
        main = "heatmap(<Mtcars data>, ..., scale = \"column\")")

## "no nothing"
heatmap(x, Rowv = NA, Colv = NA, scale="column",
        main = "heatmap(*, NA, NA) ~= image(t(x))")

round(Ca <- cor(attitude), 2)
symnum(Ca) # simple graphic
heatmap(Ca,          symm = TRUE, margin=c(6,6))# with reorder()
heatmap(Ca, Rowv=FALSE, symm = TRUE, margin=c(6,6))# NO reorder()

## For variable clustering, rather use distance based on cor():
symnum( cU <- cor(USJudgeRatings) )

hU <- heatmap(cU, Rowv = FALSE, symm = TRUE, col = topo.colors(16),
              distfun = function(c) as.dist(1 - c), keep.dendro = TRUE)
## The Correlation matrix with same reordering:
round(100 * cU[hU[[1]], hU[[2]])
## The column dendrogram:
str(hU$Colv)
```

**Description**

Computes Holt-Winters Filtering of a given time series. Unknown parameters are determined by minimizing the squared prediction error.

**Usage**

```
HoltWinters(x, alpha = NULL, beta = NULL, gamma = NULL,
            seasonal = c("additive", "multiplicative"),
            start.periods = 3, l.start = NULL, b.start = NULL,
            s.start = NULL,
            optim.start = c(alpha = 0.3, beta = 0.1, gamma = 0.1),
            optim.control = list())
```

**Arguments**

<code>x</code>	An object of class <code>ts</code>
<code>alpha</code>	<i>alpha</i> parameter of Holt-Winters Filter
<code>beta</code>	<i>beta</i> parameter of Holt-Winters Filter. If set to 0, the function will do exponential smoothing.
<code>gamma</code>	<i>gamma</i> parameter used for the seasonal component. If set to 0, a non-seasonal model is fitted.
<code>seasonal</code>	Character string to select an "additive" (the default) or "multiplicative" seasonal model. The first few characters are sufficient. (Only takes effect if <code>gamma</code> is non-zero).
<code>start.periods</code>	Start periods used in the autodetection of start values. Must be at least 3.
<code>l.start</code>	Start value for level ( $a[0]$ ).
<code>b.start</code>	Start value for trend ( $b[0]$ ).
<code>s.start</code>	Vector of start values for the seasonal component ( $s_1[0] \dots s_p[0]$ )
<code>optim.start</code>	Vector with named components <code>alpha</code> , <code>beta</code> , and <code>gamma</code> containing the starting values for the optimizer. Only the values needed must be specified.
<code>optim.control</code>	Optional list with additional control parameters passed to <code>optim</code> .

**Details**

The additive Holt-Winters prediction function (for time series with period length  $p$ ) is

$$\hat{Y}[t+h] = a[t] + hb[t] + s[t+1+(h-1) \bmod p],$$

where  $a[t]$ ,  $b[t]$  and  $s[t]$  are given by

$$a[t] = \alpha(Y[t] - s[t-p]) + (1-\alpha)(a[t-1] + b[t-1])$$

$$b[t] = \beta(a[t] - a[t-1]) + (1-\beta)b[t-1]$$

$$s[t] = \gamma(Y[t] - a[t]) + (1-\gamma)s[t-p]$$

The multiplicative Holt-Winters prediction function (for time series with period length  $p$ ) is

$$\hat{Y}[t+h] = (a[t] + hb[t]) \times s[t+1+(h-1) \bmod p].$$

where  $a[t]$ ,  $b[t]$  and  $s[t]$  are given by

$$a[t] = \alpha(Y[t]/s[t-p]) + (1 - \alpha)(a[t-1] + b[t-1])$$

$$b[t] = \beta(a[t] - a[t-1]) + (1 - \beta)b[t-1]$$

$$s[t] = \gamma(Y[t]/a[t]) + (1 - \gamma)s[t-p]$$

The function tries to find the optimal values of  $\alpha$  and/or  $\beta$  and/or  $\gamma$  by minimizing the squared one-step prediction error if they are omitted.

For seasonal models, start values for  $a$ ,  $b$  and  $s$  are detected by performing a simple decomposition in trend and seasonal component using moving averages (see function `decompose`) on the `start.periods` first periods (a simple linear regression on the trend component is used for starting level and trend.). For level/trend-models (no seasonal component), start values for  $a$  and  $b$  are `x[2]` and `x[2] - x[1]`, respectively. For level-only models (ordinary exponential smoothing), the start value for  $a$  is `x[1]`.

### Value

An object of class "HoltWinters", a list with components:

<code>fitted</code>	A multiple time series with one column for the filtered series as well as for the level, trend and seasonal components, estimated contemporaneously (that is at time $t$ and not at the end of the series).
<code>x</code>	The original series
<code>alpha</code>	alpha used for filtering
<code>beta</code>	beta used for filtering
<code>coefficients</code>	A vector with named components <code>a</code> , <code>b</code> , <code>s1</code> , ..., <code>sp</code> containing the estimated values for the level, trend and seasonal components
<code>seasonal</code>	The specified <code>seasonal</code> -parameter
<code>SSE</code>	The final sum of squared errors achieved in optimizing
<code>call</code>	The call used

### Author(s)

David Meyer (David.Meyer@wu-wien.ac.at)

### References

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, ONR Research Memorandum, Carnegie Institute 52.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

### See Also

[predict.HoltWinters,optim](#)

**Examples**

```
## Seasonal Holt-Winters
(m <- HoltWinters(co2))
plot(m)
plot(fitted(m))

(m <- HoltWinters(AirPassengers, seasonal = "mult"))
plot(m)

## Non-Seasonal Holt-Winters
x <- uspop + rnorm(uspop, sd = 5)
m <- HoltWinters(x, gamma = 0)
plot(m)

## Exponential Smoothing
m2 <- HoltWinters(x, gamma = 0, beta = 0)
lines(fitted(m2)[,1], col = 3)
```

---

Hypergeometric      *The Hypergeometric Distribution*

---

**Description**

Density, distribution function, quantile function and random generation for the hypergeometric distribution.

**Usage**

```
dhyper(x, m, n, k, log = FALSE)
phyper(q, m, n, k, lower.tail = TRUE, log.p = FALSE)
qhyper(p, m, n, k, lower.tail = TRUE, log.p = FALSE)
rhyper(nn, m, n, k)
```

**Arguments**

<code>x, q</code>	vector of quantiles representing the number of white balls drawn without replacement from an urn which contains both black and white balls.
<code>m</code>	the number of white balls in the urn.
<code>n</code>	the number of black balls in the urn.
<code>k</code>	the number of balls drawn from the urn.
<code>p</code>	probability, it must be between 0 and 1.
<code>nn</code>	number of observations. If <code>length(nn) &gt; 1</code> , the length is taken to be the number required.
<code>log, log.p</code>	logical; if TRUE, probabilities p are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

The hypergeometric distribution is used for sampling *without* replacement. The density of this distribution with parameters  $m$ ,  $n$  and  $k$  (named  $Np$ ,  $N - Np$ , and  $n$ , respectively in the reference below) is given by

$$p(x) = \binom{m}{x} \binom{n}{k-x} / \binom{m+n}{k}$$

for  $x = 0, \dots, k$ .

**Value**

`dhyper` gives the density, `phyper` gives the distribution function, `qhyper` gives the quantile function, and `rhyper` generates random deviates.

**References**

Johnson, N. L., Kotz, S., and Kemp, A. W. (1992) *Univariate Discrete Distributions*, Second Edition. New York: Wiley.

**Examples**

```
m <- 10; n <- 7; k <- 8
x <- 0:(k+1)
rbind(phyper(x, m, n, k), dhyper(x, m, n, k))
all(phyper(x, m, n, k) == cumsum(dhyper(x, m, n, k)))# FALSE
## but error is very small:
signif(phyper(x, m, n, k) - cumsum(dhyper(x, m, n, k)), dig=3)
```

---

identify.hclust      *Identify Clusters in a Dendrogram*

---

**Description**

`identify.hclust` reads the position of the graphics pointer when the (first) mouse button is pressed. It then cuts the tree at the vertical position of the pointer and highlights the cluster containing the horizontal position of the pointer. Optionally a function is applied to the index of data points contained in the cluster.

**Usage**

```
## S3 method for class 'hclust':
identify(x, FUN = NULL, N = 20, MAXCLUSTER = 20, DEV.FUN = NULL, ...)
```

**Arguments**

<code>x</code>	an object of the type produced by <code>hclust</code> .
<code>FUN</code>	(optional) function to be applied to the index numbers of the data points in a cluster (see Details below).
<code>N</code>	the maximum number of clusters to be identified.
<code>MAXCLUSTER</code>	the maximum number of clusters that can be produced by a cut (limits the effective vertical range of the pointer).
<code>DEV.FUN</code>	(optional) integer scalar. If specified, the corresponding graphics device is made active before <code>FUN</code> is applied.
<code>...</code>	further arguments to <code>FUN</code> .

**Details**

By default clusters can be identified using the mouse and an `invisible` list of indices of the respective data points is returned.

If `FUN` is not `NULL`, then the index vector of data points is passed to this function as first argument, see the examples below. The active graphics device for `FUN` can be specified using `DEV.FUN`.

The identification process is terminated by pressing any mouse button other than the first, see also `identify`.

**Value**

Either a list of data point index vectors or a list of return values of `FUN`.

**See Also**

`hclust`, `rect.hclust`

**Examples**

```
## Not run:
hca <- hclust(dist(USArrests))
plot(hca)
(x <- identify(hca)) ## Terminate with 2nd mouse button !!

hci <- hclust(dist(iris[,1:4]))
plot(hci)
identify(hci, function(k) print(table(iris[k,5])))

# open a new device (one for dendrogram, one for bars):
get(getOption("device"))()# << make that narrow (& small) and *beside* 1st one
nD <- dev.cur()          # to be for the barplot
dev.set(dev.prev())# old one for dendrogram
plot(hci)
## select subtrees in dendrogram and "see" the species distribution:
identify(hci, function(k) barplot(table(iris[k,5]),col=2:4), DEV.FUN = nD)
## End(Not run)
```

---

*influence.measures Regression Deletion Diagnostics*

---

**Description**

This suite of functions can be used to compute some of the regression (leave-one-out deletion) diagnostics for linear and generalized linear models discussed in Belsley, Kuh and Welsch (1980), Cook and Weisberg (1982), etc.

**Usage**

```
influence.measures(model)

rstandard(model, ...)
## S3 method for class 'lm':
rstandard(model, infl = lm.influence(model, do.coef = FALSE),
          sd = sqrt(deviance(model)/df.residual(model)), ...)
## S3 method for class 'glm':
rstandard(model, infl = lm.influence(model, do.coef = FALSE), ...)

rstudent(model, ...)
## S3 method for class 'lm':
rstudent(model, infl = lm.influence(model, do.coef = FALSE),
          res = infl$wt.res, ...)
## S3 method for class 'glm':
rstudent(model, infl = influence(model, do.coef = FALSE), ...)

dffits(model, infl = , res = )

dfbeta(model, ...)
## S3 method for class 'lm':
dfbeta(model, infl = lm.influence(model, do.coef = TRUE), ...)

dfbetas(model, ...)
## S3 method for class 'lm':
dfbetas(model, infl = lm.influence(model, do.coef = TRUE), ...)

covratio(model, infl = lm.influence(model, do.coef = FALSE),
          res = weighted.residuals(model))

cooks.distance(model, ...)
## S3 method for class 'lm':
cooks.distance(model, infl = lm.influence(model, do.coef = FALSE),
               res = weighted.residuals(model),
               sd = sqrt(deviance(model)/df.residual(model)),
               hat = infl$hat, ...)
## S3 method for class 'glm':
```

```

cooks.distance(model, infl = influence(model, do.coef = FALSE),
               res = infl$pear.res,
               dispersion = summary(model)$dispersion,
               hat = infl$hat, ...)

hatvalues(model, ...)
## S3 method for class 'lm':
hatvalues(model, infl = lm.influence(model, do.coef = FALSE), ...)

hat(x, intercept = TRUE)

```

### Arguments

model	an R object, typically returned by <code>lm</code> or <code>glm</code> .
infl	influence structure as returned by <code>lm.influence</code> or <code>influence</code> (the latter only for the <code>glm</code> method of <code>rstudent</code> and <code>cooks.distance</code> ).
res	(possibly weighted) residuals, with proper default.
sd	standard deviation to use, see default.
dispersion	dispersion (for <code>glm</code> objects) to use, see default.
hat	hat values $H_{ii}$ , see default.
x	the $X$ or design matrix.
intercept	should an intercept column be pre-pended to <code>x</code> ?
...	further arguments passed to or from other methods.

### Details

The primary high-level function is `influence.measures` which produces a class "infl" object tabular display showing the DFBETAS for each model variable, DFFITS, covariance ratios, Cook's distances and the diagonal elements of the hat matrix. Cases which are influential with respect to any of these measures are marked with an asterisk.

The functions `dfbetas`, `dffits`, `covratio` and `cooks.distance` provide direct access to the corresponding diagnostic quantities. Functions `rstandard` and `rstudent` give the standardized and Studentized residuals respectively. (These re-normalize the residuals to have unit variance, using an overall and leave-one-out measure of the error variance respectively.)

Values for generalized linear models are approximations, as described in Williams (1987) (except that Cook's distances are scaled as  $F$  rather than as chi-square values).

The optional `infl`, `res` and `sd` arguments are there to encourage the use of these direct access functions, in situations where, e.g., the underlying basic influence measures (from `lm.influence` or the generic `influence`) are already available.

Note that cases with `weights == 0` are *dropped* from all these functions, but that if a linear model has been fitted with `na.action = na.exclude`, suitable values are filled in for the cases excluded during fitting.

The function `hat()` exists mainly for S (version 2) compatibility; we recommend using `hatvalues()` instead.

**Note**

For hatvalues, dfbeta, and dfbetas, the method for linear models also works for generalized linear models.

**Author(s)**

Several R core team members and John Fox, originally in his 'car' package.

**References**

- Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.
- Cook, R. D. and Weisberg, S. (1982) *Residuals and Influence in Regression*. London: Chapman and Hall.
- Williams, D. A. (1987) Generalized linear model diagnostics using the deviance and single case deletions. *Applied Statistics* **36**, 181–191.
- Fox, J. (1997) *Applied Regression, Linear Models, and Related Methods*. Sage.
- Fox, J. (2002) *An R and S-Plus Companion to Applied Regression*. Sage Publ.; <http://www.socsci.mcmaster.ca/jfox/Books/Companion/>.

**See Also**

[influence](#) (containing [lm.influence](#)).

**Examples**

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)

inflm.SR <- influence.measures(lm.SR)
which(apply(inflm.SR$sis.inf, 1, any)) # which observations 'are' influential
summary(inflm.SR) # only these
inflm.SR # all
plot(rstudent(lm.SR) ~ hatvalues(lm.SR)) # recommended by some

## The 'infl' argument is not needed, but avoids recomputation:
rs <- rstandard(lm.SR)
iflSR <- influence(lm.SR)
identical(rs, rstandard(lm.SR, infl = iflSR))
## to "see" the larger values:
1000 * round(dfbetas(lm.SR, infl = iflSR), 3)

## Huber's data [Atkinson 1985]
xh <- c(-4:0, 10)
yh <- c(2.48, .73, -.04, -1.44, -1.32, 0)
summary(lmH <- lm(yh ~ xh))
(im <- influence.measures(lmH))
plot(xh,yh, main = "Huber's data: L.S. line and influential obs.")
abline(lmH); points(xh[im$sis.inf], yh[im$sis.inf], pch=20, col=2)
```

integrate

*Integration of One-Dimensional Functions***Description**

Adaptive quadrature of functions of one variable over a finite or infinite interval.

**Usage**

```
integrate(f, lower, upper, subdivisions=100,
          rel.tol = .Machine$double.eps^0.25, abs.tol = rel.tol,
          stop.on.error = TRUE, keep.xy = FALSE, aux = NULL, ...)
```

**Arguments**

<code>f</code>	an R function taking a numeric first argument and returning a numeric vector of the same length. Returning a non-finite element will generate an error.
<code>lower, upper</code>	the limits of integration. Can be infinite.
<code>subdivisions</code>	the maximum number of subintervals.
<code>rel.tol</code>	relative accuracy requested.
<code>abs.tol</code>	absolute accuracy requested.
<code>stop.on.error</code>	logical. If true (the default) an error stops the function. If false some errors will give a result with a warning in the <code>message</code> component.
<code>keep.xy</code>	unused. For compatibility with S.
<code>aux</code>	unused. For compatibility with S.
<code>...</code>	additional arguments to be passed to <code>f</code> . Remember to use argument names <i>not</i> matching those of <code>integrate(.)</code> !

**Details**

If one or both limits are infinite, the infinite range is mapped onto a finite interval.

For a finite interval, globally adaptive interval subdivision is used in connection with extrapolation by the Epsilon algorithm.

`rel.tol` cannot be less than  $\max(50 * .Machine$double.eps, 0.5e-28)$  if `abs.tol`  $\leq 0$ .

**Value**

A list of class "integrate" with components

<code>value</code>	the final estimate of the integral.
<code>abs.error</code>	estimate of the modulus of the absolute error.
<code>subdivisions</code>	the number of subintervals produced in the subdivision process.
<code>message</code>	"OK" or a character string giving the error message.
<code>call</code>	the matched call.

**Note**

Like all numerical integration routines, these evaluate the function on a finite set of points. If the function is approximately constant (in particular, zero) over nearly all its range it is possible that the result and error estimate may be seriously wrong.

When integrating over infinite intervals do so explicitly, rather than just using a large number as the endpoint. This increases the chance of a correct answer – any function whose integral over an infinite interval is finite must be near zero for most of that interval.

**References**

Based on QUADPACK routines `dqags` and `dqagi` by R. Piessens and E. deDoncker-Kapenga, available from Netlib.

See

R. Piessens, E. deDoncker-Kapenga, C. Uberhuber, D. Kahaner (1983) *Quadpack: a Subroutine Package for Automatic Integration*; Springer Verlag.

**See Also**

The function [adapt](#) in the **adapt** package on CRAN, for multivariate integration.

**Examples**

```
integrate(dnorm, -1.96, 1.96)
integrate(dnorm, -Inf, Inf)

## a slowly-convergent integral
integrand <- function(x) {1/((x+1)*sqrt(x))}
integrate(integrand, lower = 0, upper = Inf)

## don't do this if you really want the integral from 0 to Inf
integrate(integrand, lower = 0, upper = 10)
integrate(integrand, lower = 0, upper = 100000)
integrate(integrand, lower = 0, upper = 1000000, stop.on.error = FALSE)

try(integrate(function(x) 2, 0, 1)) ## no vectorizable function
integrate(function(x) rep(2, length(x)), 0, 1) ## correct

## integrate can fail if misused
integrate(dnorm,0,2)
integrate(dnorm,0,20)
integrate(dnorm,0,200)
integrate(dnorm,0,2000)
integrate(dnorm,0,20000) ## fails on many systems
integrate(dnorm,0,Inf) ## works
```

---

interaction.plot    *Two-way Interaction Plot*

---

### Description

Plots the mean (or other summary) of the response for two-way combinations of factors, thereby illustrating possible interactions.

### Usage

```
interaction.plot(x.factor, trace.factor, response, fun = mean,
               type = c("l", "p", "b"), legend = TRUE,
               trace.label = deparse(substitute(trace.factor)),
               fixed = FALSE,
               xlab = deparse(substitute(x.factor)), ylab = ylabel,
               ylim = range(cells, na.rm=TRUE),
               lty = nc:1, col = 1, pch = c(1:9, 0, letters),
               xpd = NULL, leg.bg = par("bg"), leg.bty = "n",
               xtick = FALSE, xaxt = par("xaxt"), axes = TRUE, ...)
```

### Arguments

<code>x.factor</code>	a factor whose levels will form the x axis.
<code>trace.factor</code>	another factor whose levels will form the traces.
<code>response</code>	a numeric variable giving the response
<code>fun</code>	the function to compute the summary. Should return a single real value.
<code>type</code>	the type of plot: lines or points.
<code>legend</code>	logical. Should a legend be included?
<code>trace.label</code>	overall label for the legend.
<code>fixed</code>	logical. Should the legend be in the order of the levels of <code>trace.factor</code> or in the order of the traces at their right-hand ends?
<code>xlab,ylab</code>	the x and y label of the plot each with a sensible default.
<code>ylim</code>	numeric of length 2 giving the y limits for the plot.
<code>lty</code>	line type for the lines drawn, with sensible default.
<code>col</code>	the color to be used for plotting.
<code>pch</code>	a vector of plotting symbols or characters, with sensible default.
<code>xpd</code>	determines clipping behaviour for the <code>legend</code> used, see <code>par(xpd)</code> . Per default, the legend is <i>not</i> clipped at the figure border.
<code>leg.bg, leg.bty</code>	arguments passed to <code>legend()</code> .
<code>xtick</code>	logical. Should tick marks be used on the x axis?
<code>xaxt, axes, ...</code>	graphics parameters to be passed to the plotting routines.

## Details

By default the levels of `x.factor` are plotted on the x axis in their given order, with extra space left at the right for the legend (if specified). If `x.factor` is an ordered factor and the levels are numeric, these numeric values are used for the x axis.

The response and hence its summary can contain missing values. If so, the missing values and the line segments joining them are omitted from the plot (and this can be somewhat disconcerting).

The graphics parameters `xlab`, `ylab`, `ylim`, `lty`, `col` and `pch` are given suitable defaults (and `xlim` and `xaxs` are set and cannot be overridden). The defaults are to cycle through the line types, use the foreground colour, and to use the symbols 1:9, 0, and the capital letters to plot the traces.

## Note

Some of the argument names and the precise behaviour are chosen for S-compatibility.

## References

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments*. Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## Examples

```
attach(ToothGrowth)
interaction.plot(dose, supp, len, fixed=TRUE)
dose <- ordered(dose)
interaction.plot(dose, supp, len, fixed=TRUE, col = 2:3, leg.bty = "o")
interaction.plot(dose, supp, len, fixed=TRUE, col = 2:3, type = "p")
detach()

with(OrchardSprays, {
  interaction.plot(treatment, rowpos, decrease)
  interaction.plot(rowpos, treatment, decrease, cex.axis=0.8)
  ## order the rows by their mean effect
  rowpos <- factor(rowpos, levels=sort.list(tapply(decrease, rowpos, mean)))
  interaction.plot(rowpos, treatment, decrease, col = 2:9, lty = 1)
})

with(esoph, {
  interaction.plot(agegp, alcgp, ncases/ncontrols, main = "'esoph' Data")
  interaction.plot(agegp, tobgp, ncases/ncontrols, trace.label="tobacco",
    fixed=TRUE, xaxt = "n")
})
## deal with NAs:
esoph[66,] # second to last age group: 65-74
esophNA <- esoph; esophNA$ncases[66] <- NA
with(esophNA, {
  interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5)
  # doesn't show *last* group either
  interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5, type = "b")
  ## alternative take non-NA's {"cheating"}
```

```

interaction.plot(agegp, alcgp, ncases/ncontrols, col= 2:5,
                fun = function(x) mean(x, na.rm=TRUE),
                sub = "function(x) mean(x, na.rm=TRUE)")
})
rm(esophNA) # to clear up

```

---

IQR

---

*The Interquartile Range*


---

**Description**

computes interquartile range of the  $x$  values.

**Usage**

```
IQR(x, na.rm = FALSE)
```

**Arguments**

$x$                     a numeric vector.  
 $na.rm$                 logical. Should missing values be removed?

**Details**

Note that this function computes the quartiles using the [quantile](#) function rather than following Tukey's recommendations, i.e.,  $IQR(x) = \text{quantile}(x, 3/4) - \text{quantile}(x, 1/4)$ .

For normally  $N(m, 1)$  distributed  $X$ , the expected value of  $IQR(X)$  is  $2 * \text{qnorm}(3/4) = 1.3490$ , i.e., for a normal-consistent estimate of the standard deviation, use  $IQR(x) / 1.349$ .

**References**

Tukey, J. W. (1977). *Exploratory Data Analysis*. Reading: Addison-Wesley.

**See Also**

[fivenum](#), [mad](#) which is more robust, [range](#), [quantile](#).

**Examples**

```
IQR(rivers)
```

---

is.empty.model      *Check if a Model is Empty*

---

**Description**

R model notation allows models with no intercept and no predictors. These require special handling internally. `is.empty.model()` checks whether an object describes an empty model.

**Usage**

```
is.empty.model(x)
```

**Arguments**

`x`                      A terms object or an object with a `terms` method.

**Value**

TRUE if the model is empty

**See Also**

[lm,glm](#)

**Examples**

```
y <- rnorm(20)
is.empty.model(y ~ 0)
is.empty.model(y ~ -1)
is.empty.model(lm(y ~ 0))
```

---

isoreg                      *Isotonic / Monotone Regression*

---

**Description**

Compute the isotonic (monotonely increasing nonparametric) least squares regression which is piecewise constant.

**Usage**

```
isoreg(x, y = NULL)
```

**Arguments**

`x`, `y` in `isoreg`, coordinate vectors of the regression points. Alternatively a single “plotting” structure can be specified: see `xy.coords`.

... potentially further arguments passed to methods.

**Details**

The algorithm determines the convex minorant  $m(x)$  of the *cumulative* data (i.e., `cumsum(y)`) which is piecewise linear and the result is  $m'(x)$ , a step function with level changes at locations where the convex  $m(x)$  touches the cumulative data polygon and changes slope.

`as.stepfun()` returns a `stepfun` object which can be more parsimonious.

**Value**

`isoreg()` returns an object of class `isoreg` which is basically a list with components

<code>x</code>	original (constructed) abscissa values <code>x</code> .
<code>y</code>	corresponding <code>y</code> values.
<code>yf</code>	fitted values corresponding to <i>ordered</i> <code>x</code> values.
<code>yc</code>	cumulative <code>y</code> values corresponding to <i>ordered</i> <code>x</code> values.
<code>iKnots</code>	integer vector giving indices where the fitted curve jumps, i.e., where the convex minorant has kinks.
<code>isOrd</code>	logical indicating if original <code>x</code> values were ordered increasingly already.
<code>ord</code>	<code>if(!isOrd)</code> : integer permutation <code>order(x)</code> of <i>original</i> <code>x</code> .
<code>call</code>	the <code>call</code> to <code>isoreg()</code> used.

**Note**

The code should be improved to accept *weights* additionally and solve the corresponding weighted least squares problem.

“Patches are welcome!”

**References**

Barlow, R. E., Bartholomew, D. J., Bremner, J. M., and Brunk, H. D. (1972) *Statistical inference under order restrictions*; Wiley, London.

Robertson, T., Wright, F. T. and Dykstra, R. L. (1988) *Order Restricted Statistical Inference*; Wiley, New York.

**See Also**

the plotting method `plot.isoreg` with more examples; `isoMDS()` from the **MASS** package internally uses isotonic regression.

**Examples**

```
(ir <- isoreg(c(1,0,4,3,3,5,4,2,0)))
plot(ir, plot.type = "row")

(ir3 <- isoreg(y3 <- c(1,0,4,3,3,5,4,2, 3)))# last "3", not "0"
(fi3 <- as.stepfun(ir3))
(ir4 <- isoreg(1:10, y4 <- c(5, 9, 1:2, 5:8, 3, 8)))
cat("R^2 =", formatC(sum(residuals(ir4)^2) / (9*var(y4)), dig=2),"\n")
```

KalmanLike

*Kalman Filtering***Description**

Use Kalman Filtering to find the (Gaussian) log-likelihood, or for forecasting or smoothing.

**Usage**

```
KalmanLike(y, mod, nit = 0)
KalmanRun(y, mod, nit = 0)
KalmanSmooth(y, mod, nit = 0)
KalmanForecast(n.ahead = 10, mod)
makeARIMA(phi, theta, Delta, kappa = 1e6)
```

**Arguments**

<code>y</code>	a univariate time series.
<code>mod</code>	A list describing the state-space model: see Details.
<code>nit</code>	The time at which the initialization is computed. <code>nit = 0</code> implies that the initialization is for a one-step prediction, so $P_n$ should not be computed at the first step.
<code>n.ahead</code>	The number of steps ahead for which prediction is required.
<code>phi, theta</code>	numeric vectors of length $\geq 0$ giving AR and MA parameters.
<code>Delta</code>	vector of differencing coefficients, so an ARMA model is fitted to $y[t] - \text{Delta}[1]*y[t-1] - \dots$
<code>kappa</code>	the prior variance (as a multiple of the innovations variance) for the past observations in a differenced model.

**Details**

These functions work with a general univariate state-space model with state vector  $\mathbf{a}$ , transitions  $\mathbf{a} \leftarrow \mathbf{T} \mathbf{a} + \mathbf{R} \mathbf{e}$ ,  $\mathbf{e} \sim \mathcal{N}(0, \kappa Q)$  and observation equation  $y = \mathbf{Z}' \mathbf{a} + \mathbf{e} \mathbf{t} \mathbf{a}$ , ( $\mathbf{e} \mathbf{t} \mathbf{a} \equiv \eta$ ),  $\eta \sim \mathcal{N}(0, \kappa h)$ . The likelihood is a profile likelihood after estimation of  $\kappa$ .

The model is specified as a list with at least components

**T** the transition matrix

- Z** the observation coefficients
- h** the observation variance
- V**  $RQR'$
- a** the current state estimate
- P** the current estimate of the state uncertainty matrix
- Pn** the estimate at time  $t - 1$  of the state uncertainty matrix

KalmanSmooth is the workhorse function for [tsSmooth](#).

makeARIMA constructs the state-space model for an ARIMA model.

### Value

For `KalmanLike`, a list with components `Lik` (the log-likelihood less some constants) and `s2`, the estimate of  $\kappa$ .

For `KalmanRun`, a list with components `values`, a vector of length 2 giving the output of `KalmanLike`, `resid` (the residuals) and `states`, the contemporaneous state estimates, a matrix with one row for each time.

For `KalmanSmooth`, a list with two components. Component `smooth` is a  $n$  by  $p$  matrix of state estimates based on all the observations, with one row for each time. Component `var` is a  $n$  by  $p$  by  $p$  array of variance matrices.

For `KalmanForecast`, a list with components `pred`, the predictions, and `var`, the unscaled variances of the prediction errors (to be multiplied by `s2`).

For `makeARIMA`, a model list including components for its arguments.

### Warning

These functions are designed to be called from other functions which check the validity of the arguments passed, so very little checking is done.

### References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press.

### See Also

[arima](#), [StructTS](#), [tsSmooth](#).

---

kernapply	<i>Apply Smoothing Kernel</i>
-----------	-------------------------------

---

**Description**

kernapply computes the convolution between an input sequence and a specific kernel.

**Usage**

```
kernapply(x, k, circular = FALSE, ...)  
kernapply(k1, k2)
```

**Arguments**

<code>k, k1, k2</code>	smoothing "tskernel" objects.
<code>x</code>	an input vector, matrix, or time series to be smoothed.
<code>circular</code>	a logical indicating whether the input sequence to be smoothed is treated as circular, i.e., periodic.
<code>...</code>	arguments passed to or from other methods.

**Value**

A smoothed version of the input sequence.

**Author(s)**

A. Trapletti

**See Also**

[kernel](#), [convolve](#), [filter](#), [spectrum](#)

**Examples**

```
## see 'kernel' for examples
```

---

kernel

*Smoothing Kernel Objects*


---

### Description

The "tskernel" class is designed to represent discrete symmetric normalized smoothing kernels. These kernels can be used to smooth vectors, matrices, or time series objects.

### Usage

```
kernel(coef, m, r, name)
```

```
df.kernel(k)
bandwidth.kernel(k)
is.tskernel(k)
```

### Arguments

coef	the upper half of the smoothing kernel coefficients (inclusive of coefficient zero) or the name of a kernel (currently "daniell", "dirichlet", "fejer" or "modified.daniell").
m	the kernel dimension. The number of kernel coefficients is $2*m+1$ .
name	the name of the kernel.
r	the kernel order for a Fejer kernel.
k	a "tskernel" object.

### Details

kernel is used to construct a general kernel or named specific kernels. The modified Daniell kernel halves the end coefficients (as used by S-PLUS).

df.kernel returns the "equivalent degrees of freedom" of a smoothing kernel as defined in Brockwell and Davies (1991), page 362, and bandwidth.kernel returns the equivalent bandwidth as defined in Bloomfield (1991), p. 201, with a continuity correction.

### Value

kernel returns a list with class "tskernel", and components the coefficients coef and the kernel dimension m. An additional attribute is "name".

### Author(s)

A. Trapletti; modifications by B.D. Ripley

## References

- Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction*. Wiley.
- Brockwell, P.J. and Davis, R.A. (1991) *Time Series: Theory and Methods*. Second edition. Springer, pp. 350–365.

## See Also

[kernapply](#)

## Examples

```
# Demonstrate a simple trading strategy for the
# financial time series German stock index DAX.
x <- EuStockMarkets[,1]
k1 <- kernel("daniell", 50) # a long moving average
k2 <- kernel("daniell", 10) # and a short one
plot(k1)
plot(k2)
x1 <- kernapply(x, k1)
x2 <- kernapply(x, k2)
plot(x)
lines(x1, col = "red") # go long if the short crosses the long upwards
lines(x2, col = "green") # and go short otherwise

# Reproduce example 10.4.3 from Brockwell and Davies (1991)
spectrum(sunspot.year, kernel=kernel("daniell", c(11,7,3)), log="no")
```

---

kmeans

*K-Means Clustering*

---

## Description

Perform k-means clustering on a data matrix.

## Usage

```
kmeans(x, centers, iter.max = 10)
```

## Arguments

- |                       |  |
|-----------------------|--|
| <code>x</code>        | A numeric matrix of data, or an object that can be coerced to such a matrix (such as a numeric vector or a data frame with all numeric columns).           |
| <code>centers</code>  | Either the number of clusters or a set of initial cluster centers. If the first, a random set of rows in <code>x</code> are chosen as the initial centers. |
| <code>iter.max</code> | The maximum number of iterations allowed.  |

**Details**

The data given by `x` is clustered by the k-means algorithm. When this terminates, all cluster centres are at the mean of their Voronoi sets (the set of data points which are nearest to the cluster centre).

The algorithm of Hartigan and Wong (1979) is used.

**Value**

A list with components:

<code>cluster</code>	A vector of integers indicating the cluster to which each point is allocated.
<code>centers</code>	A matrix of cluster centres.
<code>withinss</code>	The within-cluster sum of squares for each cluster.
<code>size</code>	The number of points in each cluster.

**References**

Hartigan, J.A. and Wong, M.A. (1979). A K-means clustering algorithm. *Applied Statistics* **28**, 100–108.

**Examples**

```
# a 2-dimensional example
x <- rbind(matrix(rnorm(100, sd = 0.3), ncol = 2),
           matrix(rnorm(100, mean = 1, sd = 0.3), ncol = 2))
cl <- kmeans(x, 2, 20)
plot(x, col = cl$cluster)
points(cl$centers, col = 1:2, pch = 8)
```

---

<code>kruskal.test</code>	<i>Kruskal-Wallis Rank Sum Test</i>
---------------------------	-------------------------------------

---

**Description**

Performs a Kruskal-Wallis rank sum test.

**Usage**

```
kruskal.test(x, ...)

## Default S3 method:
kruskal.test(x, g, ...)

## S3 method for class 'formula':
kruskal.test(formula, data, subset, na.action, ...)
```

**Arguments**

<code>x</code>	a numeric vector of data values, or a list of numeric data vectors.
<code>g</code>	a vector or factor object giving the group for the corresponding elements of <code>x</code> . Ignored if <code>x</code> is a list.
<code>formula</code>	a formula of the form <code>lhs ~ rhs</code> where <code>lhs</code> gives the data values and <code>rhs</code> the corresponding groups.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

**Details**

`kruskal.test` performs a Kruskal-Wallis rank sum test of the null that the location parameters of the distribution of `x` are the same in each group (sample). The alternative is that they differ in at least one.

If `x` is a list, its elements are taken as the samples to be compared, and hence have to be numeric data vectors. In this case, `g` is ignored, and one can simply use `kruskal.test(x)` to perform the test. If the samples are not yet contained in a list, use `kruskal.test(list(x, ...))`.

Otherwise, `x` must be a numeric data vector, and `g` must be a vector or factor object of the same length as `x` giving the group for the corresponding elements of `x`.

**Value**

A list with class "htest" containing the following components:

<code>statistic</code>	the Kruskal-Wallis rank sum statistic.
<code>parameter</code>	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>method</code>	the character string "Kruskal-Wallis rank sum test".
<code>data.name</code>	a character string giving the names of the data.

**References**

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 115–120.

**See Also**

The Wilcoxon rank sum test ([wilcox.test](#)) as the special case for two samples; [lm](#) together with [anova](#) for performing one-way location analysis under normality assumptions; with Student's t test ([t.test](#)) as the special case for two samples.

**Examples**

```
## Hollander & Wolfe (1973), 116.
## Mucociliary efficiency from the rate of removal of dust in normal
## subjects, subjects with obstructive airway disease, and subjects
## with asbestosis.
x <- c(2.9, 3.0, 2.5, 2.6, 3.2) # normal subjects
y <- c(3.8, 2.7, 4.0, 2.4)     # with obstructive airway disease
z <- c(2.8, 3.4, 3.7, 2.2, 2.0) # with asbestosis
kruskal.test(list(x, y, z))
## Equivalently,
x <- c(x, y, z)
g <- factor(rep(1:3, c(5, 4, 5)),
            labels = c("Normal subjects",
                      "Subjects with obstructive airway disease",
                      "Subjects with asbestosis"))
kruskal.test(x, g)

## Formula interface.
boxplot(Ozone ~ Month, data = airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

ks.test

*Kolmogorov-Smirnov Tests***Description**

Performs one or two sample Kolmogorov-Smirnov tests.

**Usage**

```
ks.test(x, y, ..., alternative = c("two.sided", "less", "greater"),
        exact = NULL)
```

**Arguments**

<code>x</code>	a numeric vector of data values.
<code>y</code>	either a numeric vector of data values, or a character string naming a distribution function.
<code>...</code>	parameters of the distribution specified (as a character string) by <code>y</code> .
<code>alternative</code>	indicates the alternative hypothesis and must be one of "two.sided" (default), "less", or "greater". You can specify just the initial letter.
<code>exact</code>	NULL or a logical indicating whether an exact p-value should be computed. See Details for the meaning of NULL. Only used in the two-sided two-sample case.

## Details

If `y` is numeric, a two-sample test of the null hypothesis that `x` and `y` were drawn from the same *continuous* distribution is performed.

Alternatively, `y` can be a character string naming a continuous distribution function. In this case, a one-sample test is carried out of the null that the distribution function which generated `x` is distribution `y` with parameters specified by `...`

The presence of ties generates a warning, since continuous distributions do not generate them.

The possible values `"two.sided"`, `"less"` and `"greater"` of `alternative` specify the null hypothesis that the true distribution function of `x` is equal to, not less than or not greater than the hypothesized distribution function (one-sample case) or the distribution function of `y` (two-sample case), respectively.

Exact p-values are only available for the two-sided two-sample test with no ties. In that case, if `exact = NULL` (the default) an exact p-value is computed if the product of the sample sizes is less than 10000. Otherwise, asymptotic distributions are used whose approximations may be inaccurate in small samples.

If a single-sample test is used, the parameters specified in `...` must be pre-specified and not estimated from the data. There is some more refined distribution theory for the KS test with estimated parameters (see Durbin, 1973), but that is not implemented in `ks.test`.

## Value

A list with class `"htest"` containing the following components:

<code>statistic</code>	the value of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>alternative</code>	a character string describing the alternative hypothesis.
<code>method</code>	a character string indicating what type of test was performed.
<code>data.name</code>	a character string giving the name(s) of the data.

## References

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 295–301 (one-sample “Kolmogorov” test), 309–314 (two-sample “Smirnov” test).

Durbin, J. (1973) *Distribution theory for tests based on the sample distribution function*. SIAM.

## See Also

[shapiro.test](#) which performs the Shapiro-Wilk test for normality.

## Examples

```
x <- rnorm(50)
y <- runif(30)
# Do x and y come from the same distribution?
ks.test(x, y)
# Does x come from a shifted gamma distribution with shape 3 and scale 2?
ks.test(x+2, "pgamma", 3, 2) # two-sided
ks.test(x+2, "pgamma", 3, 2, alternative = "gr")
```

ksmooth

*Kernel Regression Smoother***Description**

The Nadaraya-Watson kernel regression estimate.

**Usage**

```
ksmooth(x, y, kernel = c("box", "normal"), bandwidth = 0.5,
        range.x = range(x), n.points = max(100, length(x)), x.points)
```

**Arguments**

<code>x</code>	input x values
<code>y</code>	input y values
<code>kernel</code>	the kernel to be used.
<code>bandwidth</code>	the bandwidth. The kernels are scaled so that their quartiles (viewed as probability densities) are at $\pm 0.25 \cdot \text{bandwidth}$ .
<code>range.x</code>	the range of points to be covered in the output.
<code>n.points</code>	the number of points at which to evaluate the fit.
<code>x.points</code>	points at which to evaluate the smoothed fit. If missing, <code>n.points</code> are chosen uniformly to cover <code>range.x</code> .

**Value**

A list with components

<code>x</code>	values at which the smoothed fit is evaluated. Guaranteed to be in increasing order.
<code>y</code>	fitted values corresponding to <code>x</code> .

**Note**

This function is implemented purely for compatibility with S, although it is nowhere near as slow as the S function. Better kernel smoothers are available in other packages.

**Examples**

```
with(cars, {
  plot(speed, dist)
  lines(ksmooth(speed, dist, "normal", bandwidth=2), col=2)
  lines(ksmooth(speed, dist, "normal", bandwidth=5), col=3)
})
```

---

`lag`*Lag a Time Series*

---

**Description**

Compute a lagged version of a time series, shifting the time base back by a given number of observations.

**Usage**

```
lag(x, ...)  
  
## Default S3 method:  
lag(x, k = 1, ...)
```

**Arguments**

<code>x</code>	A vector or matrix or univariate or multivariate time series
<code>k</code>	The number of lags (in units of observations).
<code>...</code>	further arguments to be passed to or from methods.

**Details**

Vector or matrix arguments `x` are coerced to time series.

`lag` is a generic function; this page documents its default method.

**Value**

A time series object.

**Note**

Note the sign of `k`: a series lagged by a positive `k` starts *earlier*.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[diff](#), [deltat](#)

**Examples**

```
lag(ldeaths, 12) # starts one year earlier
```

lag.plot

*Time Series Lag Plots***Description**

Plot time series against lagged versions of themselves. Helps visualizing “auto-dependence” even when auto-correlations vanish.

**Usage**

```
lag.plot(x, lags = 1, layout = NULL, set.lags = 1:lags,
        main = NULL, asp = 1,
        font.main=par("font.main"), cex.main=par("cex.main"),
        diag = TRUE, diag.col = "gray", type = "p", oma = NULL,
        ask = NULL, do.lines = (n <= 150), labels = do.lines, ...)
```

**Arguments**

x	time-series (univariate or multivariate)
lags	number of lag plots desired, see arg <code>set.lags</code> .
layout	the layout of multiple plots, basically the <code>mfrac</code> <code>par()</code> argument. The default uses about a square layout (see <code>n2mfrow</code> such that all plots are on one page.
set.lags	positive integer vector allowing to specify the set of lags used; defaults to <code>1:lags</code> .
main	character with a main header title to be done on the top of each page.
asp	Aspect ratio to be fixed, see <code>plot.default</code> .
font.main, cex.main	attributes for the title, see <code>par()</code> .
diag	logical indicating if the x=y diagonal should be drawn.
diag.col	color to be used for the diagonal <code>if(diag)</code> .
type	plot type to be used, but see <code>plot.ts</code> about its restricted meaning.
oma	outer margins, see <code>par</code> .
ask	logical; if true, the user is asked before a new page is started.
do.lines	logical indicating if lines should be drawn.
labels	logical indicating if labels should be used.
...	Further arguments to <code>plot.ts</code> .

**Note**

It is more flexible and has different default behaviour than the S version. We use `main =` instead of `head =` for internal consistency.

**Author(s)**

Martin Maechler

**See Also**

[plot.ts](#) which is the basic work horse.

**Examples**

```
lag.plot(nhtemp, 8, diag.col = "forest green")
lag.plot(nhtemp, 5, main="Average Temperatures in New Haven")
## ask defaults to TRUE when we have more than one page:
lag.plot(nhtemp, 6, layout = c(2,1), asp = NA,
         main = "New Haven Temperatures", col.main = "blue")

## Multivariate (but non-stationary! ...)
lag.plot(freeny.x, lag = 3)
## Not run:
no lines for long series :
lag.plot(sqrt(sunspots), set = c(1:4, 9:12), pch = ".", col = "gold")
## End(Not run)
```

---

line

*Robust Line Fitting*

---

**Description**

Fit a line robustly as recommended in *Exploratory Data Analysis*.

**Usage**

```
line(x, y)
```

**Arguments**

*x, y* the arguments can be any way of specifying x-y pairs.

**Value**

An object of class "tukeyline".

Methods are available for the generic functions `coef`, `residuals`, `fitted`, and `print`.

**References**

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

**See Also**

[lm](#).

**Examples**

```
plot(cars)
(z <- line(cars))
abline(coef(z))
## Tukey-Anscombe Plot :
plot(residuals(z) ~ fitted(z), main = deparse(z$call))
```

lm

*Fitting Linear Models***Description**

lm is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although `av` may provide a more convenient interface for these).

**Usage**

```
lm(formula, data, subset, weights, na.action,
    method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
    singular.ok = TRUE, contrasts = NULL, offset, ...)
```

**Arguments**

formula	a symbolic description of the model to be fit. The details of model specification are given below.
data	an optional data frame containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which lm is called.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional vector of weights to be used in the fitting process. If specified, weighted least squares is used with weights weights (that is, minimizing $\sum(w \cdot e^2)$ ); otherwise ordinary least squares is used.
na.action	a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The “factory-fresh” default is na.omit. Another possible value is NULL, no action.
method	the method to be used; for fitting, currently only method = "qr" is supported; method = "model.frame" returns the model frame (the same as with model = TRUE, see below).
model, x, y, qr	logicals. If TRUE the corresponding components of the fit (the model frame, the model matrix, the response, the QR decomposition) are returned.
singular.ok	logical. If FALSE (the default in S but not in R) a singular fit is an error.

<code>contrasts</code>	an optional list. See the <code>contrasts.arg</code> of <code>model.matrix.default</code> .
<code>offset</code>	this can be used to specify an <i>a priori</i> known component to be included in the linear predictor during fitting. An <code>offset</code> term can be included in the formula instead or as well, and if both are specified their sum is used.
<code>...</code>	additional arguments to be passed to the low level regression fitting functions (see below).

## Details

Models for `lm` are specified symbolically. A typical model has the form `response ~ terms` where `response` is the (numeric) response vector and `terms` is a series of terms which specifies a linear predictor for `response`. A terms specification of the form `first + second` indicates all the terms in `first` together with all the terms in `second` with duplicates removed. A specification of the form `first:second` indicates the set of terms obtained by taking the interactions of all terms in `first` with all terms in `second`. The specification `first*second` indicates the *cross* of `first` and `second`. This is the same as `first + second + first:second`. If `response` is a matrix a linear model is fitted to each column of the matrix. See `model.matrix` for some further details. The terms in the formula will be re-ordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on: to avoid this pass a `terms` object as the formula.

A formula has an implied intercept term. To remove this use either `y ~ x - 1` or `y ~ 0 + x`. See `formula` for more details of allowed formulae.

`lm` calls the lower level functions `lm.fit`, etc, see below, for the actual numerical computations. For programming only, you may consider doing likewise.

All of `weights`, `subset` and `offset` are evaluated in the same way as variables in formula, that is first in data and then in the environment of formula.

## Value

`lm` returns an object of class `"lm"` or for multiple responses of class `c("mlm", "lm")`.

The functions `summary` and `anova` are used to obtain and print a summary and analysis of variance table of the results. The generic accessor functions `coefficients`, `effects`, `fitted.values` and `residuals` extract various useful features of the value returned by `lm`.

An object of class `"lm"` is a list containing at least the following components:

<code>coefficients</code>	a named vector of coefficients
<code>residuals</code>	the residuals, that is response minus fitted values.
<code>fitted.values</code>	the fitted mean values.
<code>rank</code>	the numeric rank of the fitted linear model.
<code>weights</code>	(only for weighted fits) the specified weights.
<code>df.residual</code>	the residual degrees of freedom.
<code>call</code>	the matched call.
<code>terms</code>	the <code>terms</code> object used.
<code>contrasts</code>	(only where relevant) the contrasts used.

<code>xlevels</code>	(only where relevant) a record of the levels of the factors used in fitting.
<code>y</code>	if requested, the response used.
<code>x</code>	if requested, the model matrix used.
<code>model</code>	if requested (the default), the model frame used.

In addition, non-null fits will have components `assign`, `effects` and (unless not requested) `qr` relating to the linear fit, for use by extractor functions such as `summary` and `effects`.

### Using time series

Considerable care is needed when using `lm` with time series.

Unless `na.action = NULL`, the time series attributes are stripped from the variables before the regression is done. (This is necessary as omitting NAs would invalidate the time series attributes, and if NAs are omitted in the middle of the series the result would no longer be a regular time series.)

Even if the time series attributes are retained, they are not used to line up series, so that the time shift of a lagged or differenced regressor would be ignored. It is good practice to prepare a data argument by `ts.intersect(..., dframe = TRUE)`, then apply a suitable `na.action` to that data frame and call `lm` with `na.action = NULL` so that residuals and fitted values are time series.

### Note

Offsets specified by `offset` will not be included in predictions by `predict.lm`, whereas those specified by an offset term in the formula will be.

### Author(s)

The design was inspired by the S function of the same name described in Chambers (1992). The implementation of model formula by Ross Ihaka was based on Wilkinson & Rogers (1973).

### References

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Wilkinson, G. N. and Rogers, C. E. (1973) Symbolic descriptions of factorial models for analysis of variance. *Applied Statistics*, **22**, 392–9.

### See Also

`summary.lm` for summaries and `anova.lm` for the ANOVA table; `aov` for a different interface.

The generic functions `coef`, `effects`, `residuals`, `fitted`, `vcov`.

`predict.lm` (via `predict`) for prediction, including confidence and prediction intervals.

`lm.influence` for regression diagnostics, and `glm` for **generalized** linear models.

The underlying low level functions, `lm.fit` for plain, and `lm.wfit` for weighted regression fitting.

**Examples**

```
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2,10,20, labels=c("Ctl","Trt"))
weight <- c(ctl, trt)
anova(lm.D9 <- lm(weight ~ group))
summary(lm.D90 <- lm(weight ~ group - 1))# omitting intercept
summary(resid(lm.D9) - resid(lm.D90)) #- residuals almost identical

opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(lm.D9, las = 1)      # Residuals, Fitted, ...
par(opar)

## model frame :
stopifnot(identical(lm(weight ~ group, method = "model.frame"),
                    model.frame(lm.D9)))
```

lm.fit

*Fitter Functions for Linear Models***Description**

These are the basic computing engines called by `lm` used to fit linear models. These should usually *not* be used directly unless by experienced users.

**Usage**

```
lm.fit(x, y, offset = NULL, method = "qr", tol = 1e-7,
       singular.ok = TRUE, ...)
```

```
lm.wfit(x, y, w, offset = NULL, method = "qr", tol = 1e-7,
        singular.ok = TRUE, ...)
```

**Arguments**

<code>x</code>	design matrix of dimension $n * p$ .
<code>y</code>	vector of observations of length $n$ .
<code>w</code>	vector of weights (length $n$ ) to be used in the fitting process for the <code>wfit</code> functions. Weighted least squares is used with weights $w$ , i.e., $\sum(w * e^2)$ is minimized.
<code>offset</code>	numeric of length $n$ ). This can be used to specify an <i>a priori</i> known component to be included in the linear predictor during fitting.
<code>method</code>	currently, only <code>method="qr"</code> is supported.
<code>tol</code>	tolerance for the <code>qr</code> decomposition. Default is $1e-7$ .
<code>singular.ok</code>	logical. If <code>FALSE</code> , a singular model is an error.
<code>...</code>	currently disregarded.

**Details**

The functions `lm.{w}fit.null` are called by `lm.fit` or `lm.wfit` respectively, when `x` has zero columns.

**Value**

a list with components

`coefficients` `p` vector

`residuals` `n` vector

`fitted.values`  
`n` vector

`effects` (not null fits) `n` vector of orthogonal single-df effects. The first `rank` of them correspond to non-aliased coefficients, and are named accordingly.

`weights` `n` vector — *only* for the `*wfit*` functions.

`rank` integer, giving the rank

`df.residual` degrees of freedom of residuals

`qr` (not null fits) the QR decomposition, see [qr](#).

**See Also**

[lm](#) which you should use for linear least squares regression, unless you know better.

**Examples**

```
set.seed(129)
n <- 7 ; p <- 2
X <- matrix(rnorm(n * p), n,p) # no intercept!
y <- rnorm(n)
w <- rnorm(n)^2

str(lmw <- lm.wfit(x=X, y=y, w=w))

str(lm. <- lm.fit (x=X, y=y))
```

---

lm.influence

*Regression Diagnostics*

---

**Description**

This function provides the basic quantities which are used in forming a wide variety of diagnostics for checking the quality of regression fits.

**Usage**

```
influence(model, ...)
## S3 method for class 'lm':
influence(model, do.coef = TRUE, ...)
## S3 method for class 'glm':
influence(model, do.coef = TRUE, ...)

lm.influence(model, do.coef = TRUE)
```

**Arguments**

model	an object as returned by <code>lm</code> .
do.coef	logical indicating if the changed coefficients (see below) are desired. These need $O(n^2p)$ computing time.
...	further arguments passed to or from other methods.

**Details**

The `influence.measures()` and other functions listed in **See Also** provide a more user oriented way of computing a variety of regression diagnostics. These all build on `lm.influence`.

An attempt is made to ensure that computed hat values that are probably one are treated as one, and the corresponding rows in `sigma` and `coefficients` are NaN. (Dropping such a case would normally result in a variable being dropped, so it is not possible to give simple drop-one diagnostics.)

**Value**

A list containing the following components of the same length or number of rows  $n$ , which is the number of non-zero weights. Cases omitted in the fit are omitted unless a `na.action` method was used (such as `na.exclude`) which restores them.

hat	a vector containing the diagonal of the “hat” matrix.
coefficients	(unless <code>do.coef</code> is false) a matrix whose $i$ -th row contains the change in the estimated coefficients which results when the $i$ -th case is dropped from the regression. Note that aliased coefficients are not included in the matrix.
sigma	a vector whose $i$ -th element contains the estimate of the residual standard deviation obtained when the $i$ -th case is dropped from the regression.
wt.res	a vector of <i>weighted</i> (or for class <code>glm</code> rather <i>deviance</i> ) residuals.

**Note**

The `coefficients` returned by the R version of `lm.influence` differ from those computed by S. Rather than returning the coefficients which result from dropping each case, we return the changes in the coefficients. This is more directly useful in many diagnostic measures.

Since these need  $O(n^2p)$  computing time, they can be omitted by `do.coef = FALSE`.

Note that cases with `weights == 0` are *dropped* (contrary to the situation in S).

If a model has been fitted with `na.action=na.exclude` (see `na.exclude`), cases excluded in the fit *are* considered here.

**References**

See the list in the documentation for [influence.measures](#).

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[summary.lm](#) for [summary](#) and related methods;  
[influence.measures](#),  
[hat](#) for the hat matrix diagonals,  
[dfbetas](#), [dffits](#), [covratio](#), [cooks.distance](#), [lm](#).

**Examples**

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
summary(lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi,
                   data = LifeCycleSavings),
        corr = TRUE)
str(lmI <- lm.influence(lm.SR))

## For more "user level" examples, use example(influence.measures)
```

---

lm.summaries      *Accessing Linear Model Fits*

---

**Description**

All these functions are [methods](#) for class "lm" objects.

**Usage**

```
## S3 method for class 'lm':
family(object, ...)

## S3 method for class 'lm':
formula(x, ...)

## S3 method for class 'lm':
residuals(object,
           type = c("working", "response", "deviance", "pearson",
                  "partial"),
           ...)

weights(object, ...)
```

**Arguments**

`object`, `x`     an object inheriting from class `lm`, usually the result of a call to `lm` or `aov`.  
`...`             further arguments passed to or from other methods.  
`type`             the type of residuals which should be returned.

**Details**

The generic accessor functions `coef`, `effects`, `fitted` and `residuals` can be used to extract various useful features of the value returned by `lm`.

The working and response residuals are “observed - fitted”. The deviance and pearson residuals are weighted residuals, scaled by the square root of the weights used in fitting. The partial residuals are a matrix with each column formed by omitting a term from the model. In all these, zero weight cases are never omitted (as opposed to the standardized `rstudent` residuals).

**References**

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

The model fitting function `lm`, `anova.lm`.

`coef`, `deviance`, `df.residual`, `effects`, `fitted`, `glm` for **generalized** linear models, `influence` (etc on that page) for regression diagnostics, `weighted.residuals`, `residuals`, `residuals.glm`, `summary.lm`.

**Examples**

```
##-- Continuing the lm(.) example:
coef(lm.D90)# the bare coefficients

## The 2 basic regression diagnostic plots [plot.lm(.) is preferred]
plot(resid(lm.D90), fitted(lm.D90))# Tukey-Anscombe's
abline(h=0, lty=2, col = 'gray')

qqnorm(residuals(lm.D90))
```

---

loadings

---

*Print Loadings in Factor Analysis*


---

**Description**

Extract or print loadings in factor analysis (or principal components analysis).

**Usage**

```
loadings(x)

## S3 method for class 'loadings':
print(x, digits = 3, cutoff = 0.1, sort = FALSE, ...)

## S3 method for class 'factanal':
print(x, digits = 3, ...)
```

**Arguments**

<code>x</code>	an object of class "factanal" or "princomp" or the loadings component of such an object.
<code>digits</code>	number of decimal places to use in printing uniquenesses and loadings.
<code>cutoff</code>	loadings smaller than this (in absolute value) are suppressed.
<code>sort</code>	logical. If true, the variables are sorted by their importance on each factor. Each variable with any loading larger than 0.5 (in modulus) is assigned to the factor with the largest loading, and the variables are printed in the order of the factor they are assigned to, then those unassigned.
<code>...</code>	further arguments for other methods, such as <code>cutoff</code> and <code>sort</code> for <code>print.factanal</code> .

**See Also**

[factanal](#), [princomp](#)

---

loess

*Local Polynomial Regression Fitting*

---

**Description**

Fit a polynomial surface determined by one or more numerical predictors, using local fitting.

**Usage**

```
loess(formula, data, weights, subset, na.action, model = FALSE,
      span = 0.75, enp.target, degree = 2,
      parametric = FALSE, drop.square = FALSE, normalize = TRUE,
      family = c("gaussian", "symmetric"),
      method = c("loess", "model.frame"),
      control = loess.control(...), ...)
```

**Arguments**

<code>formula</code>	a formula specifying the numeric response and one to four numeric predictors (best specified via an interaction, but can also be specified additively).
<code>data</code>	an optional data frame within which to look first for the response, predictors and weights.
<code>weights</code>	optional weights for each case.
<code>subset</code>	an optional specification of a subset of the data to be used.
<code>na.action</code>	the action to be taken with missing values in the response or predictors. The default is given by <code>getOption("na.action")</code> .
<code>model</code>	should the model frame be returned?
<code>span</code>	the parameter $\alpha$ which controls the degree of smoothing.
<code>enp.target</code>	an alternative way to specify <code>span</code> , as the approximate equivalent number of parameters to be used.
<code>degree</code>	the degree of the polynomials to be used, up to 2.
<code>parametric</code>	should any terms be fitted globally rather than locally? Terms can be specified by name, number or as a logical vector of the same length as the number of predictors.
<code>drop.square</code>	for fits with more than one predictor and <code>degree=2</code> , should the quadratic term (and cross-terms) be dropped for particular predictors? Terms are specified in the same way as for <code>parametric</code> .
<code>normalize</code>	should the predictors be normalized to a common scale if there is more than one? The normalization used is to set the 10% trimmed standard deviation to one. Set to false for spatial coordinate predictors and others known to be a common scale.
<code>family</code>	if "gaussian" fitting is by least-squares, and if "symmetric" a re-descending M estimator is used with Tukey's biweight function.
<code>method</code>	fit the model or just extract the model frame.
<code>control</code>	control parameters: see <a href="#">loess.control</a> .
<code>...</code>	control parameters can also be supplied directly.

**Details**

Fitting is done locally. That is, for the fit at point  $x$ , the fit is made using points in a neighbourhood of  $x$ , weighted by their distance from  $x$  (with differences in 'parametric' variables being ignored when computing the distance). The size of the neighbourhood is controlled by  $\alpha$  (set by `span` or `enp.target`). For  $\alpha < 1$ , the neighbourhood includes proportion  $\alpha$  of the points, and these have tricubic weighting (proportional to  $(1 - (\text{dist}/\text{maxdist})^3)^3$ ). For  $\alpha > 1$ , all points are used, with the 'maximum distance' assumed to be  $\alpha^{1/p}$  times the actual maximum distance for  $p$  explanatory variables.

For the default family, fitting is by (weighted) least squares. For `family="symmetric"` a few iterations of an M-estimation procedure with Tukey's biweight are used. Be aware that as the initial value is the least-squares fit, this need not be a very resistant fit.

It can be important to tune the control list to achieve acceptable speed. See [loess.control](#) for details.

**Value**

An object of class "loess".

**Note**

As this is based on the `cloess` package available at `netlib`, it is similar to but not identical to the `loess` function of `S`. In particular, conditioning is not implemented.

The memory usage of this implementation of `loess` is roughly quadratic in the number of points, with 1000 points taking about 10Mb.

**Author(s)**

B.D. Ripley, based on the `cloess` package of Cleveland, Grosse and Shyu available at <http://www.netlib.org/a/>.

**References**

W.S. Cleveland, E. Grosse and W.M. Shyu (1992) Local regression models. Chapter 8 of *Statistical Models in S* eds J.M. Chambers and T.J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[loess.control](#), [predict.loess](#).  
[lowess](#), the ancestor of `loess` (with different defaults!).

**Examples**

```
cars.lo <- loess(dist ~ speed, cars)
predict(cars.lo, data.frame(speed = seq(5, 30, 1)), se = TRUE)
# to allow extrapolation
cars.lo2 <- loess(dist ~ speed, cars,
  control = loess.control(surface = "direct"))
predict(cars.lo2, data.frame(speed = seq(5, 30, 1)), se = TRUE)
```

---

<code>loess.control</code>	<i>Set Parameters for Loess</i>
----------------------------	---------------------------------

---

**Description**

Set control parameters for `loess` fits.

**Usage**

```
loess.control(surface = c("interpolate", "direct"),
  statistics = c("approximate", "exact"),
  trace.hat = c("exact", "approximate"),
  cell = 0.2, iterations = 4, ...)
```

**Arguments**

<code>surface</code>	should the fitted surface be computed exactly or via interpolation from a kd tree?
<code>statistics</code>	should the statistics be computed exactly or approximately? Exact computation can be very slow.
<code>trace.hat</code>	should the trace of the smoother matrix be computed exactly or approximately? It is recommended to use the approximation for more than about 1000 data points.
<code>cell</code>	if interpolation is used this controls the accuracy of the approximation via the maximum number of points in a cell in the kd tree. Cells with more than <code>floor(n*span*cell)</code> points are subdivided.
<code>iterations</code>	the number of iterations used in robust fitting.
<code>...</code>	further arguments which are ignored.

**Value**

A list with components

```
surface
statistics
trace.hat
cell
iterations
```

with meanings as explained under 'Arguments'.

**See Also**

[loess](#)

---

Logistic

*The Logistic Distribution*

---

**Description**

Density, distribution function, quantile function and random generation for the logistic distribution with parameters `location` and `scale`.

**Usage**

```
dlogis(x, location = 0, scale = 1, log = FALSE)
plogis(q, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
qlogis(p, location = 0, scale = 1, lower.tail = TRUE, log.p = FALSE)
rlogis(n, location = 0, scale = 1)
```

**Arguments**

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>location, scale</code>	location and scale parameters.
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

If `location` or `scale` are omitted, they assume the default values of 0 and 1 respectively.

The Logistic distribution with `location` =  $\mu$  and `scale` =  $\sigma$  has distribution function

$$F(x) = \frac{1}{1 + e^{-(x-\mu)/\sigma}}$$

and density

$$f(x) = \frac{1}{\sigma} \frac{e^{(x-\mu)/\sigma}}{(1 + e^{(x-\mu)/\sigma})^2}$$

It is a long-tailed distribution with mean  $\mu$  and variance  $\pi^2/3\sigma^2$ .

**Value**

`dlogis` gives the density, `plogis` gives the distribution function, `qlogis` gives the quantile function, and `rlogis` generates random deviates.

**Note**

`qlogis(p)` is the same as the well known ‘*logit*’ function,  $\text{logit}(p) = \log(p/(1-p))$ , and `plogis(x)` has consequently been called the “inverse logit”.

The distribution function is a rescaled hyperbolic tangent, `plogis(x) == (1 + tanh\(x/2\)) / 2`, and it is called *sigmoid function* in contexts such as neural networks.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**Examples**

```
var(rlogis(4000, 0, s = 5))# approximately (+/- 3)
pi^2/3 * 5^2
```

---

logLik *Extract Log-Likelihood*

---

## Description

This function is generic; method functions can be written to handle specific classes of objects. Classes which already have methods for this function include: `glm`, `lm`, `nls` and `gls`, `lme` and others in package **nlme**.

## Usage

```
logLik(object, ...)  
  
## S3 method for class 'lm':  
logLik(object, REML = FALSE, ...)
```

## Arguments

<code>object</code>	any object from which a log-likelihood value, or a contribution to a log-likelihood value, can be extracted.
<code>...</code>	some methods for this generic function require additional arguments.
<code>REML</code>	an optional logical value. If <code>TRUE</code> the restricted log-likelihood is returned, else, if <code>FALSE</code> , the log-likelihood is returned. Defaults to <code>FALSE</code> .

## Details

For a `glm` fit the `family` does not have to specify how to calculate the log-likelihood, so this is based on the family's function to compute the AIC. For `gaussian`, `Gamma` and `inverse.gaussian` families it is assumed that the dispersion of the GLM is estimated and has been included in the AIC, and for all other families it is assumed that the dispersion is known.

Note that this procedure is not completely accurate for the gamma and inverse gaussian families, as the estimate of dispersion used is not the MLE.

## Value

Returns an object, say `r`, of class `logLik` which is a number with attributes, `attr(r, "df")` (**d**egrees of **f**reedom) giving the number of parameters in the model. There's a simple `print` method for `logLik` objects.

The details depend on the method function used; see the appropriate documentation.

## Author(s)

Jose Pinheiro and Douglas Bates

## References

For `logLik.lm`:

Harville, D.A. (1974). Bayesian inference for variance components using only error contrasts. *Biometrika*, **61**, 383–385.

## See Also

[logLik.gls](#), [logLik.lme](#), in package **nlme**, etc.

## Examples

```
x <- 1:5
lmx <- lm(x ~ 1)
logLik(lmx) # using print.logLik() method
str(logLik(lmx))

## lm method
(fm1 <- lm(rating ~ ., data = attitude))
logLik(fm1)
logLik(fm1, REML = TRUE)

res <- try(data(Orthodont, package="nlme"))
if(!inherits(res, "try-error")) {
  fm1 <- lm(distance ~ Sex * age, Orthodont)
  print(logLik(fm1))
  print(logLik(fm1, REML = TRUE))
}
```

---

loglin

*Fitting Log-Linear Models*

---

## Description

`loglin` is used to fit log-linear models to multidimensional contingency tables by Iterative Proportional Fitting.

## Usage

```
loglin(table, margin, start = rep(1, length(table)), fit = FALSE,
       eps = 0.1, iter = 20, param = FALSE, print = TRUE)
```

## Arguments

`table` a contingency table to be fit, typically the output from `table`.

`margin` a list of vectors with the marginal totals to be fit.  
(Hierarchical) log-linear models can be specified in terms of these marginal totals which give the “maximal” factor subsets contained in the model. For example, in a three-factor model, `list(c(1, 2), c(1, 3))` specifies a model

which contains parameters for the grand mean, each factor, and the 1-2 and 1-3 interactions, respectively (but no 2-3 or 1-2-3 interaction), i.e., a model where factors 2 and 3 are independent conditional on factor 1 (sometimes represented as '[12][13]').

The names of factors (i.e., `names(dimnames(table))`) may be used rather than numeric indices.

<code>start</code>	a starting estimate for the fitted table. This optional argument is important for incomplete tables with structural zeros in <code>table</code> which should be preserved in the fit. In this case, the corresponding entries in <code>start</code> should be zero and the others can be taken as one.
<code>fit</code>	a logical indicating whether the fitted values should be returned.
<code>eps</code>	maximum deviation allowed between observed and fitted margins.
<code>iter</code>	maximum number of iterations.
<code>param</code>	a logical indicating whether the parameter values should be returned.
<code>print</code>	a logical. If <code>TRUE</code> , the number of iterations and the final deviation are printed.

### Details

The Iterative Proportional Fitting algorithm as presented in Haberman (1972) is used for fitting the model. At most `iter` iterations are performed, convergence is taken to occur when the maximum deviation between observed and fitted margins is less than `eps`. All internal computations are done in double precision; there is no limit on the number of factors (the dimension of the table) in the model.

Assuming that there are no structural zeros, both the Likelihood Ratio Test and Pearson test statistics have an asymptotic chi-squared distribution with `df` degrees of freedom.

Package **MASS** contains `loglm`, a front-end to `loglin` which allows the log-linear model to be specified and fitted in a formula-based manner similar to that of other fitting functions such as `lm` or `glm`.

### Value

A list with the following components.

<code>lrt</code>	the Likelihood Ratio Test statistic.
<code>pearson</code>	the Pearson test statistic (X-squared).
<code>df</code>	the degrees of freedom for the fitted model. There is no adjustment for structural zeros.
<code>margin</code>	list of the margins that were fit. Basically the same as the input <code>margin</code> , but with numbers replaced by names where possible.
<code>fit</code>	An array like <code>table</code> containing the fitted values. Only returned if <code>fit</code> is <code>TRUE</code> .
<code>param</code>	A list containing the estimated parameters of the model. The “standard” constraints of zero marginal sums (e.g., zero row and column sums for a two factor parameter) are employed. Only returned if <code>param</code> is <code>TRUE</code> .

**Author(s)**

Kurt Hornik

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Haberman, S. J. (1972) Log-linear fit for contingency tables—Algorithm AS51. *Applied Statistics*, **21**, 218–225.

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley.

**See Also**

[table](#)

**Examples**

```
## Model of joint independence of sex from hair and eye color.
fm <- loglin(HairEyeColor, list(c(1, 2), c(1, 3), c(2, 3)))
fm
1 - pchisq(fm$lrt, fm$df)
## Model with no three-factor interactions fits well.
```

---

 Lognormal

*The Log Normal Distribution*


---

**Description**

Density, distribution function, quantile function and random generation for the log normal distribution whose logarithm has mean equal to `meanlog` and standard deviation equal to `sdlog`.

**Usage**

```
dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)
plnorm(q, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
qlnorm(p, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)
rlnorm(n, meanlog = 0, sdlog = 1)
```

**Arguments**

`x`, `q`            vector of quantiles.  
`p`                    vector of probabilities.  
`n`                    number of observations. If `length(n) > 1`, the length is taken to be the number required.  
`meanlog`, `sdlog`      mean and standard deviation of the distribution on the log scale with default values of 0 and 1 respectively.

`log`, `log.p` logical; if TRUE, probabilities p are given as log(p).  
`lower.tail` logical; if TRUE (default), probabilities are  $P[X \leq x]$ , otherwise,  $P[X > x]$ .

### Details

The log normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\log(x)-\mu)^2/2\sigma^2}$$

where  $\mu$  and  $\sigma$  are the mean and standard deviation of the logarithm. The mean is  $E(X) = \exp(\mu + 1/2\sigma^2)$ , and the variance  $Var(X) = \exp(2\mu + \sigma^2)(\exp(\sigma^2) - 1)$  and hence the coefficient of variation is  $\sqrt{\exp(\sigma^2) - 1}$  which is approximately  $\sigma$  when that is small (e.g.,  $\sigma < 1/2$ ).

### Value

`dlnorm` gives the density, `plnorm` gives the distribution function, `qlnorm` gives the quantile function, and `rlnorm` generates random deviates.

### Note

The cumulative hazard  $H(t) = -\log(1 - F(t))$  is `-plnorm(t, r, lower = FALSE, log = TRUE)`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[dnorm](#) for the normal distribution.

### Examples

```
dlnorm(1) == dnorm(0)
```

lowess

*Scatter Plot Smoothing*

### Description

This function performs the computations for the *LOWESS* smoother (see the reference below). `lowess` returns a list containing components `x` and `y` which give the coordinates of the smooth. The smooth should be added to a plot of the original points with the function `lines`.

### Usage

```
lowess(x, y = NULL, f = 2/3, iter = 3,
       delta = 0.01 * diff(range(xy$x[0])))
```

**Arguments**

<code>x</code> , <code>y</code>	vectors giving the coordinates of the points in the scatter plot. Alternatively a single plotting structure can be specified.
<code>f</code>	the smoother span. This gives the proportion of points in the plot which influence the smooth at each value. Larger values give more smoothness.
<code>iter</code>	the number of robustifying iterations which should be performed. Using smaller values of <code>iter</code> will make <code>lowess</code> run faster.
<code>delta</code>	values of <code>x</code> which lie within <code>delta</code> of each other are replaced by a single value in the output from <code>lowess</code> . Defaults to 1/100th of the range of <code>x</code> .

**References**

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
- Cleveland, W. S. (1979) Robust locally weighted regression and smoothing scatterplots. *J. Amer. Statist. Assoc.* **74**, 829–836.
- Cleveland, W. S. (1981) LOWESS: A program for smoothing scatterplots by robust locally weighted regression. *The American Statistician*, **35**, 54.

**See Also**

[loess](#)), a newer formula based version of `lowess` (with different defaults!).

**Examples**

```
plot(cars, main = "lowess(cars)")
lines(lowess(cars), col = 2)
lines(lowess(cars, f=.2), col = 3)
legend(5, 120, c(paste("f = ", c("2/3", ".2"))), lty = 1, col = 2:3)
```

---

ls.diag

*Compute Diagnostics for 'lsfit' Regression Results*

---

**Description**

Computes basic statistics, including standard errors, t- and p-values for the regression coefficients.

**Usage**

```
ls.diag(ls.out)
```

**Arguments**

`ls.out` Typically the result of `lsfit()`

**Value**

A list with the following numeric components.

std.dev	The standard deviation of the errors, an estimate of $\sigma$ .
hat	diagonal entries $h_{ii}$ of the hat matrix $H$
std.res	standardized residuals
stud.res	studentized residuals
cooks	Cook's distances
dfits	DFITS statistics
correlation	correlation matrix
std.err	standard errors of the regression coefficients
cov.scaled	Scaled covariance matrix of the coefficients
cov.unscaled	Unscaled covariance matrix of the coefficients

**References**

Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.

**See Also**

[hat](#) for the hat matrix diagonals, [ls.print](#), [lm.influence](#), [summary.lm](#), [anova](#).

**Examples**

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = as.numeric(gl(2, 10, 20)), y = weight)
dlsD9 <- ls.diag(lsD9)
str(dlsD9, give.attr=FALSE)
abs(1 - sum(dlsD9$hat) / 2) < 10*.Machine$double.eps # sum(h.ii) = p
plot(dlsD9$hat, dlsD9$stud.res, xlim=c(0,0.11))
abline(h = 0, lty = 2, col = "lightgray")
```

---

ls.print

*Print 'lsfit' Regression Results*

---

**Description**

Computes basic statistics, including standard errors, t- and p-values for the regression coefficients and prints them if `print.it` is TRUE.

**Usage**

```
ls.print(ls.out, digits = 4, print.it = TRUE)
```

**Arguments**

<code>ls.out</code>	Typically the result of <code>lsfit()</code>
<code>digits</code>	The number of significant digits used for printing
<code>print.it</code>	a logical indicating whether the result should also be printed

**Value**

	A list with the components
<code>summary</code>	The ANOVA table of the regression
<code>coef.table</code>	matrix with regression coefficients, standard errors, t- and p-values

**Note**

Usually, you'd rather use `summary(lm(...))` and `anova(lm(...))` for obtaining similar output.

**See Also**

[ls.diag](#), [lsfit](#), also for examples; [lm](#), [lm.influence](#) which usually are preferable.

---

lsfit

*Find the Least Squares Fit*

---

**Description**

The least squares estimate of  $\beta$  in the model

$$Y = X\beta + \epsilon$$

is found.

**Usage**

```
lsfit(x, y, wt = NULL, intercept = TRUE, tolerance = 1e-07, yname = NULL)
```

**Arguments**

<code>x</code>	a matrix whose rows correspond to cases and whose columns correspond to variables.
<code>y</code>	the responses, possibly a matrix if you want to fit multiple left hand sides.
<code>wt</code>	an optional vector of weights for performing weighted least squares.
<code>intercept</code>	whether or not an intercept term should be used.
<code>tolerance</code>	the tolerance to be used in the matrix decomposition.
<code>yname</code>	names to be used for the response variables.

**Details**

If weights are specified then a weighted least squares is performed with the weight given to the  $j$ th case specified by the  $j$ th entry in `wt`.

If any observation has a missing value in any field, that observation is removed before the analysis is carried out. This can be quite inefficient if there is a lot of missing data.

The implementation is via a modification of the LINPACK subroutines which allow for multiple left-hand sides.

**Value**

A list with the following named components:

<code>coef</code>	the least squares estimates of the coefficients in the model ( $\beta$ as stated above).
<code>residuals</code>	residuals from the fit.
<code>intercept</code>	indicates whether an intercept was fitted.
<code>qr</code>	the QR decomposition of the design matrix.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`lm` which usually is preferable; `ls.print`, `ls.diag`.

**Examples**

```
##-- Using the same data as the lm(.) example:
lsD9 <- lsfit(x = unclass(gl(2,10)), y = weight)
ls.print(lsD9)
```

---

mad

*Median Absolute Deviation*


---

**Description**

Compute the median absolute deviation, i.e., the (lo-/hi-) median of the absolute deviations from the median, and (by default) adjust by a factor for asymptotically normal consistency.

**Usage**

```
mad(x, center = median(x), constant = 1.4826, na.rm = FALSE,
    low = FALSE, high = FALSE)
```

**Arguments**

<code>x</code>	a numeric vector.
<code>center</code>	Optionally, the centre: defaults to the median.
<code>constant</code>	scale factor.
<code>na.rm</code>	if TRUE then NA values are stripped from <code>x</code> before computation takes place.
<code>low</code>	if TRUE, compute the “lo-median”, i.e., for even sample size, do not average the two middle values, but take the smaller one.
<code>high</code>	if TRUE, compute the “hi-median”, i.e., take the larger of the two middle values for even sample size.

**Details**

The actual value calculated is `constant * cMedian(abs(x - center))` with the default value of `center` being `median(x)`, and `cMedian` being the usual, the “low” or “high” median, see the arguments description for `low` and `high` above.

The default `constant = 1.4826` (approximately  $1/\Phi^{-1}(\frac{3}{4}) = 1/\text{qnorm}(3/4)$ ) ensures consistency, i.e.,

$$E[\text{mad}(X_1, \dots, X_n)] = \sigma$$

for  $X_i$  distributed as  $N(\mu, \sigma^2)$  and large  $n$ .

If `na.rm` is TRUE then NA values are stripped from `x` before computation takes place. If this is not done then an NA value in `x` will cause `mad` to return NA.

**See Also**

[IQR](#) which is simpler but less robust, [median](#), [var](#).

**Examples**

```
mad(c(1:9))
print(mad(c(1:9), constant=1)) ==
      mad(c(1:8,100), constant=1) # = 2 ; TRUE
x <- c(1,2,3, 5,7,8)
sort(abs(x - median(x)))
c(mad(x, co=1), mad(x, co=1, lo = TRUE), mad(x, co=1, hi = TRUE))
```

---

 mahalanobis

*Mahalanobis Distance*


---

**Description**

Returns the Mahalanobis distance of all rows in `x` and the vector  $\mu = \text{center}$  with respect to  $\Sigma = \text{cov}$ . This is (for vector `x`) defined as

$$D^2 = (x - \mu)' \Sigma^{-1} (x - \mu)$$

**Usage**

```
mahalanobis(x, center, cov, inverted=FALSE, tol.inv = 1e-7)
```

**Arguments**

**x** vector or matrix of data with, say,  $p$  columns.

**center** mean vector of the distribution or second data vector of length  $p$ .

**cov** covariance matrix ( $p \times p$ ) of the distribution.

**inverted** logical. If TRUE, **cov** is supposed to contain the *inverse* of the covariance matrix.

**tol.inv** tolerance to be used for computing the inverse (if **inverted** is false), see [solve](#).

**Author(s)**

Friedrich Leisch

**See Also**

[cov](#), [var](#)

**Examples**

```
ma <- cbind(1:6, 1:3)
(S <- var(ma))
mahalanobis(c(0,0), 1:2, S)

x <- matrix(rnorm(100*3), ncol = 3)
stopifnot(mahalanobis(x, 0, diag(ncol(x))) == rowSums(x*x))
  ##- Here, D^2 = usual Euclidean distances
Sx <- cov(x)
D2 <- mahalanobis(x, rowMeans(x), Sx)
plot(density(D2, bw=.5), main="Mahalanobis distances, n=100, p=3"); rug(D2)
qqplot(qchisq(ppoints(100), df=3), D2,
  main = expression("Q-Q plot of Mahalanobis" * ~D^2 *
    " vs. quantiles of" * ~ chi[3]^2))
abline(0, 1, col = 'gray')
```

---

make.link

*Create a Link for GLM families*

---

**Description**

This function is used with the [family](#) functions in [glm\(\)](#). Given a link, it returns a link function, an inverse link function, the derivative  $d\mu/d\eta$  and a function for domain checking.

**Usage**

```
make.link(link)
```

**Arguments**

`link` character or numeric; one of "logit", "probit", "cloglog", "identity", "log", "sqrt", "1/mu^2", "inverse", or number, say  $\lambda$  resulting in power link =  $\mu^\lambda$ .

**Value**

A list with components

<code>linkfun</code>	Link function <code>function(mu)</code>
<code>linkinv</code>	Inverse link function <code>function(eta)</code>
<code>mu.eta</code>	Derivative function <code>(eta) dμ/dη</code>
<code>valideta</code>	<code>function(eta) { TRUE if all of eta is in the domain of linkinv }</code> .

**See Also**

[glm, family.](#)

**Examples**

```
str(make.link("logit"))

l2 <- make.link(2)
l2$linkfun(0:3) # 0 1 4 9
l2$mu.eta(eta= 1:2) #= 1/(2*sqrt(eta))
```

---

makepredictcall      *Utility Function for Safe Prediction*

---

**Description**

A utility to help `model.frame.default` create the right matrices when predicting from models with terms like `poly` or `ns`.

**Usage**

```
makepredictcall(var, call)
```

**Arguments**

<code>var</code>	A variable.
<code>call</code>	The term in the formula, as a call.

## Details

This is a generic function with methods for `poly`, `bs` and `ns`: the default method handles `scale`. If `model.frame.default` encounters such a term when creating a model frame, it modifies the `predvars` attribute of the terms supplied to replace the term with one that will work for predicting new data. For example `makepredictcall.ns` adds arguments for the knots and intercept.

To make use of this, have your model-fitting function return the `terms` attribute of the model frame, or copy the `predvars` attribute of the `terms` attribute of the model frame to your `terms` object.

To extend this, make sure the term creates variables with a class, and write a suitable method for that class.

## Value

A replacement for `call` for the `predvars` attribute of the terms.

## See Also

`model.frame`, `poly`, `scale`; `bs` and `ns` in package `splines`, `cars`

## Examples

```
## using poly: this did not work in R < 1.5.0
fm <- lm(weight ~ poly(height, 2), data = women)
plot(women, xlab = "Height (in)", ylab = "Weight (lb)")
ht <- seq(57, 73, len = 200)
lines(ht, predict(fm, data.frame(height=ht)))

## see also example(cars)

## see bs and ns for spline examples.
```

---

manova

*Multivariate Analysis of Variance*

---

## Description

A class for the multivariate analysis of variance.

## Usage

```
manova(...)
```

## Arguments

... Arguments to be passed to `ao.v`.

**Details**

Class "manova" differs from class "aov" in selecting a different summary method. Function manova calls [aov](#) and then add class "manova" to the result object for each stratum.

**Value**

See [aov](#) and the comments in Details here.

**Note**

manova does not support multistratum analysis of variance, so the formula should not include an Error term.

**References**

- Krzanowski, W. J. (1988) *Principles of Multivariate Analysis. A User's Perspective*. Oxford.
- Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures*. Chapman and Hall.

**See Also**

[aov](#), [summary.manova](#), the latter containing examples.

---

`mantelhaen.test`      *Cochran-Mantel-Haenszel Chi-Squared Test for Count Data*

---

**Description**

Performs a Cochran-Mantel-Haenszel chi-squared test of the null that two nominal variables are conditionally independent in each stratum, assuming that there is no three-way interaction.

**Usage**

```
mantelhaen.test(x, y = NULL, z = NULL,  
               alternative = c("two.sided", "less", "greater"),  
               correct = TRUE, exact = FALSE, conf.level = 0.95)
```

**Arguments**

- `x`            either a 3-dimensional contingency table in array form where each dimension is at least 2 and the last dimension corresponds to the strata, or a factor object with at least 2 levels.
- `y`            a factor object with at least 2 levels; ignored if `x` is an array.
- `z`            a factor object with at least 2 levels identifying to which stratum the corresponding elements in `x` and `y` belong; ignored if `x` is an array.

<code>alternative</code>	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". You can specify just the initial letter. Only used in the 2 by 2 by $K$ case.
<code>correct</code>	a logical indicating whether to apply continuity correction when computing the test statistic. Only used in the 2 by 2 by $K$ case.
<code>exact</code>	a logical indicating whether the Mantel-Haenszel test or the exact conditional test (given the strata margins) should be computed. Only used in the 2 by 2 by $K$ case.
<code>conf.level</code>	confidence level for the returned confidence interval. Only used in the 2 by 2 by $K$ case.

### Details

If  $x$  is an array, each dimension must be at least 2, and the entries should be nonnegative integers. NA's are not allowed. Otherwise,  $x$ ,  $y$  and  $z$  must have the same length. Triples containing NA's are removed. All variables must take at least two different values.

### Value

A list with class "htest" containing the following components:

<code>statistic</code>	Only present if no exact test is performed. In the classical case of a 2 by 2 by $K$ table (i.e., of dichotomous underlying variables), the Mantel-Haenszel chi-squared statistic; otherwise, the generalized Cochran-Mantel-Haenszel statistic.
<code>parameter</code>	the degrees of freedom of the approximate chi-squared distribution of the test statistic (1 in the classical case). Only present if no exact test is performed.
<code>p.value</code>	the p-value of the test.
<code>conf.int</code>	a confidence interval for the common odds ratio. Only present in the 2 by 2 by $K$ case.
<code>estimate</code>	an estimate of the common odds ratio. If an exact test is performed, the conditional Maximum Likelihood Estimate is given; otherwise, the Mantel-Haenszel estimate. Only present in the 2 by 2 by $K$ case.
<code>null.value</code>	the common odds ratio under the null of independence, 1. Only present in the 2 by 2 by $K$ case.
<code>alternative</code>	a character string describing the alternative hypothesis. Only present in the 2 by 2 by $K$ case.
<code>method</code>	a character string indicating the method employed, and whether or not continuity correction was used.
<code>data.name</code>	a character string giving the names of the data.

### Note

The asymptotic distribution is only valid if there is no three-way interaction. In the classical 2 by 2 by  $K$  case, this is equivalent to the conditional odds ratios in each stratum being identical. Currently, no inference on homogeneity of the odds ratios is performed.

See also the example below.

## References

Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 230–235.

## Examples

```
## Agresti (1990), pages 231--237, Penicillin and Rabbits
## Investigation of the effectiveness of immediately injected or 1.5
## hours delayed penicillin in protecting rabbits against a lethal
## injection with beta-hemolytic streptococci.
Rabbits <-
array(c(0, 0, 6, 5,
        3, 0, 3, 6,
        6, 2, 0, 4,
        5, 1, 6, 0,
        2, 5, 0, 0),
      dim = c(2, 2, 5),
      dimnames = list(
        Delay = c("None", "1.5h"),
        Response = c("Cured", "Died"),
        Penicillin.Level = c("1/8", "1/4", "1/2", "1", "4")))
Rabbits
## Classical Mantel-Haenszel test
mantelhaen.test(Rabbits)
## => p = 0.047, some evidence for higher cure rate of immediate
## injection
## Exact conditional test
mantelhaen.test(Rabbits, exact = TRUE)
## => p = 0.040
## Exact conditional test for one-sided alternative of a higher
## cure rate for immediate injection
mantelhaen.test(Rabbits, exact = TRUE, alternative = "greater")
## => p = 0.020

## UC Berkeley Student Admissions
mantelhaen.test(UCBAdmissions)
## No evidence for association between admission and gender
## when adjusted for department. However,
apply(UCBAdmissions, 3, function(x) (x[1,1]*x[2,2])/(x[1,2]*x[2,1]))
## This suggests that the assumption of homogeneous (conditional)
## odds ratios may be violated. The traditional approach would be
## using the Woolf test for interaction:
woolf <- function(x) {
  x <- x + 1 / 2
  k <- dim(x)[3]
  or <- apply(x, 3, function(x) (x[1,1]*x[2,2])/(x[1,2]*x[2,1]))
  w <- apply(x, 3, function(x) 1 / sum(1 / x))
  1 - pchisq(sum(w * (log(or) - weighted.mean(log(or), w)) ^ 2), k - 1)
}
woolf(UCBAdmissions)
## => p = 0.003, indicating that there is significant heterogeneity.
## (And hence the Mantel-Haenszel test cannot be used.)
```

---

mcnemar.test                      *McNemar's Chi-squared Test for Count Data*

---

### Description

Performs McNemar's chi-squared test for symmetry of rows and columns in a two-dimensional contingency table.

### Usage

```
mcnemar.test(x, y = NULL, correct = TRUE)
```

### Arguments

<code>x</code>	either a two-dimensional contingency table in matrix form, or a factor object.
<code>y</code>	a factor object; ignored if <code>x</code> is a matrix.
<code>correct</code>	a logical indicating whether to apply continuity correction when computing the test statistic.

### Details

The null is that the probabilities of being classified into cells  $[i, j]$  and  $[j, i]$  are the same.

If `x` is a matrix, it is taken as a two-dimensional contingency table, and hence its entries should be nonnegative integers. Otherwise, both `x` and `y` must be vectors of the same length. Incomplete cases are removed, the vectors are coerced into factor objects, and the contingency table is computed from these.

Continuity correction is only used in the 2-by-2 case if `correct` is `TRUE`.

### Value

A list with class "htest" containing the following components:

<code>statistic</code>	the value of McNemar's statistic.
<code>parameter</code>	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>method</code>	a character string indicating the type of test performed, and whether continuity correction was used.
<code>data.name</code>	a character string giving the name(s) of the data.

### References

Alan Agresti (1990). *Categorical data analysis*. New York: Wiley. Pages 350–354.

**Examples**

```
## Agresti (1990), p. 350.
## Presidential Approval Ratings.
## Approval of the President's performance in office in two surveys,
## one month apart, for a random sample of 1600 voting-age Americans.
Performance <-
matrix(c(794, 86, 150, 570),
       nr = 2,
       dimnames = list("1st Survey" = c("Approve", "Disapprove"),
                       "2nd Survey" = c("Approve", "Disapprove")))
Performance
mcnemar.test(Performance)
## => very strong association between the two successive ratings
```

---

median	<i>Median Value</i>
--------	---------------------

---

**Description**

Compute the sample median of the vector of values given as its argument.

**Usage**

```
median(x, na.rm = FALSE)
```

**Arguments**

x	a numeric vector containing the values whose median is to be computed.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[quantile](#) for general quantiles.

**Examples**

```
median(1:4)# = 2.5 [even number]
median(c(1:3,100,1000))# = 3 [odd, robust]
```

---

medpolish	<i>Median Polish of a Matrix</i>
-----------	----------------------------------

---

### Description

Fits an additive model using Tukey's *median polish* procedure.

### Usage

```
medpolish(x, eps = 0.01, maxiter = 10, trace.iter = TRUE,  
          na.rm = FALSE)
```

### Arguments

<code>x</code>	a numeric matrix.
<code>eps</code>	real number greater than 0. A tolerance for convergence: see <b>Details</b> .
<code>maxiter</code>	the maximum number of iterations
<code>trace.iter</code>	logical. Should progress in convergence be reported?
<code>na.rm</code>	logical. Should missing values be removed?

### Details

The model fitted is additive (constant + rows + columns). The algorithm works by alternately removing the row and column medians, and continues until the proportional reduction in the sum of absolute residuals is less than `eps` or until there have been `maxiter` iterations. The sum of absolute residuals is printed at each iteration of the fitting process, if `trace.iter` is `TRUE`. If `na.rm` is `FALSE` the presence of any NA value in `x` will cause an error, otherwise NA values are ignored.

`medpolish` returns an object of class `medpolish` (see below). There are printing and plotting methods for this class, which are invoked via the generics `print` and `plot`.

### Value

An object of class `medpolish` with the following named components:

<code>overall</code>	the fitted constant term.
<code>row</code>	the fitted row effects.
<code>col</code>	the fitted column effects.
<code>residuals</code>	the residuals.
<code>name</code>	the name of the dataset.

### References

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

**See Also**

[median](#); [aov](#) for a *mean* instead of *median* decomposition.

**Examples**

```
## Deaths from sport parachuting; from ABC of EDA, p.224:
deaths <-
  rbind(c(14,15,14),
        c( 7, 4, 7),
        c( 8, 2,10),
        c(15, 9,10),
        c( 0, 2, 0))
dimnames(deaths) <- list(c("1-24", "25-74", "75-199", "200++", "NA"),
                        paste(1973:1975))

deaths
(med.d <- medpolish(deaths))
plot(med.d)
## Check decomposition:
all(deaths == med.d$overall + outer(med.d$row,med.d$col, "+") + med.d$resid)
```

---

model.extract

*Extract Components from a Model Frame*

---

**Description**

Returns the response, offset, subset, weights or other special components of a model frame passed as optional arguments to [model.frame](#).

**Usage**

```
model.extract(frame, component)
model.offset(x)
model.response(data, type = "any")
model.weights(x)
```

**Arguments**

frame, x, data	A model frame.
component	literal character string or name. The name of a component to extract, such as "weights", "subset".
type	One of "any", "numeric", "double". Using the either of latter two coerces the result to have storage mode "double".

**Details**

model.extract is provided for compatibility with S, which does not have the more specific functions. It is also useful to extract e.g. the etastart and mustart components of a `glm` fit.

model.offset and model.response are equivalent to model.frame(, "offset") and model.frame(, "response") respectively.

model.weights is slightly different from model.frame(, "weights") in not naming the vector it returns.

**Value**

The specified component of the model frame, usually a vector.

**See Also**

`model.frame`, `offset`

**Examples**

```
a <- model.frame(cbind(ncases,ncontrols) ~ agegp+tobgp+alcgp, data=esoph)
model.extract(a, "response")
stopifnot(model.extract(a, "response") == model.response(a))

a <- model.frame(ncases/(ncases+ncontrols) ~ agegp+tobgp+alcgp,
                 data = esoph, weights = ncases+ncontrols)
model.response(a)
model.extract(a, "weights")

a <- model.frame(cbind(ncases,ncontrols) ~ agegp,
                 something = tobgp, data = esoph)
names(a)
stopifnot(model.extract(a, "something") == esoph$tobgp)
```

---

model.frame

*Extracting the “Environment” of a Model Formula*

---

**Description**

model.frame (a generic function) and its methods return a `data.frame` with the variables needed to use formula and any ... arguments.

**Usage**

```
model.frame(formula, ...)

## Default S3 method:
model.frame(formula, data = NULL,
            subset = NULL, na.action = na.fail,
```

```

        drop.unused.levels = FALSE, xlev = NULL, ...)

## S3 method for class 'aovlist':
model.frame(formula, data = NULL, ...)

## S3 method for class 'glm':
model.frame(formula, ...)

## S3 method for class 'lm':
model.frame(formula, ...)

```

### Arguments

<code>formula</code>	a model <a href="#">formula</a> or <a href="#">terms</a> object or an R object.
<code>data</code>	<code>data.frame</code> , list, environment or object coercible to <code>data.frame</code> containing the variables in <code>formula</code> . Neither a matrix nor an array will be accepted.
<code>subset</code>	a specification of the rows to be used: defaults to all rows. This can be any valid indexing vector (see <a href="#">[.data.frame]</a> ) for the rows of <code>data</code> or if that is not supplied, a data frame made up of the variables used in <code>formula</code> .
<code>na.action</code>	how NAs are treated. The default is first, any <code>na.action</code> attribute of <code>data</code> , second a <code>na.action</code> setting of <a href="#">options</a> , and third <code>na.fail</code> if that is unset. The “factory-fresh” default is <code>na.omit</code> . Another possible value is <code>NULL</code> .
<code>drop.unused.levels</code>	should factors have unused levels dropped? Defaults to <code>FALSE</code> .
<code>xlev</code>	a named list of character vectors giving the full set of levels to be assumed for each factor.
<code>...</code>	further arguments such as <code>data</code> , <code>na.action</code> , <code>subset</code> . Any additional arguments such as <code>offset</code> and <code>weights</code> which reach the default method are used to create further columns in the model frame, with parenthesised names such as <code>"(offset)"</code> .

### Details

Exactly what happens depends on the class and attributes of the object `formula`. If this is an object of fitted-model class such as `"lm"`, the method will either returned the saved model frame used when fitting the model (if any, often selected by argument `model = TRUE`) or pass the call used when fitting on to the default method. The default method itself can cope with rather standard model objects such as those of classes `"lqs"` and `"ppr"` from package **MASS** if no other arguments are supplied.

The rest of this section applies only to the default method.

If either `formula` or `data` is already a model frame (a data frame with a `"terms"` attribute and the other is missing, the model frame is returned. Unless `formula` is a terms object, `terms` is called on it. (If you wish to use the `keep.order` argument of `terms.formula`, pass a terms object rather than a formula.)

All the variables in `formula`, `subset` and in `...` are looked for first in `data` and then in the environment of `formula` (see the help for [formula\(\)](#) for further details) and collected into a

data frame. Then the `subset` expression is evaluated, and it is used as a row index to the data frame. Then the `na.action` function is applied to the data frame (and may well add attributes). The levels of any factors in the data frame are adjusted according to the `drop.unused.levels` and `xlev` arguments.

Unless `na.action = NULL`, time-series attributes will be removed from the variables found (since they will be wrong if NAs are removed).

## Value

A `data.frame` containing the variables used in `formula` plus those specified . . .

## References

Chambers, J. M. (1992) *Data for models*. Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

`model.matrix` for the “design matrix”, `formula` for formulas and `expand.model.frame` for `model.frame` manipulation.

## Examples

```
data.class(model.frame(dist ~ speed, data = cars))
```

---

model.matrix	<i>Construct Design Matrices</i>
--------------	----------------------------------

---

## Description

`model.matrix` creates a design matrix.

## Usage

```
model.matrix(object, ...)
```

```
## Default S3 method:  
model.matrix(object, data = environment(object),  
             contrasts.arg = NULL, xlev = NULL, ...)
```

## Arguments

<code>object</code>	an object of an appropriate class. For the default method, a model formula or terms object.
<code>data</code>	a data frame created with <code>model.frame</code> .

<code>contrasts.arg</code>	A list, whose entries are contrasts suitable for input to the <code>contrasts</code> replacement function and whose names are the names of columns of data containing <code>factors</code> .
<code>xlev</code>	to be used as argument of <code>model.frame</code> if data has no "terms" attribute.
<code>...</code>	further arguments passed to or from other methods.

## Details

`model.matrix` creates a design matrix from the description given in `terms(formula)`, using the data in `data` which must contain columns with the same names as would be created by a call to `model.frame(formula)` or, more precisely, by evaluating `attr(terms(formula), "variables")`. There may be other columns and the order is not important. If `contrasts` is specified it overrides the default factor coding for that variable.

In interactions, the variable whose levels vary fastest is the first one to appear in the formula (and not in the term), so in `~ a + b + b:a` the interaction will have a varying fastest.

By convention, if the response variable also appears on the right-hand side of the formula it is dropped (with a warning), although interactions involving the term are retained.

## Value

The design matrix for a regression model with the specified formula and data.

## References

Chambers, J. M. (1992) *Data for models*. Chapter 3 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

`model.frame`, `model.extract`, `terms`

## Examples

```
ff <- log(Volume) ~ log(Height) + log(Girth)
str(m <- model.frame(ff, trees))
mat <- model.matrix(ff, m)

dd <- data.frame(a = gl(3,4), b = gl(4,1,12))# balanced 2-way
options("contrasts")
model.matrix(~ a + b, dd)
model.matrix(~ a + b, dd, contrasts = list(a="contr.sum"))
model.matrix(~ a + b, dd, contrasts = list(a="contr.sum", b="contr.poly"))
m.orth <- model.matrix(~a+b, dd, contrasts = list(a="contr.helmert"))
crossprod(m.orth)# m.orth is ALMOST orthogonal
```

---

model.tables            *Compute Tables of Results from an Aov Model Fit*

---

### Description

Computes summary tables for model fits, especially complex aov fits.

### Usage

```
model.tables(x, ...)  
  
## S3 method for class 'aov':  
model.tables(x, type = "effects", se = FALSE, cterms, ...)  
  
## S3 method for class 'aovlist':  
model.tables(x, type = "effects", se = FALSE, ...)
```

### Arguments

x	a model object, usually produced by aov
type	type of table: currently only "effects" and "means" are implemented.
se	should standard errors be computed?
cterm	A character vector giving the names of the terms for which tables should be computed. The default is all tables.
...	further arguments passed to or from other methods.

### Details

For type = "effects" give tables of the coefficients for each term, optionally with standard errors.

For type = "means" give tables of the mean response for each combinations of levels of the factors in a term.

### Value

An object of class "tables.aov", as list which may contain components

tables	A list of tables for each requested term.
n	The replication information for each term.
se	Standard error information.

### Warning

The implementation is incomplete, and only the simpler cases have been tested thoroughly. Weighted aov fits are not supported.

**See Also**

[aov](#), [proj](#), [replications](#), [TukeyHSD](#), [se.contrast](#)

**Examples**

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
npk.aov <- aov(yield ~ block + N*P*K, npk)
model.tables(npk.aov, "means", se=TRUE)

## as a test, not particularly sensible statistically
options(contrasts=c("contr.helmert", "contr.treatment"))
npk.aovE <- aov(yield ~ N*P*K + Error(block), npk)
model.tables(npk.aovE, se=TRUE)
model.tables(npk.aovE, "means")
```

---

monthplot

*Plot a Seasonal or other Subseries*

---

**Description**

These functions plot seasonal (or other) subseries of a time series. For each season (or other category), a time series is plotted.

**Usage**

```
monthplot(x, labels = NULL, times, phase, base, choice, ...)
```

**Arguments**

x	Time series or related object.
labels	Labels to use for each “season”.
times	Time of each observation.
phase	Indicator for each “season”.
base	Function to use for reference line for subseries.
choice	Which series of an <code>stl</code> or <code>StructTS</code> object?
...	Graphical parameters.

## Details

These functions extract subseries from a time series and plot them all in one frame. The `ts`, `stl`, and `StructTS` methods use the internally recorded frequency and start and finish times to set the scale and the seasons. The default method assumes observations come in groups of 12 (though this can be changed).

If the `labels` are not given but the `phase` is given, then the `labels` default to the unique values of the `phase`. If both are given, then the `phase` values are assumed to be indices into the `labels` array, i.e., they should be in the range from 1 to `length(labels)`.

## Value

These functions are executed for their side effect of drawing a seasonal subseries plot on the current graphical window.

## Author(s)

Duncan Murdoch

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

`ts`, `stl`, `StructTS`

## Examples

```
## The CO2 data
fit <- stl(log(co2), s.window = 20, t.window = 20)
plot(fit)
op <- par(mfrow = c(2,2))
monthplot(co2, ylab = "data", cex.axis = 0.8)
monthplot(fit, choice = "seasonal", cex.axis = 0.8)
monthplot(fit, choice = "trend", cex.axis = 0.8)
monthplot(fit, choice = "remainder", type = "h", cex.axis = 0.8)
par(op)

## The CO2 data, grouped quarterly
quarter <- (cycle(co2) - 1) %/% 3
monthplot(co2, phase = quarter)

## see also JohnsonJohnson
```

---

mood.test

*Mood Two-Sample Test of Scale*


---

### Description

Performs Mood's two-sample test for a difference in scale parameters.

### Usage

```
mood.test(x, ...)

## Default S3 method:
mood.test(x, y, alternative = c("two.sided", "less", "greater"), ...)

## S3 method for class 'formula':
mood.test(formula, data, subset, na.action, ...)
```

### Arguments

<code>x</code> , <code>y</code>	numeric vectors of data values.
<code>alternative</code>	indicates the alternative hypothesis and must be one of "two.sided" (default), "greater" or "less" all of which can be abbreviated.
<code>formula</code>	a formula of the form <code>lhs ~ rhs</code> where <code>lhs</code> is a numeric variable giving the data values and <code>rhs</code> a factor with two levels giving the corresponding groups.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

### Details

The underlying model is that the two samples are drawn from  $f(x-l)$  and  $f((x-l)/s)/s$ , respectively, where  $l$  is a common location parameter and  $s$  is a scale parameter.

The null hypothesis is  $s = 1$ .

There are more useful tests for this problem.

### Value

A list with class "htest" containing the following components:

<code>statistic</code>	the value of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>alternative</code>	a character string describing the alternative hypothesis.
<code>method</code>	the character string "Mood two-sample test of scale".
<code>data.name</code>	a character string giving the names of the data.

**References**

William J. Conover (1971), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 234f.

**See Also**

[fligner.test](#) for a rank-based (nonparametric) k-sample test for homogeneity of variances; [ansari.test](#) for another rank-based two-sample test for a difference in scale parameters; [var.test](#) and [bartlett.test](#) for parametric tests for the homogeneity in variance.

**Examples**

```
## Same data as for the Ansari-Bradley test:
## Serum iron determination using Hyland control sera
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
           101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
               100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
mood.test(ramsay, jung.parekh)
## Compare this to ansari.test(ramsay, jung.parekh)
```

---

 Multinomial

*The Multinomial Distribution*


---

**Description**

Generate multinomially distributed random number vectors and compute multinomial “density” probabilities.

**Usage**

```
rmultinom(n, size, prob)
dmultinom(x, size = NULL, prob, log = FALSE)
```

**Arguments**

<code>x</code>	vector of length $K$ of integers in $0 : \text{size}$ .
<code>n</code>	number of random vectors to draw.
<code>size</code>	integer, say $N$ , specifying the total number of objects that are put into $K$ boxes in the typical multinomial experiment. For <code>dmultinom</code> , it defaults to <code>sum(x)</code> .
<code>prob</code>	numeric non-negative vector of length $K$ , specifying the probability for the $K$ classes; is internally normalized to sum 1.
<code>log</code>	logical; if TRUE, log probabilities are computed.

**Details**

If  $\mathbf{x}$  is a  $K$ -component vector, `dmultinom(x, prob)` is the probability

$$P(X_1 = x_1, \dots, X_K = x_k) = C \times \prod_{j=1}^K \pi_j^{x_j}$$

where  $C$  is the “multinomial coefficient”  $C = N!/(x_1! \cdots x_K!)$  and  $N = \sum_{j=1}^K x_j$ .  
By definition, each component  $X_j$  is binomially distributed as  $\text{Bin}(\text{size}, \text{prob}[j])$  for  $j = 1, \dots, K$ .

The `rmultinom()` algorithm draws binomials from  $\text{Bin}(n_j, P_j)$  sequentially, where  $n_1 = N$  ( $N := \text{size}$ ),  $P_1 = \pi_1$  ( $\pi$  is `prob` scaled to sum 1), and for  $j \geq 2$ , recursively  $n_j = N - \sum_{k=1}^{j-1} n_k$  and  $P_j = \pi_j / (1 - \sum_{k=1}^{j-1} \pi_k)$ .

**Value**

For `rmultinom()`, an integer  $K \times n$  matrix where each column is a random vector generated according to the desired multinomial law, and hence summing to `size`. Whereas the *transposed* result would seem more natural at first, the returned matrix is more efficient because of columnwise storage.

**Note**

`dmultinom` is currently *not vectorized* at all and has no C interface (API); this may be amended in the future.

**See Also**

[rbinom](#) which is a special case conceptually.

**Examples**

```
rmultinom(10, size = 12, prob=c(0.1,0.2,0.8))

pr <- c(1,3,6,10) # normalization not necessary for generation
rmultinom(10, 20, prob = pr)

## all possible outcomes of Multinom(N = 3, K = 3)
X <- t(as.matrix(expand.grid(0:3, 0:3))); X <- X[, colSums(X) <= 3]
X <- rbind(X, 3:3 - colSums(X)); dimnames(X) <- list(letters[1:3], NULL)
X
round(apply(X, 2, function(x) dmultinom(x, prob = c(1,2,5))), 3)
```

---

na.action	<i>NA Action</i>
-----------	------------------

---

**Description**

na.action is a generic function, and na.action.default its default method.

**Usage**

```
na.action(object, ...)
```

**Arguments**

object	any object whose NA action is given.
...	further arguments special methods could require.

**Value**

The “NA action” which should be applied to object whenever NAs are not desired.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

`options("na.action")`, `na.omit`, `na.fail`

**Examples**

```
na.action(c(1, NA))
```

---

na.contiguous	<i>Find Longest Contiguous Stretch of non-NAs</i>
---------------	---

---

**Description**

Find the longest consecutive stretch of non-missing values in a time series object. (In the event of a tie, the first such stretch.)

**Usage**

```
na.contiguous(frame)
```

**Arguments**

frame	a univariate or multivariate time series.
-------	---

**Value**

A time series without missing values. The class of `frame` will be preserved.

**See Also**

`na.omit` and `na.omit.ts`; `na.fail`

**Examples**

```
na.contiguous(presidents)
```

---

`na.fail`

*Handle Missing Values in Objects*

---

**Description**

These generic functions are useful for dealing with **NAs** in e.g., data frames. `na.fail` returns the object if it does not contain any missing values, and signals an error otherwise. `na.omit` returns the object with incomplete cases removed. `na.pass` returns the object unchanged.

**Usage**

```
na.fail(object, ...)
na.omit(object, ...)
na.exclude(object, ...)
na.pass(object, ...)
```

**Arguments**

`object` an R object, typically a data frame  
`...` further arguments special methods could require.

**Details**

At present these will handle vectors, matrices and data frames comprising vectors and matrices (only).

If `na.omit` removes cases, the row numbers of the cases form the `"na.action"` attribute of the result, of class `"omit"`.

`na.exclude` differs from `na.omit` only in the class of the `"na.action"` attribute of the result, which is `"exclude"`. This gives different behaviour in functions making use of `naresid` and `napredict`: when `na.exclude` is used the residuals and predictions are padded to the correct length by inserting **NAs** for cases omitted by `na.exclude`.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

`na.action`; `options` with argument `na.action` for setting “NA actions”; and `lm` and `glm` for functions using these.

**Examples**

```
DF <- data.frame(x = c(1, 2, 3), y = c(0, 10, NA))
na.omit(DF)
m <- as.matrix(DF)
na.omit(m)
stopifnot(all(na.omit(1:3) == 1:3)) # does not affect objects with no NA's
try(na.fail(DF))#> Error: missing values in ...

options("na.action")
```

---

naprint

*Adjust for Missing Values*

---

**Description**

Use missing value information to report the effects of an `na.action`.

**Usage**

```
naprint(x, ...)
```

**Arguments**

<code>x</code>	An object produced by an <code>na.action</code> function.
<code>...</code>	further arguments passed to or from other methods.

**Details**

This is a generic function, and the exact information differs by method. `naprint.omit` reports the number of rows omitted: `naprint.default` reports an empty string.

**Value**

A character string providing information on missing values, for example the number.

---

`naresid`*Adjust for Missing Values*

---

**Description**

Use missing value information to adjust residuals and predictions.

**Usage**

```
naresid(omit, x, ...)  
napredict(omit, x, ...)
```

**Arguments**

<code>omit</code>	an object produced by an <code>na.action</code> function, typically the <code>"na.action"</code> attribute of the result of <code>na.omit</code> or <code>na.exclude</code> .
<code>x</code>	a vector, data frame, or matrix to be adjusted based upon the missing value information.
<code>...</code>	further arguments passed to or from other methods.

**Details**

These are utility functions used to allow `predict` and `resid` methods for modelling functions to compensate for the removal of NAs in the fitting process. They are used by the default, `"lm"` and `"glm"` methods, and by further methods in packages **MASS**, **rpart** and **survival**.

The default methods do nothing. The default method for the `na.exclude` action is to pad the object with NAs in the correct positions to have the same number of rows as the original data frame.

Currently `naresid` and `napredict` are identical, but future methods need not be. `naresid` is used for residuals, and `napredict` for fitted values and predictions.

**Value**

These return a similar object to `x`.

**Note**

Packages `rpart` and `survival5` used to contain versions of these functions that had an `na.omit` action equivalent to that now used for `na.exclude`.

NegBinomial

*The Negative Binomial Distribution***Description**

Density, distribution function, quantile function and random generation for the negative binomial distribution with parameters `size` and `prob`.

**Usage**

```
dnbinom(x, size, prob, mu, log = FALSE)
pnbinom(q, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
qnbinom(p, size, prob, mu, lower.tail = TRUE, log.p = FALSE)
rnbinom(n, size, prob, mu)
```

**Arguments**

<code>x</code>	vector of (non-negative integer) quantiles.
<code>q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>size</code>	target for number of successful trials, or dispersion parameter (the shape parameter of the gamma mixing distribution).
<code>prob</code>	probability of success in each trial.
<code>mu</code>	alternative parametrization via mean: see Details
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

The negative binomial distribution with `size = n` and `prob = p` has density

$$p(x) = \frac{\Gamma(x+n)}{\Gamma(n)x!} p^n (1-p)^x$$

for  $x = 0, 1, 2, \dots$

This represents the number of failures which occur in a sequence of Bernoulli trials before a target number of successes is reached.

A negative binomial distribution can arise as a mixture of Poisson distributions with mean distributed as a  $\Gamma$  ([pgamma](#)) distribution with scale parameter  $(1 - \text{prob})/\text{prob}$  and shape parameter `size`. (This definition allows non-integer values of `size`.) In this model `prob = scale/(1+scale)`, and the mean is `size * (1 - prob)/prob`.

The alternative parametrization (often used in ecology) is by the *mean*  $\mu$ , and *size*, the *dispersion parameter*, where  $\text{prob} = \text{size} / (\text{size} + \mu)$ . The variance is  $\mu + \mu^2 / \text{size}$  in this parametrization or  $n(1 - p)/p^2$  in the first one.

If an element of  $x$  is not integer, the result of `dnbinom` is zero, with a warning.

The quantile is defined as the smallest value  $x$  such that  $F(x) \geq p$ , where  $F$  is the distribution function.

## Value

`dnbinom` gives the density, `pnbinom` gives the distribution function, `qnbinom` gives the quantile function, and `rnbinom` generates random deviates.

## See Also

[dbinom](#) for the binomial, [dpois](#) for the Poisson and [dgeom](#) for the geometric distribution, which is a special case of the negative binomial.

## Examples

```
x <- 0:11
dnbinom(x, size = 1, prob = 1/2) * 2^(1 + x) # == 1
126 / dnbinom(0:8, size = 2, prob = 1/2) #- theoretically integer

## Cumulative ('p') = Sum of discrete prob.s ('d'); Relative error :
summary(1 - cumsum(dnbinom(x, size = 2, prob = 1/2)) /
        pnbinom(x, size = 2, prob = 1/2))

x <- 0:15
size <- (1:20)/4
persp(x,size, dnb <- outer(x,size,function(x,s)dnbinom(x,s, pr= 0.4)),
      xlab = "x", ylab = "s", zlab="density", theta = 150)
title(tit <- "negative binomial density(x,s, pr = 0.4) vs. x & s")

image (x,size, log10(dnb), main= paste("log [",tit,""])
contour(x,size, log10(dnb),add=TRUE)

## Alternative parametrization
x1 <- rnbinom(500, mu = 4, size = 1)
x2 <- rnbinom(500, mu = 4, size = 10)
x3 <- rnbinom(500, mu = 4, size = 100)
h1 <- hist(x1, breaks = 20, plot = FALSE)
h2 <- hist(x2, breaks = h1$breaks, plot = FALSE)
h3 <- hist(x3, breaks = h1$breaks, plot = FALSE)
barplot(rbind(h1$counts, h2$counts, h3$counts),
        beside = TRUE, col = c("red","blue","cyan"),
        names.arg = round(h1$breaks[-length(h1$breaks)]))
```

---

nextn	<i>Highly Composite Numbers</i>
-------	---------------------------------

---

### Description

`nextn` returns the smallest integer, greater than or equal to `n`, which can be obtained as a product of powers of the values contained in `factors`. `nextn` is intended to be used to find a suitable length to zero-pad the argument of `fft` to so that the transform is computed quickly. The default value for `factors` ensures this.

### Usage

```
nextn(n, factors = c(2,3,5))
```

### Arguments

`n` an integer.  
`factors` a vector of positive integer factors.

### See Also

[convolve](#), [fft](#).

### Examples

```
nextn(1001) # 1024  
table(sapply(599:630, nextn))
```

---

nlm	<i>Non-Linear Minimization</i>
-----	--------------------------------

---

### Description

This function carries out a minimization of the function `f` using a Newton-type algorithm. See the references for details.

### Usage

```
nlm(f, p, hessian = FALSE, typsize=rep(1, length(p)), fscale=1,  
    print.level = 0, ndigit=12, gradtol = 1e-6,  
    stepmax = max(1000 * sqrt(sum((p/typsize)^2)), 1000),  
    steptol = 1e-6, iterlim = 100, check.analyticals = TRUE, ...)
```

**Arguments**

<code>f</code>	the function to be minimized. If the function value has an attribute called <code>gradient</code> or both <code>gradient</code> and <code>hessian</code> attributes, these will be used in the calculation of updated parameter values. Otherwise, numerical derivatives are used. <code>deriv</code> returns a function with suitable <code>gradient</code> attribute. This should be a function of a vector of the length of <code>p</code> followed by any other arguments specified by the <code>...</code> argument.
<code>p</code>	starting parameter values for the minimization.
<code>hessian</code>	if TRUE, the hessian of <code>f</code> at the minimum is returned.
<code>typsize</code>	an estimate of the size of each parameter at the minimum.
<code>fscale</code>	an estimate of the size of <code>f</code> at the minimum.
<code>print.level</code>	this argument determines the level of printing which is done during the minimization process. The default value of 0 means that no printing occurs, a value of 1 means that initial and final details are printed and a value of 2 means that full tracing information is printed.
<code>ndigit</code>	the number of significant digits in the function <code>f</code> .
<code>gradtol</code>	a positive scalar giving the tolerance at which the scaled gradient is considered close enough to zero to terminate the algorithm. The scaled gradient is a measure of the relative change in <code>f</code> in each direction <code>p[i]</code> divided by the relative change in <code>p[i]</code> .
<code>stepmax</code>	a positive scalar which gives the maximum allowable scaled step length. <code>stepmax</code> is used to prevent steps which would cause the optimization function to overflow, to prevent the algorithm from leaving the area of interest in parameter space, or to detect divergence in the algorithm. <code>stepmax</code> would be chosen small enough to prevent the first two of these occurrences, but should be larger than any anticipated reasonable step.
<code>steptol</code>	A positive scalar providing the minimum allowable relative step length.
<code>iterlim</code>	a positive integer specifying the maximum number of iterations to be performed before the program is terminated.
<code>check.analyticals</code>	a logical scalar specifying whether the analytic gradients and Hessians, if they are supplied, should be checked against numerical derivatives at the initial parameter values. This can help detect incorrectly formulated gradients or Hessians.
<code>...</code>	additional arguments to <code>f</code> .

**Details**

If a gradient or hessian is supplied but evaluates to the wrong mode or length, it will be ignored if `check.analyticals = TRUE` (the default) with a warning. The hessian is not even checked unless the gradient is present and passes the sanity checks.

From the three methods available in the original source, we always use method “1” which is line search.

**Value**

A list containing the following components:

minimum	the value of the estimated minimum of $f$ .
estimate	the point at which the minimum value of $f$ is obtained.
gradient	the gradient at the estimated minimum of $f$ .
hessian	the hessian at the estimated minimum of $f$ (if requested).
code	an integer indicating why the optimization process terminated. <ol style="list-style-type: none"> <li><b>1:</b> relative gradient is close to zero, current iterate is probably solution.</li> <li><b>2:</b> successive iterates within tolerance, current iterate is probably solution.</li> <li><b>3:</b> last global step failed to locate a point lower than <code>estimate</code>. Either <code>estimate</code> is an approximate local minimum of the function or <code>steptol</code> is too small.</li> <li><b>4:</b> iteration limit exceeded.</li> <li><b>5:</b> maximum step size <code>stepmax</code> exceeded five consecutive times. Either the function is unbounded below, becomes asymptotic to a finite value from above in some direction or <code>stepmax</code> is too small.</li> </ol>
iterations	the number of iterations performed.

**References**

Dennis, J. E. and Schnabel, R. B. (1983) *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, NJ.

Schnabel, R. B., Koontz, J. E. and Weiss, B. E. (1985) A modular system of algorithms for unconstrained minimization. *ACM Trans. Math. Software*, **11**, 419–440.

**See Also**

[optim](#), [optimize](#) for one-dimensional minimization and [uniroot](#) for root finding, [deriv](#) to calculate analytical derivatives.

For nonlinear regression, [nls](#) may be better.

**Examples**

```
f <- function(x) sum((x-1:length(x))^2)
nlm(f, c(10,10))
nlm(f, c(10,10), print.level = 2)
str(nlm(f, c(5), hessian = TRUE))

f <- function(x, a) sum((x-a)^2)
nlm(f, c(10,10), a=c(3,5))
f <- function(x, a)
{
  res <- sum((x-a)^2)
  attr(res, "gradient") <- 2*(x-a)
  res
}
```

```
nlm(f, c(10,10), a=c(3,5))

## more examples, including the use of derivatives.
## Not run: demo(nlm)
```

---

nls

*Nonlinear Least Squares*


---

### Description

Determine the nonlinear least-squares estimates of the nonlinear model parameters and return a class `nls` object.

### Usage

```
nls(formula, data = parent.frame(), start, control = nls.control(),
    algorithm = "default", trace = FALSE, subset,
    weights, na.action, model = FALSE)
```

### Arguments

<code>formula</code>	a nonlinear model formula including variables and parameters
<code>data</code>	an optional data frame in which to evaluate the variables in <code>formula</code>
<code>start</code>	a named list or named numeric vector of starting estimates
<code>control</code>	an optional list of control settings. See <code>nls.control</code> for the names of the settable control values and their effect.
<code>algorithm</code>	character string specifying the algorithm to use. The default algorithm is a Gauss-Newton algorithm. The other alternative is "plinear", the Golub-Pereyra algorithm for partially linear least-squares models.
<code>trace</code>	logical value indicating if a trace of the iteration progress should be printed. Default is <code>FALSE</code> . If <code>TRUE</code> the residual sum-of-squares and the parameter values are printed at the conclusion of each iteration. When the "plinear" algorithm is used, the conditional estimates of the linear parameters are printed after the nonlinear parameters.
<code>subset</code>	an optional vector specifying a subset of observations to be used in the fitting process.
<code>weights</code>	an optional numeric vector of (fixed) weights. When present, the objective function is weighted least squares. <i>not yet implemented</i>
<code>na.action</code>	a function which indicates what should happen when the data contain NAs.
<code>model</code>	logical. If true, the model frame is returned as part of the object.

## Details

### Do not use nls on artificial "zero-residual" data.

The `nls` function uses a relative-offset convergence criterion that compares the numerical imprecision at the current parameter estimates to the residual sum-of-squares. This performs well on data of the form

$$y = f(x, \theta) + \epsilon$$

(with  $\text{var}(\epsilon) > 0$ ). It fails to indicate convergence on data of the form

$$y = f(x, \theta)$$

because the criterion amounts to comparing two components of the round-off error. If you wish to test `nls` on artificial data please add a noise component, as shown in the example below.

An `nls` object is a type of fitted model object. It has methods for the generic functions `coef`, `formula`, `resid`, `print`, `summary`, `AIC`, `fitted` and `vcov`.

Variables in `formula` are looked for first in `data`, then the environment of `formula` (added in 1.9.1) and finally along the search path. Functions in `formula` are searched for first in the environment of `formula` (added in 1.9.1) and then along the search path.

## Value

A list of

<code>m</code>	an <code>nlsModel</code> object incorporating the model
<code>data</code>	the expression that was passed to <code>nls</code> as the data argument. The actual data values are present in the environment of the <code>m</code> component.

## Author(s)

Douglas M. Bates and Saikat DebRoy

## References

Bates, D.M. and Watts, D.G. (1988) *Nonlinear Regression Analysis and Its Applications*, Wiley

Bates, D. M. and Chambers, J. M. (1992) *Nonlinear models*. Chapter 10 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

## See Also

[nlsModel](#)

## Examples

```
DNase1 <- DNase[ DNase$Run == 1, ]
## using a selfStart model
fm1DNase1 <- nls( density ~ SSlogis( log(conc), Asym, xmid, scal ), DNase1 )
summary( fm1DNase1 )
## using conditional linearity
fm2DNase1 <- nls( density ~ 1/(1 + exp(( xmid - log(conc) )/scal ) ),
```

```

        data = DNase1,
        start = list( xmid = 0, scal = 1 ),
        alg = "plinear", trace = TRUE )
summary( fm2DNase1 )
## without conditional linearity
fm3DNase1 <- nls( density ~ Asym/(1 + exp(( xmid - log(conc) )/scal ) ),
        data = DNase1,
        start = list( Asym = 3, xmid = 0, scal = 1 ),
        trace = TRUE )
summary( fm3DNase1 )

## weighted nonlinear regression
Treated <- Puromycin[Puromycin$state == "treated", ]
weighted.MM <- function(resp, conc, Vm, K)
{
  ## Purpose: exactly as white book p.451 -- RHS for nls()
  ## Weighted version of Michaelis-Menten model
  ## -----
  ## Arguments: 'y', 'x' and the two parameters (see book)
  ## -----
  ## Author: Martin Maechler, Date: 23 Mar 2001, 18:48

  pred <- (Vm * conc)/(K + conc)
  (resp - pred) / sqrt(pred)
}

Pur.wt <- nls( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
        start = list(Vm = 200, K = 0.1),
        trace = TRUE)

## The two examples below show that you can fit a model to
## artificial data with noise but not to artificial data
## without noise.
x = 1:10
y = x                                # perfect fit
yeps = y + rnorm(length(y), sd = 0.01) # added noise
nls(yeps ~ a + b*x, start = list(a = 0.12345, b = 0.54321),
        trace = TRUE)
## Not run:
nls(y ~ a + b*x, start = list(a = 0.12345, b = 0.54321),
        trace = TRUE)
## End(Not run)

```

---

nls.control

*Control the Iterations in nls*


---

## Description

Allow the user to set some characteristics of the nls nonlinear least squares algorithm.

**Usage**

```
nls.control(maxiter = 50, tol = 1e-05, minFactor = 1/1024)
```

**Arguments**

maxiter	A positive integer specifying the maximum number of iterations allowed.
tol	A positive numeric value specifying the tolerance level for the relative offset convergence criterion.
minFactor	A positive numeric value specifying the minimum step-size factor allowed on any step in the iteration. The increment is calculated with a Gauss-Newton algorithm and successively halved until the residual sum of squares has been decreased or until the step-size factor has been reduced below this limit.

**Value**

A list with exactly three components:

```
maxiter  
tol  
minFactor
```

with meanings as explained under 'Arguments'.

**Author(s)**

Douglas Bates and Saikat DebRoy

**References**

Bates and Watts (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley.

**See Also**

[nls](#)

**Examples**

```
nls.control(minFactor = 1/2048)
```

---

nlsModel

*Create an nlsModel Object*


---

### Description

This is the constructor for `nlsModel` objects, which are function closures for several functions in a list. The closure includes a nonlinear model formula, data values for the formula, as well as parameters and their values.

### Usage

```
nlsModel(form, data, start)
```

### Arguments

<code>form</code>	a nonlinear model formula
<code>data</code>	a data frame or a list in which to evaluate the variables from the model formula
<code>start</code>	a named list or named numeric vector of starting estimates for the parameters in the model

### Details

An `nlsModel` object is primarily used within the `nls` function. It encapsulates the model, the data, and the parameters in an environment and provides several methods to access characteristics of the model. It forms an important component of the object returned by the `nls` function.

See [nls](#) for where elements of the formula `form` are looked for. In normal use all the variables will be in `data`.

### Value

The value is a list of functions that share a common environment.

<code>resid</code>	returns the residual vector evaluated at the current parameter values
<code>fitted</code>	returns the fitted responses and their gradient at the current parameter values
<code>formula</code>	returns the model formula
<code>deviance</code>	returns the residual sum-of-squares at the current parameter values
<code>gradient</code>	returns the gradient of the model function at the current parameter values
<code>conv</code>	returns the relative-offset convergence criterion evaluated at the current parameter values
<code>incr</code>	returns the parameter increment calculated according to the Gauss-Newton formula
<code>setPars</code>	a function with one argument, <code>pars</code> . It sets the parameter values for the <code>nlsModel</code> object and returns a logical value denoting a singular gradient array.
<code>getPars</code>	returns the current value of the model parameters as a numeric vector

<code>getAllPars</code>	returns the current value of the model parameters as a numeric vector
<code>getEnv</code>	returns the environment shared by these functions, which contains copies of all the variables in <code>data</code> and has as parent the environment of <code>form</code> .
<code>trace</code>	the function that is called at each iteration if tracing is enabled
<code>Rmat</code>	the upper triangular factor of the gradient array at the current parameter values
<code>predict</code>	takes as argument <code>newdata</code> , a <code>data.frame</code> and returns the predicted response for <code>newdata</code> .

**Author(s)**

Douglas M. Bates and Saikat DebRoy

**References**

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley

**See Also**

[nls](#)

**Examples**

```
DNase1 <- DNase[ DNase$Run == 1, ]
mod <-
  nlsModel(density ~ SSlogis( log(conc), Asym, xmid, scal ),
           DNase1, list( Asym = 3, xmid = 0, scal = 1 ))
mod$getPars()      # returns the parameters as a list
mod$deviance()    # returns the residual sum-of-squares
mod$resid()       # returns the residual vector and the gradient
mod$incr()        # returns the suggested increment
mod$setPars( unlist(mod$getPars()) + mod$incr() ) # set new parameter values
mod$getPars()     # check the parameters have changed
mod$deviance()    # see if the parameter increment was successful
mod$trace()       # check the tracing
mod$Rmat()        # R matrix from the QR decomposition of the gradient
```

---

NLSstAsymptotic      *Fit the Asymptotic Regression Model*

---

**Description**

Fits the asymptotic regression model, in the form  $b_0 + b_1 \cdot (1 - \exp(-\exp(lrc) \cdot x))$  to the `xy` data. This can be used as a building block in determining starting estimates for more complicated models.

**Usage**

```
NLSstAsymptotic(xy)
```

**Arguments**

`xy` a sortedXyData object

**Value**

A numeric value of length 3 with components labelled `b0`, `b1`, and `lrc`. `b0` is the estimated intercept on the  $y$ -axis, `b1` is the estimated difference between the asymptote and the  $y$ -intercept, and `lrc` is the estimated logarithm of the rate constant.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[SSasymp](#)

**Examples**

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]  
NLSstAsymptotic(sortedXyData(expression(age), expression(height), Lob.329 ))
```

---

NLSstClosestX

*Inverse Interpolation*

---

**Description**

Use inverse linear interpolation to approximate the  $x$  value at which the function represented by `xy` is equal to `yval`.

**Usage**

```
NLSstClosestX(xy, yval)
```

**Arguments**

`xy` a sortedXyData object  
`yval` a numeric value on the  $y$  scale

**Value**

A single numeric value on the  $x$  scale.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[sortedXyData](#), [NLSstLfAsymptote](#), [NLSstRtAsymptote](#), [selfStart](#)

**Examples**

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstClosestX( DN.srt, 1.0 )
```

---

NLSstLfAsymptote *Horizontal Asymptote on the Left Side*

---

**Description**

Provide an initial guess at the horizontal asymptote on the left side (i.e., small values of  $x$ ) of the graph of  $y$  versus  $x$  from the `xy` object. Primarily used within `initial` functions for self-starting nonlinear regression models.

**Usage**

```
NLSstLfAsymptote(xy)
```

**Arguments**

`xy` a `sortedXyData` object

**Value**

A single numeric value estimating the horizontal asymptote for small  $x$ .

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[sortedXyData](#), [NLSstClosestX](#), [NLSstRtAsymptote](#), [selfStart](#)

**Examples**

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstLfAsymptote( DN.srt )
```

---

`NLSstRtAsymptote`     *Horizontal Asymptote on the Right Side*

---

**Description**

Provide an initial guess at the horizontal asymptote on the right side (i.e., large values of  $x$ ) of the graph of  $y$  versus  $x$  from the `xy` object. Primarily used within `initial` functions for self-starting nonlinear regression models.

**Usage**

```
NLSstRtAsymptote(xy)
```

**Arguments**

`xy`                    a `sortedXyData` object

**Value**

A single numeric value estimating the horizontal asymptote for large  $x$ .

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[sortedXyData](#), [NLSstClosestX](#), [NLSstRtAsymptote](#), [selfStart](#)

**Examples**

```
DNase.2 <- DNase[ DNase$Run == "2", ]
DN.srt <- sortedXyData( expression(log(conc)), expression(density), DNase.2 )
NLSstRtAsymptote( DN.srt )
```

---

Normal

*The Normal Distribution*

---

**Description**

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

**Usage**

```
dnorm(x, mean=0, sd=1, log = FALSE)
pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean=0, sd=1)
```

**Arguments**

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>mean</code>	vector of means.
<code>sd</code>	vector of standard deviations.
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

If `mean` or `sd` are not specified they assume the default values of 0 and 1, respectively.

The normal distribution has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

where  $\mu$  is the mean of the distribution and  $\sigma$  the standard deviation.

`qnorm` is based on Wichura's algorithm AS 241 which provides precise results up to about 16 digits.

**Value**

`dnorm` gives the density, `pnorm` gives the distribution function, `qnorm` gives the quantile function, and `rnorm` generates random deviates.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Wichura, M. J. (1988) Algorithm AS 241: The Percentage Points of the Normal Distribution. *Applied Statistics*, **37**, 477–484.

**See Also**

`runif` and `.Random.seed` about random number generation, and `dlnorm` for the Lognormal distribution.

**Examples**

```

dnorm(0) == 1/ sqrt(2*pi)
dnorm(1) == exp(-1/2)/ sqrt(2*pi)
dnorm(1) == 1/ sqrt(2*pi*exp(1))

## Using "log = TRUE" for an extended range :
par(mfrow=c(2,1))
plot(function(x)dnorm(x, log=TRUE), -60, 50, main = "log { Normal density }")
curve(log(dnorm(x)), add=TRUE, col="red",lwd=2)
mtext("dnorm(x, log=TRUE)", adj=0); mtext("log(dnorm(x))", col="red", adj=1)

plot(function(x)pnorm(x, log=TRUE), -50, 10, main = "log { Normal Cumulative }")
curve(log(pnorm(x)), add=TRUE, col="red",lwd=2)
mtext("pnorm(x, log=TRUE)", adj=0); mtext("log(pnorm(x))", col="red", adj=1)

## if you want the so-called 'error function'
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
## (see Abrahamowitz and Stegun 29.2.29)
## and the so-called 'complementary error function'
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower=FALSE)

```

---

numericDeriv

*Evaluate derivatives numerically*


---

**Description**

numericDeriv numerically evaluates the gradient of an expression.

**Usage**

```
numericDeriv(expr, theta, rho = parent.frame())
```

**Arguments**

expr	The expression to be differentiated. The value of this expression should be a numeric vector.
theta	A character vector of names of variables used in expr
rho	An environment containing all the variables needed to evaluate expr

**Details**

This is a front end to the C function `numeric_deriv`, which is described in *Writing R Extensions*.

**Value**

The value of `eval(expr, envir = rho)` plus a matrix attribute called `gradient`. The columns of this matrix are the derivatives of the value with respect to the variables listed in `theta`.

**Author(s)**

Saikat DebRoy <saikat@stat.wisc.edu>

**Examples**

```
myenv <- new.env()
assign("mean", 0., env = myenv)
assign("sd", 1., env = myenv)
assign("x", seq(-3., 3., len = 31), env = myenv)
numericDeriv(quote(pnorm(x, mean, sd)), c("mean", "sd"), myenv)
```

---

offset

*Include an Offset in a Model Formula*

---

**Description**

An offset is a term to be added to a linear predictor, such as in a generalised linear model, with known coefficient 1 rather than an estimated coefficient.

**Usage**

```
offset(object)
```

**Arguments**

object            An offset to be included in a model frame

**Details**

There can be more than one offset in a model formula, but `-` is not supported for `offset` terms (and is equivalent to `+`).

**Value**

The input value.

**See Also**

[model.offset](#), [model.frame](#).

For examples see [glm](#) and [Insurance](#) in package **MASS**.

---

oneway.test                      *Test for Equal Means in a One-Way Layout*

---

### Description

Test whether two or more samples from normal distributions have the same means. The variances are not necessarily assumed to be equal.

### Usage

```
oneway.test(formula, data, subset, na.action, var.equal = FALSE)
```

### Arguments

formula	a formula of the form <code>lhs ~ rhs</code> where <code>lhs</code> gives the sample values and <code>rhs</code> the corresponding groups.
data	an optional data frame containing the variables in the model formula.
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
var.equal	a logical variable indicating whether to treat the variances in the samples as equal. If <code>TRUE</code> , then a simple F test for the equality of means in a one-way analysis of variance is performed. If <code>FALSE</code> , an approximate method of Welch (1951) is used, which generalizes the commonly known 2-sample Welch test to the case of arbitrarily many samples.

### Value

A list with class "htest" containing the following components:

statistic	the value of the test statistic.
parameter	the degrees of freedom of the exact or approximate F distribution of the test statistic.
p.value	the p-value of the test.
method	a character string indicating the test performed.
data.name	a character string giving the names of the data.

### References

B. L. Welch (1951), On the comparison of several mean values: an alternative approach. *Biometrika*, **38**, 330–336.

### See Also

The standard t test ([t.test](#)) as the special case for two samples; the Kruskal-Wallis test [kruskal.test](#) for a nonparametric test for equal location parameters in a one-way layout.

**Examples**

```
## Not assuming equal variances
oneway.test(extra ~ group, data = sleep)
## Assuming equal variances
oneway.test(extra ~ group, data = sleep, var.equal = TRUE)
## which gives the same result as
anova(lm(extra ~ group, data = sleep))
```

---

optim

*General-purpose Optimization*


---

**Description**

General-purpose optimization based on Nelder–Mead, quasi-Newton and conjugate-gradient algorithms. It includes an option for box-constrained optimization and simulated annealing.

**Usage**

```
optim(par, fn, gr = NULL,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE, ...)
```

**Arguments**

<code>par</code>	Initial values for the parameters to be optimized over.
<code>fn</code>	A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.
<code>gr</code>	A function to return the gradient for the "BFGS", "CG" and "L-BFGS-B" methods. If it is <code>NULL</code> , a finite-difference approximation will be used. For the "SANN" method it specifies a function to generate a new candidate point. If it is <code>NULL</code> a default Gaussian Markov kernel is used.
<code>method</code>	The method to be used. See <b>Details</b> .
<code>lower</code> , <code>upper</code>	Bounds on the variables for the "L-BFGS-B" method.
<code>control</code>	A list of control parameters. See <b>Details</b> .
<code>hessian</code>	Logical. Should a numerically differentiated Hessian matrix be returned?
<code>...</code>	Further arguments to be passed to <code>fn</code> and <code>gr</code> .

## Details

By default this function performs minimization, but it will maximize if `control$fnscale` is negative.

The default method is an implementation of that of Nelder and Mead (1965), that uses only function values and is robust but relatively slow. It will work reasonably well for non-differentiable functions.

Method "BFGS" is a quasi-Newton method (also known as a variable metric algorithm), specifically that published simultaneously in 1970 by Broyden, Fletcher, Goldfarb and Shanno. This uses function values and gradients to build up a picture of the surface to be optimized.

Method "CG" is a conjugate gradients method based on that by Fletcher and Reeves (1964) (but with the option of Polak–Ribiere or Beale–Sorenson updates). Conjugate gradient methods will generally be more fragile than the BFGS method, but as they do not store a matrix they may be successful in much larger optimization problems.

Method "L-BFGS-B" is that of Byrd *et. al.* (1995) which allows *box constraints*, that is each variable can be given a lower and/or upper bound. The initial value must satisfy the constraints. This uses a limited-memory modification of the BFGS quasi-Newton method. If non-trivial bounds are supplied, this method will be selected, with a warning.

Nocedal and Wright (1999) is a comprehensive reference for the previous three methods.

Method "SANN" is by default a variant of simulated annealing given in Belisle (1992). Simulated-annealing belongs to the class of stochastic global optimization methods. It uses only function values but is relatively slow. It will also work for non-differentiable functions. This implementation uses the Metropolis function for the acceptance probability. By default the next candidate point is generated from a Gaussian Markov kernel with scale proportional to the actual temperature. If a function to generate a new candidate point is given, method "SANN" can also be used to solve combinatorial optimization problems. Temperatures are decreased according to the logarithmic cooling schedule as given in Belisle (1992, p. 890). Note that the "SANN" method depends critically on the settings of the control parameters. It is not a general-purpose method but can be very useful in getting to a good value on a very rough surface.

Function `fn` can return `NA` or `Inf` if the function cannot be evaluated at the supplied value, but the initial value must have a computable finite value of `fn`. (Except for method "L-BFGS-B" where the values should always be finite.)

`optim` can be used recursively, and for a single parameter as well as many.

The `control` argument is a list that can supply any of the following components:

**trace** Non-negative integer. If positive, tracing information on the progress of the optimization is produced. Higher values may produce more tracing information: for method "L-BFGS-B" there are six levels of tracing. (To understand exactly what these do see the source code: higher levels give more detail.)

**fnscale** An overall scaling to be applied to the value of `fn` and `gr` during optimization. If negative, turns the problem into a maximization problem. Optimization is performed on `fn(par)/fnscale`.

**parscale** A vector of scaling values for the parameters. Optimization is performed on `par/parscale` and these should be comparable in the sense that a unit change in any element produces about a unit change in the scaled value.

- ndeps** A vector of step sizes for the finite-difference approximation to the gradient, on `par/parscale` scale. Defaults to  $1e-3$ .
- maxit** The maximum number of iterations. Defaults to 100 for the derivative-based methods, and 500 for "Nelder-Mead". For "SANN" `maxit` gives the total number of function evaluations. There is no other stopping criterion. Defaults to 10000.
- abstol** The absolute convergence tolerance. Only useful for non-negative functions, as a tolerance for reaching zero.
- reltol** Relative convergence tolerance. The algorithm stops if it is unable to reduce the value by a factor of `reltol * (abs(val) + reltol)` at a step. Defaults to `sqrt(.Machine$double.eps)`, typically about  $1e-8$ .
- alpha, beta, gamma** Scaling parameters for the "Nelder-Mead" method. `alpha` is the reflection factor (default 1.0), `beta` the contraction factor (0.5) and `gamma` the expansion factor (2.0).
- REPORT** The frequency of reports for the "BFGS" and "L-BFGS-B" methods if `control$trace` is positive. Defaults to every 10 iterations.
- type** for the conjugate-gradients method. Takes value 1 for the Fletcher-Reeves update, 2 for Polak-Ribiere and 3 for Beale-Sorenson.
- lmm** is an integer giving the number of BFGS updates retained in the "L-BFGS-B" method, It defaults to 5.
- factr** controls the convergence of the "L-BFGS-B" method. Convergence occurs when the reduction in the objective is within this factor of the machine tolerance. Default is  $1e7$ , that is a tolerance of about  $1e-8$ .
- pgtol** helps controls the convergence of the "L-BFGS-B" method. It is a tolerance on the projected gradient in the current search direction. This defaults to zero, when the check is suppressed.
- temp** controls the "SANN" method. It is the starting temperature for the cooling schedule. Defaults to 10.
- tmax** is the number of function evaluations at each temperature for the "SANN" method. Defaults to 10.

## Value

A list with components:

- |                          |   |
|--------------------------|---|
| <code>par</code>         | The best set of parameters found.   |
| <code>value</code>       | The value of <code>fn</code> corresponding to <code>par</code> .  |
| <code>counts</code>      | A two-element integer vector giving the number of calls to <code>fn</code> and <code>gr</code> respectively. This excludes those calls needed to compute the Hessian, if requested, and any calls to <code>fn</code> to compute a finite-difference approximation to the gradient.  |
| <code>convergence</code> | An integer code. 0 indicates successful convergence. Error codes are <ul style="list-style-type: none"> <li><b>1</b> indicates that the iteration limit <code>maxit</code> had been reached.</li> <li><b>10</b> indicates degeneracy of the Nelder-Mead simplex.</li> <li><b>51</b> indicates a warning from the "L-BFGS-B" method; see component message for further details.</li> </ul> |

	<b>52</b> indicates an error from the "L-BFGS-B" method; see component message for further details.
message	A character string giving any additional information returned by the optimizer, or NULL.
hessian	Only if argument <code>hessian</code> is true. A symmetric matrix giving an estimate of the Hessian at the solution found. Note that this is the Hessian of the unconstrained problem even if the box constraints are active.

### Note

`optim` will work with one-dimensional `pars`, but the default method does not work well (and will warn). Use `optimize` instead.

The code for methods "Nelder-Mead", "BFGS" and "CG" was based originally on Pascal code in Nash (1990) that was translated by `p2c` and then hand-optimized. Dr Nash has agreed that the code can be made freely available.

The code for method "L-BFGS-B" is based on Fortran code by Zhu, Byrd, Lu-Chen and Nocedal obtained from Netlib (file 'opt/lbfgs\_bcm.shar': another version is in 'toms/778').

The code for method "SANN" was contributed by A. Trapletti.

### References

- Belisle, C. J. P. (1992) Convergence theorems for a class of simulated annealing algorithms on  $R^d$ . *J Applied Probability*, **29**, 885–895.
- Byrd, R. H., Lu, P., Nocedal, J. and Zhu, C. (1995) A limited memory algorithm for bound constrained optimization. *SIAM J. Scientific Computing*, **16**, 1190–1208.
- Fletcher, R. and Reeves, C. M. (1964) Function minimization by conjugate gradients. *Computer Journal* **7**, 148–154.
- Nash, J. C. (1990) *Compact Numerical Methods for Computers. Linear Algebra and Function Minimisation*. Adam Hilger.
- Nelder, J. A. and Mead, R. (1965) A simplex algorithm for function minimization. *Computer Journal* **7**, 308–313.
- Nocedal, J. and Wright, S. J. (1999) *Numerical Optimization*. Springer.

### See Also

`nlm`, `optimize`, `constrOptim`

### Examples

```
fr <- function(x) { ## Rosenbrock Banana function
  x1 <- x[1]
  x2 <- x[2]
  100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
grr <- function(x) { ## Gradient of 'fr'
  x1 <- x[1]
  x2 <- x[2]
```

```

      c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
        200 *      (x2 - x1 * x1))
    }
  optim(c(-1.2,1), fr)
  optim(c(-1.2,1), fr, grr, method = "BFGS")
  optim(c(-1.2,1), fr, NULL, method = "BFGS", hessian = TRUE)
  optim(c(-1.2,1), fr, grr, method = "CG")
  optim(c(-1.2,1), fr, grr, method = "CG", control=list(type=2))
  optim(c(-1.2,1), fr, grr, method = "L-BFGS-B")

  flb <- function(x)
    { p <- length(x); sum(c(1, rep(4, p-1)) * (x - c(1, x[-p])^2)^2) }
  ## 25-dimensional box constrained
  optim(rep(3, 25), flb, NULL, "L-BFGS-B",
        lower=rep(2, 25), upper=rep(4, 25)) # par[24] is *not* at boundary

  ## "wild" function , global minimum at about -15.81515
  fw <- function (x)
    10*sin(0.3*x)*sin(1.3*x^2) + 0.00001*x^4 + 0.2*x+80
  plot(fw, -50, 50, n=1000, main = "optim() minimising 'wild function'")

  res <- optim(50, fw, method="SANN",
              control=list(maxit=20000, temp=20, parscale=20))
  res
  ## Now improve locally
  (r2 <- optim(res$par, fw, method="BFGS"))
  points(r2$par, r2$val, pch = 8, col = "red", cex = 2)

  ## Combinatorial optimization: Traveling salesman problem
  library(stats) # normally loaded

  eurodistmat <- as.matrix(eurodist)

  distance <- function(sq) { # Target function
    sq2 <- embed(sq, 2)
    return(sum(eurodistmat[cbind(sq2[,2],sq2[,1])]))
  }

  genseq <- function(sq) { # Generate new candidate sequence
    idx <- seq(2, NROW(eurodistmat)-1, by=1)
    changepoints <- sample(idx, size=2, replace=FALSE)
    tmp <- sq[changepoints[1]]
    sq[changepoints[1]] <- sq[changepoints[2]]
    sq[changepoints[2]] <- tmp
    return(sq)
  }

  sq <- c(1,2:NROW(eurodistmat),1) # Initial sequence
  distance(sq)

  set.seed(2222) # chosen to get a good soln quickly
  res <- optim(sq, distance, genseq, method="SANN",
              control = list(maxit=6000, temp=2000, trace=TRUE))

```

```

res # Near optimum distance around 12842

loc <- cmdscale(eurodist)
rx <- range(x <- loc[,1])
ry <- range(y <- -loc[,2])
tspinit <- loc[sq,]
tspres <- loc[res$par,]
s <- seq(NROW(tspres)-1)

plot(x, y, type="n", asp=1, xlab="", ylab="",
     main="initial solution of traveling salesman problem")
arrows(tspinit[s,1], -tspinit[s,2], tspinit[s+1,1], -tspinit[s+1,2],
       angle=10, col="green")
text(x, y, names(eurodist), cex=0.8)

plot(x, y, type="n", asp=1, xlab="", ylab="",
     main="optim() 'solving' traveling salesman problem")
arrows(tspres[s,1], -tspres[s,2], tspres[s+1,1], -tspres[s+1,2],
       angle=10, col="red")
text(x, y, names(eurodist), cex=0.8)

```

---

optimize

*One Dimensional Optimization*


---

### Description

The function `optimize` searches the interval from `lower` to `upper` for a minimum or maximum of the function `f` with respect to its first argument.

`optimise` is an alias for `optimize`.

### Usage

```

optimize(f = , interval = , lower = min(interval),
        upper = max(interval), maximum = FALSE,
        tol = .Machine$double.eps^0.25, ...)
optimise(f = , interval = , lower = min(interval),
        upper = max(interval), maximum = FALSE,
        tol = .Machine$double.eps^0.25, ...)

```

### Arguments

<code>f</code>	the function to be optimized. The function is either minimized or maximized over its first argument depending on the value of <code>maximum</code> .
<code>interval</code>	a vector containing the end-points of the interval to be searched for the minimum.
<code>lower</code>	the lower end point of the interval to be searched.
<code>upper</code>	the upper end point of the interval to be searched.

maximum	logical. Should we maximize or minimize (the default)?
tol	the desired accuracy.
...	additional arguments to <code>f</code> .

### Details

The method used is a combination of golden section search and successive parabolic interpolation. Convergence is never much slower than that for a Fibonacci search. If `f` has a continuous second derivative which is positive at the minimum (which is not at `lower` or `upper`), then convergence is superlinear, and usually of the order of about 1.324.

The function `f` is never evaluated at two points closer together than  $\epsilon|x_0| + (tol/3)$ , where  $\epsilon$  is approximately `sqrt(.Machine$double.eps)` and  $x_0$  is the final abscissa `optimize()`\$minimum.

If `f` is a unimodal function and the computed values of `f` are always unimodal when separated by at least  $\epsilon|x| + (tol/3)$ , then  $x_0$  approximates the abscissa of the global minimum of `f` on the interval `lower, upper` with an error less than  $\epsilon|x_0| + tol$ .

If `f` is not unimodal, then `optimize()` may approximate a local, but perhaps non-global, minimum to the same accuracy.

The first evaluation of `f` is always at  $x_1 = a + (1 - \phi)(b - a)$  where  $(a, b) = (lower, upper)$  and  $\phi = (\sqrt{5} - 1)/2 = 0.61803\dots$  is the golden section ratio. Almost always, the second evaluation is at  $x_2 = a + \phi(b - a)$ . Note that a local minimum inside  $[x_1, x_2]$  will be found as solution, even when `f` is constant in there, see the last example.

It uses a C translation of Fortran code (from Netlib) based on the Algol 60 procedure `localmin` given in the reference.

### Value

A list with components `minimum` (or `maximum`) and `objective` which give the location of the minimum (or maximum) and the value of the function at that point.

### References

Brent, R. (1973) *Algorithms for Minimization without Derivatives*. Englewood Cliffs N.J.: Prentice-Hall.

### See Also

[nlm](#), [uniroot](#).

### Examples

```
f <- function(x,a) (x-a)^2
xmin <- optimize(f, c(0, 1), tol = 0.0001, a = 1/3)
xmin

## See where the function is evaluated:
optimize(function(x) x^2*(print(x)-1), l=0, u=10)

## "wrong" solution with unlucky interval and piecewise constant f():
```

```
f <- function(x) ifelse(x > -1, ifelse(x < 4, exp(-1/abs(x - 1)), 10), 10)
fp <- function(x) { print(x); f(x) }

plot(f, -2,5, ylim = 0:1, col = 2)
optimize(fp, c(-4, 20))# doesn't see the minimum
optimize(fp, c(-7, 20))# ok
```

---

order.dendrogram    *Ordering or Labels of the Leaves in a Dendrogram*

---

### Description

These functions return the order (index) or the "label" attribute for the leaves in a dendrogram. These indices can then be used to access the appropriate components of any additional data.

### Usage

```
order.dendrogram(x)

## S3 method for class 'dendrogram':
labels(object, ...)
```

### Arguments

`x`, `object`    a dendrogram (see [as.dendrogram](#)).

`...`            additional arguments

### Details

The indices or labels for the leaves in left to right order are retrieved.

### Value

A vector with length equal to the number of leaves in the dendrogram is returned. From `r <- order.dendrogram( )`, each element is the index into the original data (from which the dendrogram was computed).

### Author(s)

R. Gentleman (`order.dendrogram`) and Martin Maechler (`labels.dendrogram`).

### See Also

[reorder.dendrogram](#).

**Examples**

```

set.seed(123)
x <- rnorm(10)
hc <- hclust(dist(x))
hc$order
dd <- as.dendrogram(hc)
order.dendrogram(dd) ## the same :
stopifnot(hc$order == order.dendrogram(dd))

d2 <- as.dendrogram(hclust(dist(USArrests)))
labels(d2) ## in this case the same as
stopifnot(labels(d2) == rownames(USArrests)[order.dendrogram(d2)])

```

---

p.adjust

*Adjust p-values for multiple comparisons*


---

**Description**

Given a set of p-values, returns p-values adjusted using one of several methods.

**Usage**

```
p.adjust(p, method = p.adjust.methods, n = length(p))
```

```
p.adjust.methods
```

**Arguments**

p	vector of p-values
method	correction method
n	number of comparisons

**Details**

The adjustment methods include the Bonferroni correction ("bonferroni") in which the p-values are multiplied by the number of comparisons. Four less conservative corrections are also included by Holm (1979) ("holm"), Hochberg (1988) ("hochberg"), Hommel (1988) ("hommel") and Benjamini & Hochberg (1995) ("fdr"), respectively. A pass-through option ("none") is also included. The set of methods are contained in the p.adjust.methods vector for the benefit of methods that need to have the method as an option and pass it on to p.adjust.

The first four methods are designed to give strong control of the family wise error rate. There seems no reason to use the unmodified Bonferroni correction because it is dominated by Holm's method, which is also valid under arbitrary assumptions.

Hochberg's and Hommel's methods are valid when the hypothesis tests are independent or when they are non-negatively associated (Sarkar, 1998; Sarkar and Chang, 1997). Hommel's method is more powerful than Hochberg's, but the difference is usually small and the Hochberg p-values are faster to compute.

The "fdr" method of Benjamini and Hochberg (1995) controls the false discovery rate, the expected proportion of false discoveries amongst the rejected hypotheses. The false discovery rate is a less stringent condition than the family wise error rate, so Benjamini and Hochberg's method is more powerful than the other methods.

### Value

A vector of corrected p-values.

### References

- Benjamini, Y., and Hochberg, Y. (1995). Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society Series B*, **57**, 289–300.
- Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, **6**, 65–70.
- Hommel, G. (1988). A stagewise rejective multiple test procedure based on a modified Bonferroni test. *Biometrika*, **75**, 383–386.
- Hochberg, Y. (1988). A sharper Bonferroni procedure for multiple tests of significance. *Biometrika*, **75**, 800–803.
- Shaffer, J. P. (1995). Multiple hypothesis testing. *Annual Review of Psychology*, **46**, 561–576. (An excellent review of the area.)
- Sarkar, S. (1998). Some probability inequalities for ordered MTP2 random variables: a proof of Simes conjecture. *Annals of Statistics*, **26**, 494–504.
- Sarkar, S., and Chang, C. K. (1997). Simes' method for multiple hypothesis testing with positively dependent test statistics. *Journal of the American Statistical Association*, **92**, 1601–1608.
- Wright, S. P. (1992). Adjusted P-values for simultaneous inference. *Biometrics*, **48**, 1005–1013. (Explains the adjusted P-value approach.)

### See Also

pairwise.\* functions in the **stats** package, such as [pairwise.t.test](#).

### Examples

```
x <- rnorm(50, m=c(rep(0,25),rep(3,25)))
p <- 2*pnorm(-abs(x))

round(p, 3)

round(p.adjust(p), 3)
round(p.adjust(p,"bonferroni"), 3)

round(p.adjust(p,"fdr"), 3)
```

---

pairwise.prop.test *Pairwise comparisons of proportions*

---

**Description**

Calculate pairwise comparisons between pairs of proportions with correction for multiple testing

**Usage**

```
pairwise.prop.test(x, n, p.adjust.method = p.adjust.methods, ...)
```

**Arguments**

**x** Vector of counts of successes or a matrix with 2 columns giving the counts of successes and failures, respectively.

**n** Vector of counts of trials; ignored if **x** is a matrix.

**p.adjust.method** Method for adjusting p values (see [p.adjust](#))

**...** Additional arguments to pass to `prop.test`

**Value**

Object of class "pairwise.htest"

**See Also**

[prop.test](#), [p.adjust](#)

**Examples**

```
smokers <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
pairwise.prop.test(smokers, patients)
```

---

pairwise.t.test *Pairwise t tests*

---

**Description**

Calculate pairwise comparisons between group levels with corrections for multiple testing

**Usage**

```
pairwise.t.test(x, g, p.adjust.method = p.adjust.methods,
               pool.sd = TRUE, ...)
```

**Arguments**

`x`                    Response vector  
`g`                     Grouping vector or factor  
`p.adjust.method`  
                          Method for adjusting p values (see [p.adjust](#))  
`pool.sd`                Switch to allow/disallow the use of a pooled SD  
`...`                    Additional arguments to pass to `t.test`

**Value**

Object of class "pairwise.htest"

**See Also**

[t.test](#), [p.adjust](#)

**Examples**

```
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
pairwise.t.test(Ozone, Month)
pairwise.t.test(Ozone, Month, p.adj = "bonf")
pairwise.t.test(Ozone, Month, pool.sd = FALSE)
detach()
```

---

`pairwise.table`            *Tabulate p values for pairwise comparisons*

---

**Description**

Creates table of p values for pairwise comparisons with corrections for multiple testing.

**Usage**

```
pairwise.table(compare.levels, level.names, p.adjust.method)
```

**Arguments**

`compare.levels`  
                          Function to compute (raw) p value given indices `i` and `j`  
`level.names`            Names of the group levels  
`p.adjust.method`  
                          Method for multiple testing adjustment

**Details**

Functions that do multiple group comparisons create separate `compare.levels` functions (assumed to be symmetrical in `i` and `j`) and passes them to this function.

**Value**

Table of p values in lower triangular form.

**See Also**

[pairwise.t.test](#), et al.

---

`pairwise.wilcox.test`

*Pairwise Wilcoxon rank sum tests*

---

**Description**

Calculate pairwise comparisons between group levels with corrections for multiple testing.

**Usage**

```
pairwise.wilcox.test(x, g, p.adjust.method = p.adjust.methods, ...)
```

**Arguments**

<code>x</code>	Response vector
<code>g</code>	Grouping vector or factor
<code>p.adjust.method</code>	Method for adjusting p values (see <a href="#">p.adjust</a> )
<code>...</code>	Additional arguments to pass to <a href="#">wilcox.test</a> .

**Value**

Object of class "pairwise.htest"

**See Also**

[wilcox.test](#), [p.adjust](#)

**Examples**

```
attach(airquality)
Month <- factor(Month, labels = month.abb[5:9])
## These give warnings because of ties :
pairwise.wilcox.test(Ozone, Month)
pairwise.wilcox.test(Ozone, Month, p.adj = "bonf")
detach()
```

**Description**

Plot method for objects of class "acf".

**Usage**

```
## S3 method for class 'acf':
plot(x, ci = 0.95, type = "h", xlab = "Lag", ylab = NULL,
     ylim = NULL, main = NULL,
     ci.col = "blue", ci.type = c("white", "ma"),
     max.mfrow = 6, ask = Npgs > 1 && dev.interactive(),
     mar = if(nser > 2) c(3,2,2,0.8) else par("mar"),
     oma = if(nser > 2) c(1,1.2,1,1) else par("oma"),
     mgp = if(nser > 2) c(1.5,0.6,0) else par("mgp"),
     xpd = par("xpd"), cex.main = if(nser > 2) 1 else par("cex.main"),
     verbose = getOption("verbose"),
     ...)
```

**Arguments**

<code>x</code>	an object of class "acf".
<code>ci</code>	coverage probability for confidence interval. Plotting of the confidence interval is suppressed if <code>ci</code> is zero or negative.
<code>type</code>	the type of plot to be drawn, default to histogram like vertical lines.
<code>xlab</code>	the x label of the plot.
<code>ylab</code>	the y label of the plot.
<code>ylim</code>	numeric of length 2 giving the y limits for the plot.
<code>main</code>	overall title for the plot.
<code>ci.col</code>	colour to plot the confidence interval lines.
<code>ci.type</code>	should the confidence limits assume a white noise input or for lag $k$ an $MA(k-1)$ input?
<code>max.mfrow</code>	positive integer; for multivariate <code>x</code> indicating how many rows and columns of plots should be put on one page, using <code>par(mfrow = c(m,m))</code> .
<code>ask</code>	logical; if TRUE, the user is asked before a new page is started.
<code>mar, oma, mgp, xpd, cex.main</code>	graphics parameters as in <code>par(*)</code> , by default adjusted to use smaller than default margins for multivariate <code>x</code> only. <code>xpd = NA</code> used to be the default for R version $\leq 1.4.0$ .
<code>verbose</code>	logical. Should R report extra information on progress?
<code>...</code>	graphics parameters to be passed to the plotting routines.

**Note**

The confidence interval plotted in `plot.acf` is based on an *uncorrelated* series and should be treated with appropriate caution. Using `ci.type = "ma"` may be less potentially misleading.

**See Also**

`acf` which calls `plot.acf` by default.

**Examples**

```
z4 <- ts(matrix(rnorm(400), 100, 4), start=c(1961, 1), frequency=12)
z7 <- ts(matrix(rnorm(700), 100, 7), start=c(1961, 1), frequency=12)
acf(z4)
acf(z7, max.mfrow = 7)# squeeze on 1 page
acf(z7) # multi-page
```

---

`plot.density`*Plot Method for Kernel Density Estimation*

---

**Description**

The plot method for density objects.

**Usage**

```
## S3 method for class 'density':
plot(x, main = NULL, xlab = NULL, ylab = "Density", type = "l",
      zero.line = TRUE, ...)
```

**Arguments**

<code>x</code>	a “density” object.
<code>main</code> , <code>xlab</code> , <code>ylab</code> , <code>type</code>	plotting parameters with useful defaults.
<code>...</code>	further plotting parameters.
<code>zero.line</code>	logical; if TRUE, add a base line at $y = 0$

**Value**

None.

**References****See Also**

`density`.

---

plot.HoltWinters    *Plot function for HoltWinters objects*

---

### Description

Produces a chart of the original time series along with the fitted values. Optionally, predicted values (and their confidence bounds) can also be plotted.

### Usage

```
## S3 method for class 'HoltWinters':
plot(x, predicted.values = NA, intervals = TRUE,
      separator = TRUE, col = 1, col.predicted = 2,
      col.intervals = 4, col.separator = 1, lty = 1,
      lty.predicted = 1, lty.intervals = 1, lty.separator = 3,
      ylab = "Observed / Fitted", main = "Holt-Winters filtering",
      ylim = NULL, ...)
```

### Arguments

x	Object of class "HoltWinters"
predicted.values	Predicted values as returned by predict.HoltWinters
intervals	If TRUE, the prediction intervals are plotted (default).
separator	If TRUE, a separating line between fitted and predicted values is plotted (default).
col, lty	Color/line type of original data (default: black solid).
col.predicted, lty.predicted	Color/line type of fitted and predicted values (default: red solid).
col.intervals, lty.intervals	Color/line type of prediction intervals (default: blue solid).
col.separator, lty.separator	Color/line type of observed/predicted values separator (default: black dashed).
ylab	Label of the y-axis.
main	Main title.
ylim	Limits of the y-axis. If NULL, the range is chosen such that the plot contains the original series, the fitted values, and the predicted values if any.
...	Other graphics parameters.

### Author(s)

David Meyer (David.Meyer@wu-wien.ac.at)

**References**

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, ONR Research Memorandum, Carnegie Institute 52.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

**See Also**

[HoltWinters](#), [predict.HoltWinters](#)

---

plot.isoreg                      *Plot Method for isoreg Objects*

---

**Description**

The `plot` method for R objects of class `isoreg`.

**Usage**

```
## S3 method for class 'isoreg':
plot(x, plot.type = c("single", "row.wise", "col.wise"),
     main = paste("Isotonic regression", deparse(x$call)),
     main2 = "Cumulative Data and Convex Minorant",
     xlab = "x0", ylab = "x$y",
     par.fit = list(col = "red", cex = 1.5, pch = 13, lwd = 1.5),
     mar = if (both) 0.1 + c(3.5, 2.5, 1, 1) else par("mar"),
     mgp = if (both) c(1.6, 0.7, 0) else par("mgp"),
     grid = length(x$x) < 12, ...)
```

**Arguments**

<code>x</code>	an <code>isoreg</code> object.
<code>plot.type</code>	character indicating which type of plot is desired. The first (default) only draws the data and the fit, where the others add a plot of the cumulative data and fit.
<code>main</code>	main title of plot, see <code>title</code> .
<code>main2</code>	title for second (cumulative) plot.
<code>xlab, ylab</code>	x- and y- axis annotation.
<code>par.fit</code>	a <code>list</code> of arguments (for <code>points</code> and <code>lines</code> ) for drawing the fit.
<code>mar, mgp</code>	graphical parameters, see <code>par</code> , mainly for the case of two plots.
<code>grid</code>	logical indicating if grid lines should be drawn. If true, <code>grid( )</code> is used for the first plot, where as vertical lines are drawn at “touching” points for the cumulative plot.
<code>...</code>	further arguments passed to and from methods.

**See Also**

[isoreg](#) for computation of isoreg objects.

**Examples**

```
example(isoreg) # for the examples there

## 'same' plot as above, "proving" that only ranks of 'x' are important
plot(isoreg(2^(1:9), c(1,0,4,3,3,5,4,2,0)), plot.t = "row", log = "x")

plot(ir3, plot.type = "row", ylab = "y3")
plot(isoreg(y3 - 4), plot.t="r", ylab = "y3 - 4")
plot(ir4, plot.type = "ro", ylab = "y4", xlab = "x = 1:n")

## experiment a bit with these (C-c C-j):
plot(isoreg(sample(9), y3), plot.type="row")
plot(isoreg(sample(9), y3), plot.type="col.wise")

plot(ir <- isoreg(sample(10), sample(10, replace = TRUE)), plot.t = "r")
```

---

plot.lm

*Plot Diagnostics for an lm Object*


---

**Description**

Four plots (selectable by `which`) are currently provided: a plot of residuals against fitted values, a Scale-Location plot of  $\sqrt{|residuals|}$  against fitted values, a Normal Q-Q plot, and a plot of Cook's distances versus row labels.

**Usage**

```
## S3 method for class 'lm':
plot(x, which = 1:4,
      caption = c("Residuals vs Fitted", "Normal Q-Q plot",
                  "Scale-Location plot", "Cook's distance plot"),
      panel = points,
      sub.caption = deparse(x$call), main = "",
      ask = prod(par("mfcol")) < length(which) && dev.interactive(),
      ...,
      id.n = 3, labels.id = names(residuals(x)), cex.id = 0.75)
```

**Arguments**

<code>x</code>	lm object, typically result of <code>lm</code> or <code>glm</code> .
<code>which</code>	If a subset of the plots is required, specify a subset of the numbers 1 : 4.
<code>caption</code>	Captions to appear above the plots
<code>panel</code>	Panel function. A useful alternative to <code>points</code> is <code>panel.smooth</code> .

sub.caption	common title—above figures if there are multiple; used as sub(s.title) otherwise.
main	title to each plot—in addition to the above caption.
ask	logical; if TRUE, the user is asked before each plot, see par(ask=.).
...	other parameters to be passed through to plotting functions.
id.n	number of points to be labelled in each plot, starting with the most extreme.
labels.id	vector of labels, from which the labels for extreme points will be chosen. NULL uses observation numbers.
cex.id	magnification of point labels.

### Details

sub.caption—by default the function call—is shown as a subtitle (under the x-axis title) on each plot when plots are on separate pages, or as a subtitle in the outer margin (if any) when there are multiple plots per page.

The “Scale-Location” plot, also called “Spread-Location” or “S-L” plot, takes the square root of the absolute residuals in order to diminish skewness ( $\sqrt{|E|}$  is much less skewed than  $|E|$  for Gaussian zero-mean  $E$ ).

This ‘S-L’ and the Q-Q plot use *standardized* residuals which have identical variance (under the hypothesis). They are given as  $R_i/(s \times \sqrt{1-h_{ii}})$  where  $h_{ii}$  are the diagonal entries of the hat matrix, `influence()` \$hat, see also `hat`.

### Author(s)

John Maindonald and Martin Maechler.

### References

- Belsley, D. A., Kuh, E. and Welsch, R. E. (1980) *Regression Diagnostics*. New York: Wiley.
- Cook, R. D. and Weisberg, S. (1982) *Residuals and Influence in Regression*. London: Chapman and Hall.
- Hinkley, D. V. (1975) On power transformations to symmetry. *Biometrika* **62**, 101–111.
- McCullagh, P. and Nelder, J. A. (1989) *Generalized Linear Models*. London: Chapman and Hall.

### See Also

`termplot`, `lm.influence`, `cooks.distance`.

### Examples

```
## Analysis of the life-cycle savings data
## given in Belsley, Kuh and Welsch.
plot(lm.SR <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings))

## 4 plots on 1 page; allow room for printing model formula in outer margin:
par(mfrow = c(2, 2), oma = c(0, 0, 2, 0))
plot(lm.SR)
```

```

plot(lm.SR, id.n = NULL)           # no id's
plot(lm.SR, id.n = 5, labels.id = NULL)# 5 id numbers

## Fit a smooth curve, where applicable:
plot(lm.SR, panel = panel.smooth)
## Gives a smoother curve
plot(lm.SR, panel = function(x,y) panel.smooth(x, y, span = 1))

par(mfrow=c(2,1))# same oma as above
plot(lm.SR, which = 1:2, sub.caption = "Saving Rates, n=50, p=5")

```

---

plot.ppr

*Plot Ridge Functions for Projection Pursuit Regression Fit*


---

### Description

Plot ridge functions for projection pursuit regression fit.

### Usage

```

## S3 method for class 'ppr':
plot(x, ask, type = "o", ...)

```

### Arguments

x	A fit of class "ppr" as produced by a call to ppr.
ask	the graphics parameter ask: see par for details. If set to TRUE will ask between the plot of each cross-section.
type	the type of line to draw
...	further graphical parameters

### Value

None

### Side Effects

A series of plots are drawn on the current graphical device, one for each term in the fit.

### See Also

[ppr](#), [par](#)

**Examples**

```
attach(rock)
areal <- area/10000; peril <- peri/10000
par(mfrow=c(3,2))# maybe: , pty="s")
rock.ppr <- ppr(log(perm) ~ areal + peril + shape,
               data = rock, nterms = 2, max.terms = 5)
plot(rock.ppr, main="ppr(log(perm)~ ., nterms=2, max.terms=5)")
plot(update(rock.ppr, bass=5), main = "update(..., bass = 5)")
plot(update(rock.ppr, sm.method="gcv", gcvpen=2),
     main = "update(..., sm.method=\"gcv\", gcvpen=2)")
detach()
```

---

plot.profile.nls    *Plot a profile.nls Object*

---

**Description**

Displays a series of plots of the profile t function and interpolated confidence intervals for the parameters in a nonlinear regression model that has been fit with `nls` and profiled with `profile.nls`.

**Usage**

```
## S3 method for class 'profile.nls':
plot(x, levels, conf= c(99, 95, 90, 80, 50)/100,
     nseg = 50, absVal =TRUE, ...)
```

**Arguments**

<code>x</code>	an object of class "profile.nls"
<code>levels</code>	levels, on the scale of the absolute value of a t statistic, at which to interpolate intervals. Usually <code>conf</code> is used instead of giving <code>levels</code> explicitly.
<code>conf</code>	a numeric vector of confidence levels for profile-based confidence intervals on the parameters. Defaults to <code>c(0.99, 0.95, 0.90, 0.80, 0.50)</code> .
<code>nseg</code>	an integer value giving the number of segments to use in the spline interpolation of the profile t curves. Defaults to 50.
<code>absVal</code>	a logical value indicating whether or not the plots should be on the scale of the absolute value of the profile t. Defaults to <code>TRUE</code> .
<code>...</code>	other arguments to the <code>plot</code> function can be passed here.

**Author(s)**

Douglas M. Bates and Saikat DebRoy

**References**

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley (chapter 6)

**See Also**

[nls](#), [profile](#), [profile.nls](#)

**Examples**

```
# obtain the fitted object
fml <- nls(demand ~ SSasymOrig( Time, A, lrc ), data = BOD)
# get the profile for the fitted model
pr1 <- profile( fml )
opar <- par(mfrow = c(2,2), oma = c(1.1, 0, 1.1, 0), las = 1)
plot(pr1, conf = c(95, 90, 80, 50)/100)
plot(pr1, conf = c(95, 90, 80, 50)/100, absVal = FALSE)
mtext("Confidence intervals based on the profile sum of squares",
      side = 3, outer = TRUE)
mtext("BOD data - confidence levels of 50%, 80%, 90% and 95%",
      side = 1, outer = TRUE)
par(opar)
```

---

plot.spec

*Plotting Spectral Densities*

---

**Description**

Plotting method for objects of class "spec". For multivariate time series it plots the marginal spectra of the series or pairs plots of the coherency and phase of the cross-spectra.

**Usage**

```
## S3 method for class 'spec':
plot(x, add = FALSE, ci = 0.95, log = c("yes", "dB", "no"),
     xlab = "frequency", ylab, type = "l", ci.col = "blue",
     main = NULL, sub = NULL,
     plot.type = c("marginal", "coherency", "phase"),
     ci.lty = 3, ...)
```

**Arguments**

x	an object of class "spec".
add	logical. If TRUE, add to already existing plot.
ci	Coverage probability for confidence interval. Plotting of the confidence bar is omitted unless ci is strictly positive.
log	If "dB", plot on log10 (decibel) scale (as S-PLUS), otherwise use conventional log scale or linear scale. Logical values are also accepted. The default is "yes" unless options(ts.S.compat = TRUE) has been set, when it is "dB".
xlab	the x label of the plot.
ylab	the y label of the plot.

type	the type of plot to be drawn, defaults to lines.
ci.col	Colour for plotting confidence bar or confidence intervals for coherency and phase.
main	overall title for the plot.
sub	a sub title for the plot.
plot.type	For multivariate time series, the type of plot required. Only the first character is needed.
ci.lty	line type for confidence intervals for coherency and phase.
...	Further graphical parameters.

**See Also**

[spectrum](#)

---

plot.stepfun      *Plot Step Functions*

---

**Description**

Method of the generic [plot](#) for [stepfun](#) objects and utility for plotting piecewise constant functions.

**Usage**

```
## S3 method for class 'stepfun':
plot(x, xval, xlim, ylim,
     xlab = "x", ylab = "f(x)", main = NULL,
     add = FALSE, verticals = TRUE, do.points = TRUE,
     pch = par("pch"),
     col.points = par("col"), cex.points = par("cex"),
     col.hor = par("col"), col.vert = par("col"),
     lty = par("lty"), lwd = par("lwd"), ...)

## S3 method for class 'stepfun':
lines(x, ...)
```

**Arguments**

x	an R object inheriting from "stepfun".
xval	numeric vector of abscissa values at which to evaluate x. Defaults to <a href="#">knots</a> (x) restricted to xlim.
xlim,ylim	numeric(2) each; range of x or y values to use. Both have sensible defaults.
xlab,ylab	labels of x and y axis.
main	main title.

add	logical; if TRUE only <i>add</i> to an existing plot.
verticals	logical; if TRUE, draw vertical lines at steps.
do.points	logical; if true, also draw points at the (xlim restricted) knot locations.
pch	character; point character if do.points.
col.points	character or integer code; color of points if do.points.
cex.points	numeric; character expansion factor if do.points.
col.hor	color of horizontal lines.
col.vert	color of vertical lines.
lty, lwd	line type and thickness for all lines.
...	further arguments of <code>plot( . )</code> , or if (add) <code>segments( . )</code> .

**Value**

A list with two components

t	abscissa (x) values, including the two outermost ones.
y	y values ‘in between’ the t [ ].

**Author(s)**

Martin Maechler <maechler@stat.math.ethz.ch>, 1990, 1993; ported to R, 1997.

**See Also**

[ecdf](#) for empirical distribution functions as special step functions, [approxfun](#) and [splinefun](#).

**Examples**

```

y0 <- c(1,2,4,3)
sfun0 <- stepfun(1:3, y0, f = 0)
sfun.2 <- stepfun(1:3, y0, f = .2)
sfun1 <- stepfun(1:3, y0, right = TRUE)

tt <- seq(0,3, by=0.1)
op <- par(mfrow=c(2,2))
plot(sfun0); plot(sfun0, xval=tt, add=TRUE, col.h="bisque")
plot(sfun.2);plot(sfun.2,xval=tt, add=TRUE, col.h="orange")
plot(sfun1);lines(sfun1, xval=tt, col.h="coral")
##-- This is revealing :
plot(sfun0, verticals= FALSE,
      main = "stepfun(x, y0, f=f) for f = 0, .2, 1")
for(i in 1:3)
  lines(list(sfun0,sfun.2,stepfun(1:3,y0,f = 1))[[i]], col.h=i, col.v=i)
legend(2.5, 1.9, paste("f =", c(0,0.2,1)), col=1:3, lty=1, y.inter=1); par(op)

# Extend and/or restrict 'viewport':
plot(sfun0, xlim = c(0,5), ylim = c(0, 3.5),
      main = "plot(stepfun(*), xlim= . , ylim = .)")

```

```
##-- this works too (automatic call to ecdf(.)):
plot.stepfun(rt(50, df=3), col.vert = "gray20")
```

---

plot.ts

*Plotting Time-Series Objects*


---

## Description

Plotting method for objects inheriting from class "ts".

## Usage

```
## S3 method for class 'ts':
plot(x, y = NULL, plot.type = c("multiple", "single"),
      xy.labels, xy.lines, panel = lines, nc, yax.flip = FALSE,
      mar.multi = c(0, 5.1, 0, if(yax.flip) 5.1 else 2.1),
      oma.multi = c(6, 0, 5, 0), axes = TRUE, ...)

## S3 method for class 'ts':
lines(x, ...)
```

## Arguments

<code>x, y</code>	time series objects, usually inheriting from class "ts".
<code>plot.type</code>	for multivariate time series, should the series be plotted separately (with a common time axis) or on a single plot?
<code>xy.labels</code>	logical, indicating if <code>text()</code> labels should be used for an x-y plot, <i>or</i> character, supplying a vector of labels to be used. The default is to label for up to 150 points, and not for more.
<code>xy.lines</code>	logical, indicating if <code>lines</code> should be drawn for an x-y plot. Defaults to the value of <code>xy.labels</code> if that is logical, otherwise to <code>TRUE</code> .
<code>panel</code>	a function( <code>x, col, bg, pch, type, ...</code> ) which gives the action to be carried out in each panel of the display for <code>plot.type="multiple"</code> . The default is <code>lines</code> .
<code>nc</code>	the number of columns to use when <code>type="multiple"</code> . Defaults to 1 for up to 4 series, otherwise to 2.
<code>yax.flip</code>	logical indicating if the y-axis (ticks and numbering) should flip from side 2 (left) to 4 (right) from series to series when <code>type="multiple"</code> .
<code>mar.multi, oma.multi</code>	the (default) <code>par</code> settings for <code>plot.type="multiple"</code> . Modify with care!
<code>axes</code>	logical indicating if x- and y- axes should be drawn.
<code>...</code>	additional graphical arguments, see <code>plot</code> , <code>plot.default</code> and <code>par</code> .

**Details**

If `y` is missing, this function creates a time series plot, for multivariate series of one of two kinds depending on `plot.type`.

If `y` is present, both `x` and `y` must be univariate, and a “scatter” plot  $y \sim x$  will be drawn, enhanced by using `text` if `xy.labels` is `TRUE` or character, and `lines` if `xy.lines` is `TRUE`.

**See Also**

`ts` for basic time series construction and access functionality.

**Examples**

```
## Multivariate
z <- ts(matrix(rt(200 * 8, df = 3), 200, 8), start=c(1961, 1), frequency=12)
plot(z, yax.flip = TRUE)
plot(z, axes= FALSE, ann= FALSE, frame.plot= TRUE,
      mar.mult= c(0,0,0,0), oma.mult= c(1,1,5,1))
title("plot(ts(..), axes=FALSE, ann=FALSE, frame.plot=TRUE, mar..., oma...)")

z <- window(z[,1:3], end = c(1969,12))
plot(z, type = "b")      # multiple
plot(z, plot.type="single", lty=1:3, col=4:2)

## A phase plot:
plot(nhtemp, c(nhtemp[-1], NA), cex = .8, col="blue",
      main = "Lag plot of New Haven temperatures")
## a clearer way to do this would be
## Not run:
plot(nhtemp, lag(nhtemp, 1), cex = .8, col="blue",
      main = "Lag plot of New Haven temperatures")
## End(Not run)

## xy.lines and xy.labels are FALSE for large series:
plot(lag(sunspots, 1), sunspots, pch = ".")

SMI <- EuStockMarkets[, "SMI"]
plot(lag(SMI, 1), SMI, pch = ".")
plot(lag(SMI, 20), SMI, pch = ".", log = "xy",
      main = "4 weeks lagged SMI stocks -- log scale", xy.lines= TRUE)
```

**Description**

Density, distribution function, quantile function and random generation for the Poisson distribution with parameter `lambda`.

**Usage**

```
dpois(x, lambda, log = FALSE)
ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)
qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)
rpois(n, lambda)
```

**Arguments**

`x` vector of (non-negative integer) quantiles.  
`q` vector of quantiles.  
`p` vector of probabilities.  
`n` number of random values to return.  
`lambda` vector of positive means.  
`log, log.p` logical; if TRUE, probabilities `p` are given as  $\log(p)$ .  
`lower.tail` logical; if TRUE (default), probabilities are  $P[X \leq x]$ , otherwise,  $P[X > x]$ .

**Details**

The Poisson distribution has density

$$p(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

for  $x = 0, 1, 2, \dots$ . The mean and variance are  $E(X) = Var(X) = \lambda$ .

If an element of `x` is not integer, the result of `dpois` is zero, with a warning.  $p(x)$  is computed using Loader's algorithm, see the reference in [dbinom](#).

The quantile is left continuous: `qgeom(q, prob)` is the largest integer  $x$  such that  $P(X \leq x) < q$ .

Setting `lower.tail = FALSE` allows to get much more precise results when the default, `lower.tail = TRUE` would return 1, see the example below.

**Value**

`dpois` gives the (log) density, `ppois` gives the (log) distribution function, `qpois` gives the quantile function, and `rpois` generates random deviates.

**See Also**

[dbinom](#) for the binomial and [dnbinom](#) for the negative binomial distribution.

**Examples**

```
-log(dpois(0:7, lambda=1) * gamma(1+ 0:7)) # == 1
Ni <- rpois(50, lam= 4); table(factor(Ni, 0:max(Ni)))

1 - ppois(10*(15:25), lambda=100) # becomes 0 (cancellation)
  ppois(10*(15:25), lambda=100, lower=FALSE) # no cancellation
```

```

par(mfrow = c(2, 1))
x <- seq(-0.01, 5, 0.01)
plot(x, ppois(x, 1), type="s", ylab="F(x)", main="Poisson(1) CDF")
plot(x, pbinom(x, 100, 0.01), type="s", ylab="F(x)",
      main="Binomial(100, 0.01) CDF")

```

poly

*Compute Orthogonal Polynomials***Description**

Returns or evaluates orthogonal polynomials of degree 1 to degree over the specified set of points  $x$ . These are all orthogonal to the constant polynomial of degree 0.

**Usage**

```

poly(x, ..., degree = 1, coefs = NULL)
polym(..., degree = 1)

## S3 method for class 'poly':
predict(object, newdata, ...)

```

**Arguments**

<code>x</code> , <code>newdata</code>	a numeric vector at which to evaluate the polynomial. <code>x</code> can also be a matrix.
<code>degree</code>	the degree of the polynomial
<code>coefs</code>	for prediction, coefficients from a previous fit.
<code>object</code>	an object inheriting from class "poly", normally the result of a call to <code>poly</code> with a single vector argument.
<code>...</code>	<code>poly</code> , <code>polym</code> : further vectors. <code>predict.poly</code> : arguments to be passed to or from other methods.

**Details**

Although formally `degree` should be named (as it follows `...`), an unnamed second argument of length 1 will be interpreted as the degree.

The orthogonal polynomial is summarized by the coefficients, which can be used to evaluate it via the three-term recursion given in Kennedy & Gentle (1980, pp. 343-4), and used in the “predict” part of the code.

**Value**

For `poly` with a single vector argument:

A matrix with rows corresponding to points in `x` and columns corresponding to the degree, with attributes "degree" specifying the degrees of the columns and "coefs" which contains the centering and normalization constants used in constructing the orthogonal polynomials. The matrix is given class `c("poly", "matrix")` as from R 1.5.0.

Other cases of `poly` and `polym`, and `predict.poly`: a matrix.

**Note**

This routine is intended for statistical purposes such as `contr.poly`: it does not attempt to orthogonalize to machine accuracy.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.  
 Kennedy, W. J. Jr and Gentle, J. E. (1980) *Statistical Computing* Marcel Dekker.

**See Also**

`contr.poly`

**Examples**

```
(z <- poly(1:10, 3))
predict(z, seq(2, 4, 0.5))
poly(seq(4, 6, 0.5), 3, coefs = attr(z, "coefs"))

polym(1:4, c(1, 4:6), degree=3) # or just poly()
poly(cbind(1:4, c(1, 4:6)), degree=3)
```

---

power

*Create a Power Link Object*

---

**Description**

Creates a link object based on the link function  $\eta = \mu^\lambda$ .

**Usage**

```
power(lambda = 1)
```

**Arguments**

lambda            a real number.

**Details**

If lambda is non-negative, it is taken as zero, and the log link is obtained. The default lambda = 1 gives the identity link.

**Value**

A list with components `linkfun`, `linkinv`, `mu.eta`, and `valideta`. See [make.link](#) for information on their meaning.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

[make.link.family](#)

To raise a number to a power, see [Arithmetic](#).

To calculate the power of a test, see various functions in the **stats** package, e.g., [power.t.test](#).

**Examples**

```
power()
quasi(link=power(1/3))[c("linkfun", "linkinv")]
```

---

`power.anova.test`    *Power calculations for balanced one-way analysis of variance tests*

---

**Description**

Compute power of test or determine parameters to obtain target power.

**Usage**

```
power.anova.test(groups = NULL, n = NULL, between.var = NULL,
                 within.var = NULL, sig.level = 0.05, power = NULL)
```

**Arguments**

<code>groups</code>	Number of groups
<code>n</code>	Number of observations (per group)
<code>between.var</code>	Between group variance
<code>within.var</code>	Within group variance
<code>sig.level</code>	Significance level (Type I error probability)
<code>power</code>	Power of test (1 minus Type II error probability)

**Details**

Exactly one of the parameters `groups`, `n`, `between.var`, `power`, `within.var`, and `sig.level` must be passed as `NULL`, and that parameter is determined from the others. Notice that `sig.level` has non-`NULL` default so `NULL` must be explicitly passed if you want it computed.

**Value**

Object of class `"power.htest"`, a list of the arguments (including the computed one) augmented with `method` and `note` elements.

**Note**

uniroot is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given.

**Author(s)**

Claus Ekstrøm

**See Also**

[anova](#), [lm](#), [uniroot](#)

**Examples**

```
power.anova.test(groups=4, n=5, between.var=1, within.var=3)
# Power = 0.3535594

power.anova.test(groups=4, between.var=1, within.var=3,
                 power=.80)
# n = 11.92613

## Assume we have prior knowledge of the group means:
groupmeans <- c(120, 130, 140, 150)
power.anova.test(groups = length(groupmeans),
                 between.var=var(groupmeans),
                 within.var=500, power=.90) # n = 15.18834
```

---

power.prop.test      *Power calculations two sample test for of proportions*

---

**Description**

Compute power of test, or determine parameters to obtain target power.

**Usage**

```
power.prop.test(n = NULL, p1 = NULL, p2 = NULL, sig.level = 0.05,
               power = NULL,
               alternative = c("two.sided", "one.sided"),
               strict = FALSE)
```

**Arguments**

n	Number of observations (per group)
p1	probability in one group
p2	probability in other group

<code>sig.level</code>	Significance level (Type I error probability)
<code>power</code>	Power of test (1 minus Type II error probability)
<code>alternative</code>	One- or two-sided test
<code>strict</code>	Use strict interpretation in two-sided case

### Details

Exactly one of the parameters `n`, `p1`, `p2`, `power`, and `sig.level` must be passed as `NULL`, and that parameter is determined from the others. Notice that `sig.level` has a non-`NULL` default so `NULL` must be explicitly passed if you want it computed.

If `strict = TRUE` is used, the power will include the probability of rejection in the opposite direction of the true effect, in the two-sided case. Without this the power will be half the significance level if the true difference is zero.

### Value

Object of class `"power.htest"`, a list of the arguments (including the computed one) augmented with `method` and `note` elements.

### Note

`uniroot` is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given. If one of them is computed  $p_1 < p_2$  will hold, although this is not enforced when both are specified.

### Author(s)

Peter Dalgaard. Based on previous work by Claus Ekstrøm

### See Also

[prop.test](#), [uniroot](#)

### Examples

```
power.prop.test(n = 50, p1 = .50, p2 = .75)
power.prop.test(p1 = .50, p2 = .75, power = .90)
power.prop.test(n = 50, p1 = .5, power = .90)
```

---

power.t.test            *Power calculations for one and two sample t tests*

---

### Description

Compute power of test, or determine parameters to obtain target power.

### Usage

```
power.t.test(n = NULL, delta = NULL, sd = 1, sig.level = 0.05,
             power = NULL,
             type = c("two.sample", "one.sample", "paired"),
             alternative = c("two.sided", "one.sided"),
             strict = FALSE)
```

### Arguments

n	Number of observations (per group)
delta	True difference in means
sd	Standard deviation
sig.level	Significance level (Type I error probability)
power	Power of test (1 minus Type II error probability)
type	Type of t test
alternative	One- or two-sided test
strict	Use strict interpretation in two-sided case

### Details

Exactly one of the parameters `n`, `delta`, `power`, `sd`, and `sig.level` must be passed as `NULL`, and that parameter is determined from the others. Notice that the last two have non-`NULL` defaults so `NULL` must be explicitly passed if you want to compute them.

If `strict = TRUE` is used, the power will include the probability of rejection in the opposite direction of the true effect, in the two-sided case. Without this the power will be half the significance level if the true difference is zero.

### Value

Object of class "power.htest", a list of the arguments (including the computed one) augmented with `method` and `note` elements.

### Note

`uniroot` is used to solve power equation for unknowns, so you may see errors from it, notably about inability to bracket the root when invalid arguments are given.

**Author(s)**

Peter Dalgaard. Based on previous work by Claus Ekstrøm

**See Also**

`t.test`, `uniroot`

**Examples**

```
power.t.test(n = 20, delta = 1)
power.t.test(power = .90, delta = 1)
power.t.test(power = .90, delta = 1, alt = "one.sided")
```

---

PP.test

*Phillips-Perron Test for Unit Roots*

---

**Description**

Computes the Phillips-Perron test for the null hypothesis that  $x$  has a unit root against a stationary alternative.

**Usage**

```
PP.test(x, lshort = TRUE)
```

**Arguments**

<code>x</code>	a numeric vector or univariate time series.
<code>lshort</code>	a logical indicating whether the short or long version of the truncation lag parameter is used.

**Details**

The general regression equation which incorporates a constant and a linear trend is used and the corrected t-statistic for a first order autoregressive coefficient equals one is computed. To estimate  $\sigma^2$  the Newey-West estimator is used. If `lshort` is TRUE, then the truncation lag parameter is set to `trunc(4*(n/100)^0.25)`, otherwise `trunc(12*(n/100)^0.25)` is used. The  $p$ -values are interpolated from Table 4.2, page 103 of Banerjee *et al.* (1993).

Missing values are not handled.

**Value**

A list with class "htest" containing the following components:

<code>statistic</code>	the value of the test statistic.
<code>parameter</code>	the truncation lag parameter.
<code>p.value</code>	the $p$ -value of the test.
<code>method</code>	a character string indicating what type of test was performed.
<code>data.name</code>	a character string giving the name of the data.

**Author(s)**

A. Trapletti

**References**

A. Banerjee, J. J. Dolado, J. W. Galbraith, and D. F. Hendry (1993) *Cointegration, Error Correction, and the Econometric Analysis of Non-Stationary Data*, Oxford University Press, Oxford.

P. Perron (1988) Trends and random walks in macroeconomic time series. *Journal of Economic Dynamics and Control* **12**, 297–332.

**Examples**

```
x <- rnorm(1000)
PP.test(x)
y <- cumsum(x) # has unit root
PP.test(y)
```

---

ppoints

*Ordinates for Probability Plotting*


---

**Description**

Generates the sequence of “probability” points  $(1:m - a) / (m + (1-a) - a)$  where  $m$  is either  $n$ , if  $\text{length}(n) == 1$ , or  $\text{length}(n)$ .

**Usage**

```
ppoints(n, a = ifelse(n <= 10, 3/8, 1/2))
```

**Arguments**

$n$  either the number of points generate or a vector of observations.  
 $a$  the offset fraction to be used; typically in  $(0, 1)$ .

**Details**

If  $0 < a < 1$ , the resulting values are within  $(0, 1)$  (excluding boundaries). In any case, the resulting sequence is symmetric in  $[0, 1]$ , i.e.,  $p + \text{rev}(p) == 1$ .

`ppoints()` is used in `qqplot` and `qqnorm` to generate the set of probabilities at which to evaluate the inverse distribution.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[qqplot](#), [qqnorm](#).

**Examples**

```
ppoints(4) # the same as ppoints(1:4)
ppoints(10)
ppoints(10, a=1/2)
```

---

 ppr

*Projection Pursuit Regression*


---

**Description**

Fit a projection pursuit regression model.

**Usage**

```
ppr(x, ...)

## S3 method for class 'formula':
ppr(formula, data, weights, subset, na.action,
     contrasts = NULL, ..., model = FALSE)

## Default S3 method:
ppr(x, y, weights = rep(1,n),
     ww = rep(1,q), nterms, max.terms = nterms, optlevel = 2,
     sm.method = c("supsmu", "spline", "gcv spline"),
     bass = 0, span = 0, df = 5, gcvpen = 1, ...)
```

**Arguments**

<code>formula</code>	a formula specifying one or more numeric response variables and the explanatory variables.
<code>x</code>	numeric matrix of explanatory variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
<code>y</code>	numeric matrix of response variables. Rows represent observations, and columns represent variables. Missing values are not accepted.
<code>nterms</code>	number of terms to include in the final model.
<code>data</code>	data frame from which variables specified in <code>formula</code> are preferentially to be taken.
<code>weights</code>	a vector of weights $w_i$ for each <i>case</i> .
<code>ww</code>	a vector of weights for each <i>response</i> , so the fit criterion is the sum over case $i$ and responses $j$ of $w_i ww_j (y_{ij} - fit_{ij})^2$ divided by the sum of $w_i$ .

<code>subset</code>	an index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
<code>na.action</code>	a function to specify the action to be taken if <code>NA</code> s are found. The default action is given by <code>getOption("na.action")</code> . (NOTE: If given, this argument must be named.)
<code>contrasts</code>	the contrasts to be used when any factor explanatory variables are coded.
<code>max.terms</code>	maximum number of terms to choose from when building the model.
<code>optlevel</code>	integer from 0 to 3 which determines the thoroughness of an optimization routine in the SMART program. See the <b>Details</b> section.
<code>sm.method</code>	the method used for smoothing the ridge functions. The default is to use Friedman's super smoother <code>supsmu</code> . The alternatives are to use the smoothing spline code underlying <code>smooth.spline</code> , either with a specified (equivalent) degrees of freedom for each ridge functions, or to allow the smoothness to be chosen by GCV.
<code>bass</code>	super smoother bass tone control used with automatic span selection (see <code>supsmu</code> ); the range of values is 0 to 10, with larger values resulting in increased smoothing.
<code>span</code>	super smoother span control (see <code>supsmu</code> ). The default, 0, results in automatic span selection by local cross validation. <code>span</code> can also take a value in <code>(0, 1]</code> .
<code>df</code>	if <code>sm.method</code> is "spline" specifies the smoothness of each ridge term via the requested equivalent degrees of freedom.
<code>gcvpen</code>	if <code>sm.method</code> is "gcv spline" this is the penalty used in the GCV selection for each degree of freedom used.
<code>...</code>	arguments to be passed to or from other methods.
<code>model</code>	logical. If true, the model frame is returned.

### Details

The basic method is given by Friedman (1984), and is essentially the same code used by S-PLUS's `ppreg`. This code is extremely sensitive to the compiler used.

The algorithm first adds up to `max.terms` ridge terms one at a time; it will use less if it is unable to find a term to add that makes sufficient difference. It then removes the least "*important*" term at each step until `nterm` terms are left.

The levels of optimization (argument `optlevel`) differ in how thoroughly the models are refitted during this process. At level 0 the existing ridge terms are not refitted. At level 1 the projection directions are not refitted, but the ridge functions and the regression coefficients are. Levels 2 and 3 refit all the terms and are equivalent for one response; level 3 is more careful to re-balance the contributions from each regressor at each step and so is a little less likely to converge to a saddle point of the sum of squares criterion.

### Value

A list with the following components, many of which are for use by the method functions.

<code>call</code>	the matched call
-------------------	------------------

p	the number of explanatory variables (after any coding)
q	the number of response variables
mu	the argument <code>nterms</code>
ml	the argument <code>max.terms</code>
gof	the overall residual (weighted) sum of squares for the selected model
gofn	the overall residual (weighted) sum of squares against the number of terms, up to <code>max.terms</code> . Will be invalid (and zero) for less than <code>nterms</code> .
df	the argument <code>df</code>
edf	if <code>sm.method</code> is "spline" or "gcv spline" the equivalent number of degrees of freedom for each ridge term used.
xnames	the names of the explanatory variables
yname	the names of the response variables
alpha	a matrix of the projection directions, with a column for each ridge term
beta	a matrix of the coefficients applied for each response to the ridge terms: the rows are the responses and the columns the ridge terms
yb	the weighted means of each response
ys	the overall scale factor used: internally the responses are divided by <code>ys</code> to have unit total weighted sum of squares.
fitted.values	the fitted values, as a matrix if <code>q &gt; 1</code> .
residuals	the residuals, as a matrix if <code>q &gt; 1</code> .
smod	internal work array, which includes the ridge functions evaluated at the training set points.
model	(only if <code>model=TRUE</code> ) the model frame.

## References

- Friedman, J. H. and Stuetzle, W. (1981) Projection pursuit regression. *Journal of the American Statistical Association*, **76**, 817–823.
- Friedman, J. H. (1984) SMART User's Guide. Laboratory for Computational Statistics, Stanford University Technical Report No. 1.
- Venables, W. N. & Ripley, B. D. (2002) *Modern Applied Statistics with S*. Springer.

## See Also

[plot.ppr](#), [supsmu](#), [smooth.spline](#)

## Examples

```
# Note: your numerical values may differ
attach(rock)
areal <- area/10000; peril <- peri/10000
rock.ppr <- ppr(log(perm) ~ areal + peril + shape,
               data = rock, nterms = 2, max.terms = 5)
```

```

rock.ppr
# Call:
# ppr.formula(formula = log(perm) ~ areal + peril + shape, data = rock,
#   nterms = 2, max.terms = 5)
#
# Goodness of fit:
# 2 terms 3 terms 4 terms 5 terms
# 8.737806 5.289517 4.745799 4.490378

summary(rock.ppr)
# ..... (same as above)
# .....
#
# Projection direction vectors:
#   term 1      term 2
# areal  0.34357179  0.37071027
# peril -0.93781471 -0.61923542
# shape  0.04961846  0.69218595
#
# Coefficients of ridge terms:
#   term 1      term 2
# 1.6079271 0.5460971

par(mfrow=c(3,2))# maybe: , pty="s")
plot(rock.ppr, main="ppr(log(perm)~ ., nterms=2, max.terms=5)")
plot(update(rock.ppr, bass=5), main = "update(..., bass = 5)")
plot(update(rock.ppr, sm.method="gcv", gcvpen=2),
      main = "update(..., sm.method=\"gcv\", gcvpen=2)")
cbind(perm=rock$perm, prediction=round(exp(predict(rock.ppr)), 1))
detach()

```

---

prcomp

*Principal Components Analysis*


---

### Description

Performs a principal components analysis on the given data matrix and returns the results as an object of class `prcomp`.

### Usage

```

prcomp(x, ...)

## S3 method for class 'formula':
prcomp(formula, data = NULL, subset, na.action, ...)

## Default S3 method:
prcomp(x, retx = TRUE, center = TRUE, scale. = FALSE, tol = NULL, ...)

```

**Arguments**

<code>formula</code>	a formula with no response variable.
<code>data</code>	an optional data frame containing the variables in the formula <code>formula</code> . By default the variables are taken from <code>environment(formula)</code> .
<code>subset</code>	an optional vector used to select rows (observations) of the data matrix <code>x</code> .
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. The default is set by the <code>na.action</code> setting of <code>options</code> , and is <code>na.fail</code> if that is unset. The “factory-fresh” default is <code>na.omit</code> .
<code>...</code>	arguments passed to or from other methods. If <code>x</code> is a formula one might specify <code>scale.</code> or <code>tol</code> .
<code>x</code>	a numeric or complex matrix (or data frame) which provides the data for the principal components analysis.
<code>retx</code>	a logical value indicating whether the rotated variables should be returned.
<code>center</code>	a logical value indicating whether the variables should be shifted to be zero centered. Alternately, a vector of length equal the number of columns of <code>x</code> can be supplied. The value is passed to <code>scale</code> .
<code>scale.</code>	a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place. The default is <code>FALSE</code> for consistency with <code>S</code> , but in general scaling is advisable. Alternatively, a vector of length equal the number of columns of <code>x</code> can be supplied. The value is passed to <code>scale</code> .
<code>tol</code>	a value indicating the magnitude below which components should be omitted. (Components are omitted if their standard deviations are less than or equal to <code>tol</code> times the standard deviation of the first component.) With the default null setting, no components are omitted. Other settings for <code>tol</code> could be <code>tol = 0</code> or <code>tol = sqrt(.Machine\$double.eps)</code> , which would omit essentially constant components.

**Details**

The calculation is done by a singular value decomposition of the (centered and possibly scaled) data matrix, not by using `eigen` on the covariance matrix. This is generally the preferred method for numerical accuracy. The `print` method for these objects prints the results in a nice format and the `plot` method produces a scree plot.

**Value**

`prcomp` returns an list with class `"prcomp"` containing the following components:

<code>sdev</code>	the standard deviations of the principal components (i.e., the square roots of the eigenvalues of the covariance/correlation matrix, though the calculation is actually done with the singular values of the data matrix).
<code>rotation</code>	the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors). The function <code>princomp</code> returns this in the element <code>loadings</code> .
<code>x</code>	if <code>retx</code> is true the value of the rotated data (the centred (and scaled if requested) data multiplied by the <code>rotation</code> matrix) is returned.
<code>center, scale</code>	the centering and scaling used, or <code>FALSE</code> .

**Note**

The signs of the columns of the rotation matrix are arbitrary, and so may differ between different programs for PCA, and even between different builds of R.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Mardia, K. V., J. T. Kent, and J. M. Bibby (1979) *Multivariate Analysis*, London: Academic Press.

Venables, W. N. and B. D. Ripley (2002) *Modern Applied Statistics with S*, Springer-Verlag.

**See Also**

[biplot.prcomp](#), [princomp](#), [cor](#), [cov](#), [svd](#), [eigen](#).

**Examples**

```
## the variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
prcomp(USArrests) # inappropriate
prcomp(USArrests, scale = TRUE)
prcomp(~ Murder + Assault + Rape, data = USArrests, scale = TRUE)
plot(prcomp(USArrests))
summary(prcomp(USArrests, scale = TRUE))
biplot(prcomp(USArrests, scale = TRUE))
```

---

predict

*Model Predictions*

---

**Description**

`predict` is a generic function for predictions from the results of various model fitting functions. The function invokes particular *methods* which depend on the `class` of the first argument.

**Usage**

```
predict (object, ...)
```

**Arguments**

`object` a model object for which prediction is desired.  
`...` additional arguments affecting the predictions produced.

**Details**

Most prediction methods which similar to fitting linear models have an argument `newdata` specifying the first place to look for explanatory variables to be used for prediction. Some considerable attempts are made to match up the columns in `newdata` to those used for fitting, for example that they are of comparable types and that any factors have the same level set in the same order (or can be transformed to be so).

Time series prediction methods in package **stats** have an argument `n.ahead` specifying how many time steps ahead to predict.

Many methods have a logical argument `se.fit` saying if standard errors are to returned.

**Value**

The form of the value returned by `predict` depends on the class of its argument. See the documentation of the particular methods for details of what is produced by that method.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

[predict.glm](#), [predict.lm](#), [predict.loess](#), [predict.nls](#), [predict.poly](#), [predict.princomp](#), [predict.smooth.spline](#).

For time-series prediction, [predict.ar](#), [predict.Arima](#), [predict.arima0](#), [predict.HoltWinters](#), [predict.StructTS](#).

**Examples**

```
## All the "predict" methods found
## NB most of the methods in the standard packages are hidden.
for(fn in methods("predict"))
  try({
    f <- eval(substitute(getAnywhere(fn)$objs[[1]]), list(fn = fn))
    cat(fn, ":\n\t", deparse(args(f)), "\n")
  }, silent = TRUE)
```

---

<code>predict.Arima</code>	<i>Forecast from ARIMA fits</i>
----------------------------	---------------------------------

---

**Description**

Forecast from models fitted by [arima](#).

**Usage**

```
## S3 method for class 'Arima':
predict(object, n.ahead = 1, newxreg = NULL,
        se.fit = TRUE, ...)
```

**Arguments**

<code>object</code>	The result of an <code>arima</code> fit.
<code>n.ahead</code>	The number of steps ahead for which prediction is required.
<code>newxreg</code>	New values of <code>xreg</code> to be used for prediction. Must have at least <code>n.ahead</code> rows.
<code>se.fit</code>	Logical: should standard errors of prediction be returned?
<code>...</code>	arguments passed to or from other methods.

**Details**

Finite-history prediction is used, via [KalmanForecast](#). This is only statistically efficient if the MA part of the fit is invertible, so `predict.Arima` will give a warning for non-invertible MA models.

The standard errors of prediction exclude the uncertainty in the estimation of the ARMA model and the regression coefficients. According to Harvey (1993, pp. 58–9) the effect is small.

**Value**

A time series of predictions, or if `se.fit = TRUE`, a list with components `pred`, the predictions, and `se`, the estimated standard errors. Both components are time series.

**References**

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press.

Harvey, A. C. and McKenzie, C. R. (1982) Algorithm AS182. An algorithm for finite sample prediction from ARIMA processes. *Applied Statistics* **31**, 180–187.

Harvey, A. C. (1993) *Time Series Models*, 2nd Edition, Harvester Wheatsheaf, sections 3.3 and 4.4.

**See Also**

[arima](#)

**Examples**

```
predict(arima(lh, order = c(3,0,0)), n.ahead = 12)

(fit <- arima(USAccDeaths, order = c(0,1,1),
             seasonal = list(order=c(0,1,1))))
predict(fit, n.ahead = 6)
```

---

predict.glm                      *Predict Method for GLM Fits*

---

### Description

Obtains predictions and optionally estimates standard errors of those predictions from a fitted generalized linear model object.

### Usage

```
## S3 method for class 'glm':
predict(object, newdata = NULL,
        type = c("link", "response", "terms"),
        se.fit = FALSE, dispersion = NULL, terms = NULL,
        na.action = na.pass, ...)
```

### Arguments

object	a fitted object of class inheriting from "glm".
newdata	optionally, a data frame in which to look for variables with which to predict. If omitted, the fitted linear predictors are used.
type	the type of prediction required. The default is on the scale of the linear predictors; the alternative "response" is on the scale of the response variable. Thus for a default binomial model the default predictions are of log-odds (probabilities on logit scale) and type = "response" gives the predicted probabilities. The "terms" option returns a matrix giving the fitted values of each term in the model formula on the linear predictor scale. The value of this argument can be abbreviated.
se.fit	logical switch indicating if standard errors are required.
dispersion	the dispersion of the GLM fit to be assumed in computing the standard errors. If omitted, that returned by summary applied to the object is used.
terms	with type="terms" by default all terms are returned. A character vector specifies which terms are to be returned
na.action	function determining what should be done with missing values in newdata. The default is to predict NA.
...	further arguments passed to or from other methods.

### Value

If se = FALSE, a vector or matrix of predictions. If se = TRUE, a list with components

fit	Predictions
se.fit	Estimated standard errors
residual.scale	A scalar giving the square root of the dispersion used in computing the standard errors.

**Note**

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). As from R 2.0.0 a warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

**See Also**

[glm](#), [SafePrediction](#)

**Examples**

```
## example from Venables and Ripley (2002, pp. 190-2.)
ldose <- rep(0:5, 2)
numdead <- c(1, 4, 9, 13, 18, 20, 0, 2, 6, 10, 12, 16)
sex <- factor(rep(c("M", "F"), c(6, 6)))
SF <- cbind(numdead, numalive=20-numdead)
budworm.lg <- glm(SF ~ sex*ldose, family=binomial)
summary(budworm.lg)

plot(c(1,32), c(0,1), type = "n", xlab = "dose",
      ylab = "prob", log = "x")
text(2^ldose, numdead/20, as.character(sex))
ld <- seq(0, 5, 0.1)
lines(2^ld, predict(budworm.lg, data.frame(ldose=ld,
      sex=factor(rep("M", length(ld)), levels=levels(sex))),
      type = "response"))
lines(2^ld, predict(budworm.lg, data.frame(ldose=ld,
      sex=factor(rep("F", length(ld)), levels=levels(sex))),
      type = "response"))
```

---

predict.HoltWinters

*prediction function for fitted Holt-Winters models*

---

**Description**

Computes predictions and prediction intervals for models fitted by the Holt-Winters method.

**Usage**

```
## S3 method for class 'HoltWinters':
predict(object, n.ahead=1, prediction.interval = FALSE,
        level = 0.95, ...)
```

**Arguments**

<code>object</code>	An object of class <code>HoltWinters</code> .
<code>n.ahead</code>	Number of future periods to predict.
<code>prediction.interval</code>	logical. If <code>TRUE</code> , the lower and upper bounds of the corresponding prediction intervals are computed.
<code>level</code>	Confidence level for the prediction interval.
<code>...</code>	arguments passed to or from other methods.

**Value**

A time series of the predicted values. If prediction intervals are requested, a multiple time series is returned with columns `fit`, `lwr` and `upr` for the predicted values and the lower and upper bounds respectively.

**Author(s)**

David Meyer (David.Meyer@wu-wien.ac.at)

**References**

C. C. Holt (1957) Forecasting seasonals and trends by exponentially weighted moving averages, ONR Research Memorandum, Carnegie Institute 52.

P. R. Winters (1960) Forecasting sales by exponentially weighted moving averages, *Management Science* **6**, 324–342.

**See Also**

[HoltWinters](#)

**Examples**

```
m <- HoltWinters(co2)
p <- predict(m, 50, prediction.interval = TRUE)
plot(m, p)
```

---

`predict.lm`

*Predict method for Linear Model Fits*

---

**Description**

Predicted values based on linear model object

**Usage**

```
## S3 method for class 'lm':
predict(object, newdata, se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, type = c("response", "terms"),
        terms = NULL, na.action = na.pass, ...)
```

**Arguments**

<code>object</code>	Object of class inheriting from "lm"
<code>newdata</code>	An optional data frame in which to look for variables with which to predict. If omitted, the fitted values are used.
<code>se.fit</code>	A switch indicating if standard errors are required.
<code>scale</code>	Scale parameter for std.err. calculation
<code>df</code>	Degrees of freedom for scale
<code>interval</code>	Type of interval calculation
<code>level</code>	Tolerance/confidence level
<code>type</code>	Type of prediction (response or model term)
<code>terms</code>	If <code>type="terms"</code> , which terms (default is all terms)
<code>na.action</code>	function determining what should be done with missing values in <code>newdata</code> . The default is to predict NA.
<code>...</code>	further arguments passed to or from other methods.

**Details**

`predict.lm` produces predicted values, obtained by evaluating the regression function in the frame `newdata` (which defaults to `model.frame(object)`). If the logical `se.fit` is TRUE, standard errors of the predictions are calculated. If the numeric argument `scale` is set (with optional `df`), it is used as the residual standard deviation in the computation of the standard errors, otherwise this is extracted from the model fit. Setting `intervals` specifies computation of confidence or prediction (tolerance) intervals at the specified `level`.

If the fit is rank-deficient, some of the columns of the design matrix will have been dropped. Prediction from such a fit only makes sense if `newdata` is contained in the same subspace as the original data. That cannot be checked accurately, so a warning is issued.

**Value**

`predict.lm` produces a vector of predictions or a matrix of predictions and bounds with column names `fit`, `lwr`, and `upr` if `interval` is set. If `se.fit` is TRUE, a list with the following components is returned:

<code>fit</code>	vector or matrix as above
<code>se.fit</code>	standard error of predictions
<code>residual.scale</code>	residual standard deviations
<code>df</code>	degrees of freedom for residual

**Note**

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). As from R 2.0.0 a warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

Offsets specified by `offset` in the fit by `lm` will not be included in predictions, whereas those specified by an `offset` term in the formula will be.

**See Also**

The model fitting function `lm`, `predict`, `SafePrediction`

**Examples**

```
## Predictions
x <- rnorm(15)
y <- x + rnorm(15)
predict(lm(y ~ x))
new <- data.frame(x = seq(-3, 3, 0.5))
predict(lm(y ~ x), new, se.fit = TRUE)
pred.w.plim <- predict(lm(y ~ x), new, interval="prediction")
pred.w.clim <- predict(lm(y ~ x), new, interval="confidence")
matplot(new$x, cbind(pred.w.clim, pred.w.plim[, -1]),
        lty=c(1,2,2,3,3), type="l", ylab="predicted y")
```

---

predict.loess

*Predict Loess Curve or Surface*

---

**Description**

Predictions from a loess fit, optionally with standard errors.

**Usage**

```
## S3 method for class 'loess':
predict(object, newdata = NULL, se = FALSE, ...)
```

**Arguments**

<code>object</code>	an object fitted by loess.
<code>newdata</code>	an optional data frame in which to look for variables with which to predict. If missing, the original data points are used.
<code>se</code>	should standard errors be computed?
<code>...</code>	arguments passed to or from other methods.

**Details**

The standard errors calculation is slower than prediction.

When the fit was made using `surface="interpolate"` (the default), `predict.loess` will not extrapolate – so points outside an axis-aligned hypercube enclosing the original data will have missing (NA) predictions and standard errors.

**Value**

If `se = FALSE`, a vector giving the prediction for each row of `newdata` (or the original data). If `se = TRUE`, a list containing components

<code>fit</code>	the predicted values.
<code>se</code>	an estimated standard error for each predicted value.
<code>residual.scale</code>	the estimated scale of the residuals used in computing the standard errors.
<code>df</code>	an estimate of the effective degrees of freedom used in estimating the residual scale, intended for use with t-based confidence intervals.

If `newdata` was the result of a call to [expand.grid](#), the predictions (and s.e.'s if requested) will be an array of the appropriate dimensions.

**Note**

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). As from R 2.0.0 a warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

**Author(s)**

B.D. Ripley, based on the `cloess` package of Cleveland, Grosse and Shyu.

**See Also**

[loess](#)

**Examples**

```
cars.lo <- loess(dist ~ speed, cars)
predict(cars.lo, data.frame(speed=seq(5, 30, 1)), se=TRUE)
# to get extrapolation
cars.lo2 <- loess(dist ~ speed, cars,
  control=loess.control(surface="direct"))
predict(cars.lo2, data.frame(speed=seq(5, 30, 1)), se=TRUE)
```

---

 predict.nls

*Predicting from Nonlinear Least Squares Fits*


---

### Description

predict.nls produces predicted values, obtained by evaluating the regression function in the frame newdata. If the logical se.fit is TRUE, standard errors of the predictions are calculated. If the numeric argument scale is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this is extracted from the model fit. Setting interval specifies computation of confidence or prediction (tolerance) intervals at the specified level.

At present se.fit and interval are ignored.

### Usage

```
## S3 method for class 'nls':
predict(object, newdata, se.fit = FALSE, scale = NULL, df = Inf,
        interval = c("none", "confidence", "prediction"),
        level = 0.95, ...)
```

### Arguments

object	An object that inherits from class nls.
newdata	A named list or data frame in which to look for variables with which to predict. If newdata is missing the fitted values at the original data points are returned.
se.fit	A logical value indicating if the standard errors of the predictions should be calculated. Defaults to FALSE. At present this argument is ignored.
scale	A numeric scalar. If it is set (with optional df), it is used as the residual standard deviation in the computation of the standard errors, otherwise this information is extracted from the model fit. At present this argument is ignored.
df	A positive numeric scalar giving the number of degrees of freedom for the scale estimate. At present this argument is ignored.
interval	A character string indicating if prediction intervals or a confidence interval on the mean responses are to be calculated. At present this argument is ignored.
level	A numeric scalar between 0 and 1 giving the confidence level for the intervals (if any) to be calculated. At present this argument is ignored.
...	Additional optional arguments. At present no optional arguments are used.

### Value

predict.nls produces a vector of predictions. When implemented, interval will produce a matrix of predictions and bounds with column names fit, lwr, and upr. When implemented, if se.fit is TRUE, a list with the following components will be returned:

fit	vector or matrix as above
-----	---------------------------

se.fit	standard error of predictions
residual.scale	residual standard deviations
df	degrees of freedom for residual

**Note**

Variables are first looked for in `newdata` and then searched for in the usual way (which will include the environment of the formula used in the fit). As from R 2.0.0 a warning will be given if the variables found are not of the same length as those in `newdata` if it was supplied.

**See Also**

The model fitting function [nls](#), [predict](#).

**Examples**

```
fm <- nls(demand ~ SSasymOrig(Time, A, lrc), data = BOD)
predict(fm)           # fitted values at observed times
## Form data plot and smooth line for the predictions
opar <- par(las = 1)
plot(demand ~ Time, data = BOD, col = 4,
     main = "BOD data and fitted first-order curve",
     xlim = c(0,7), ylim = c(0, 20) )
tt <- seq(0, 8, length = 101)
lines(tt, predict(fm, list(Time = tt)))
par(opar)
```

---

predict.smooth.spline

*Predict from Smoothing Spline Fit*

---

**Description**

Predict a smoothing spline fit at new points, return the derivative if desired. The predicted fit is linear beyond the original data.

**Usage**

```
## S3 method for class 'smooth.spline':
predict(object, x, deriv = 0, ...)
```

**Arguments**

object	a fit from <code>smooth.spline</code> .
x	the new values of x.
deriv	integer; the order of the derivative required.
...	further arguments passed to or from other methods.

**Value**

A list with components

`x`                   The input `x`.  
`y`                    The fitted values or derivatives at `x`.

**See Also**

[smooth.spline](#)

**Examples**

```
attach(cars)
cars.spl <- smooth.spline(speed, dist, df=6.4)

## "Proof" that the derivatives are okay, by comparing with approximation
diff.quot <- function(x,y) {
  ## Difference quotient (central differences where available)
  n <- length(x); i1 <- 1:2; i2 <- (n-1):n
  c(diff(y[i1]) / diff(x[i1]), (y[-i1] - y[-i2]) / (x[-i1] - x[-i2]),
    diff(y[i2]) / diff(x[i2]))
}

xx <- unique(sort(c(seq(0,30, by = .2), kn <- unique(speed))))
i.kn <- match(kn, xx)# indices of knots within xx
op <- par(mfrow = c(2,2))
plot(speed, dist, xlim = range(xx), main = "Smooth.spline & derivatives")
lines(pp <- predict(cars.spl, xx), col = "red")
points(kn, pp$y[i.kn], pch = 3, col="dark red")
mtext("s(x)", col = "red")
for(d in 1:3){
  n <- length(pp$x)
  plot(pp$x, diff.quot(pp$x,pp$y), type = 'l', xlab="x", ylab="",
    col = "blue", col.main = "red",
    main= paste("s",paste(rep("'",d), collapse=""),"(x)", sep=""))
  mtext("Difference quotient approx.(last)", col = "blue")
  lines(pp <- predict(cars.spl, xx, deriv = d), col = "red")

  points(kn, pp$y[i.kn], pch = 3, col="dark red")
  abline(h=0, lty = 3, col = "gray")
}
detach(); par(op)
```

---

```
preplot
```

---

*Pre-computations for a Plotting Object*

---

**Description**

Compute an object to be used for plots relating to the given model object.

**Usage**

```
preplot(object, ...)
```

**Arguments**

`object` a fitted model object.  
`...` additional arguments for specific methods.

**Details**

Only the generic function is currently provided in base R, but some add-on packages have methods. Principally here for S compatibility.

**Value**

An object set up to make a plot that describes `object`.

---

<code>princomp</code>	<i>Principal Components Analysis</i>
-----------------------	--------------------------------------

---

**Description**

`princomp` performs a principal components analysis on the given numeric data matrix and returns the results as an object of class `princomp`.

**Usage**

```
princomp(x, ...)
```

```
## S3 method for class 'formula':
princomp(formula, data = NULL, subset, na.action, ...)
```

```
## Default S3 method:
princomp(x, cor = FALSE, scores = TRUE, covmat = NULL,
         subset = rep(TRUE, nrow(as.matrix(x))), ...)
```

```
## S3 method for class 'princomp':
predict(object, newdata, ...)
```

**Arguments**

`formula` a formula with no response variable.  
`data` an optional data frame containing the variables in the formula `formula`. By default the variables are taken from `environment(formula)`.  
`subset` an optional vector used to select rows (observations) of the data matrix `x`.

<code>na.action</code>	a function which indicates what should happen when the data contain NAs. The default is set by the <code>na.action</code> setting of <code>options</code> , and is <code>na.fail</code> if that is unset. The “factory-fresh” default is <code>na.omit</code> .
<code>x</code>	a numeric matrix or data frame which provides the data for the principal components analysis.
<code>cor</code>	a logical value indicating whether the calculation should use the correlation matrix or the covariance matrix.
<code>scores</code>	a logical value indicating whether the score on each principal component should be calculated.
<code>covmat</code>	a covariance matrix, or a covariance list as returned by <code>cov.wt</code> (and <code>cov.mve</code> or <code>cov.mcd</code> from package <b>MASS</b> ). If supplied, this is used rather than the covariance matrix of <code>x</code> .
<code>...</code>	arguments passed to or from other methods. If <code>x</code> is a formula one might specify <code>cor</code> or <code>scores</code> .
<code>object</code>	Object of class inheriting from "princomp"
<code>newdata</code>	An optional data frame in which to look for variables with which to predict. If omitted, the scores are used.

### Details

`princomp` is a generic function with "formula" and "default" methods.

The calculation is done using `eigen` on the correlation or covariance matrix, as determined by `cor`. This is done for compatibility with the S-PLUS result. A preferred method of calculation is to use `svd` on `x`, as is done in `prcomp`.

Note that the default calculation uses divisor `N` for the covariance matrix.

The `print` method for these objects prints the results in a nice format and the `plot` method produces a scree plot (`screeplot`). There is also a `biplot` method.

If `x` is a formula then the standard NA-handling is applied to the scores (if requested): see `napredict`.

`princomp` only handles so-called R-mode PCA, that is feature extraction of variables. If a data matrix is supplied (possibly via a formula) it is required that there are at least as many units as variables. For Q-mode PCA use `prcomp`.

### Value

`princomp` returns a list with class "princomp" containing the following components:

<code>sdev</code>	the standard deviations of the principal components.
<code>loadings</code>	the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors). This is of class "loadings": see <code>loadings</code> for its <code>print</code> method.
<code>center</code>	the means that were subtracted.
<code>scale</code>	the scalings applied to each variable.
<code>n.obs</code>	the number of observations.

scores	if scores = TRUE, the scores of the supplied data on the principal components. These are non-null only if x was supplied, and if covmat was also supplied if it was a covariance list.
call	the matched call.
na.action	If relevant.

**Note**

The signs of the columns of the loadings and scores are arbitrary, and so may differ between different programs for PCA, and even between different builds of R.

**References**

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.  
 Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag.

**See Also**

[summary.princomp](#), [screeplot](#), [biplot.princomp](#), [prcomp](#), [cor](#), [cov](#), [eigen](#).

**Examples**

```
## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
(pc.cr <- princomp(USArrests)) # inappropriate
princomp(USArrests, cor = TRUE) # ^= prcomp(USArrests, scale=TRUE)
## Similar, but different:
## The standard deviations differ by a factor of sqrt(49/50)

summary(pc.cr <- princomp(USArrests, cor = TRUE))
loadings(pc.cr) ## note that blank entries are small but not zero
plot(pc.cr) # shows a screeplot.
biplot(pc.cr)

## Formula interface
princomp(~ ., data = USArrests, cor = TRUE)
# NA-handling
USArrests[1, 2] <- NA
pc.cr <- princomp(~ Murder + Assault + UrbanPop,
                  data = USArrests, na.action=na.exclude, cor = TRUE)
pc.cr$scores
```

---

print.power.htest *Print method for power calculation object*

---

**Description**

Print object of class "power.htest" in nice layout.

**Usage**

```
## S3 method for class 'power.htest':  
print(x, ...)
```

**Arguments**

x                    Object of class "power.htest".  
...                  further arguments to be passed to or from methods.

**Details**

A `power.htest` object is just a named list of numbers and character strings, supplemented with method and note elements. The method is displayed as a title, the note as a footnote, and the remaining elements are given in an aligned 'name = value' format.

**Value**

none

**Author(s)**

Peter Dalgaard

**See Also**

[power.t.test](#), [power.prop.test](#)

---

print.ts

*Printing Time-Series Objects*

---

**Description**

Print method for time series objects.

**Usage**

```
## S3 method for class 'ts':  
print(x, calendar, ...)
```

**Arguments**

x                    a time series object.  
calendar            enable/disable the display of information about month names, quarter names or year when printing. The default is TRUE for a frequency of 4 or 12, FALSE otherwise.  
...                  additional arguments to [print](#).

**Details**

This is the `print` methods for objects inheriting from class "ts".

**See Also**

`print`, `ts`.

**Examples**

```
print(ts(1:10, freq = 7, start = c(12, 2)), calendar = TRUE)
```

---

<code>printCoefmat</code>	<i>Print Coefficient Matrices</i>
---------------------------	-----------------------------------

---

**Description**

Utility function to be used in “higher level” `print` methods, such as `print.summary.lm`, `print.summary.glm` and `print.anova`. The goal is to provide a flexible interface with smart defaults such that often, only `x` needs to be specified.

**Usage**

```
printCoefmat(x, digits=max(3, getOption("digits") - 2),
             signif.stars = getOption("show.signif.stars"),
             signif.legend = signif.stars,
             dig.tst = max(1, min(5, digits - 1)),
             cs.ind = 1:k, tst.ind = k + 1, zap.ind = integer(0),
             P.values = NULL,
             has.Pvalue = nc >= 4 &&
                 substr(colnames(x)[nc], 1, 3) == "Pr(",
             eps.Pvalue = .Machine$double.eps,
             na.print = "NA", ...)
```

**Arguments**

<code>x</code>	a numeric matrix like object, to be printed.
<code>digits</code>	minimum number of significant digits to be used for most numbers.
<code>signif.stars</code>	logical; if TRUE, P-values are additionally encoded visually as “significance stars” in order to help scanning of long coefficient tables. It defaults to the <code>show.signif.stars</code> slot of <code>options</code> .
<code>signif.legend</code>	logical; if TRUE, a legend for the “significance stars” is printed.
<code>dig.tst</code>	minimum number of significant digits for the test statistics, see <code>tst.ind</code> .
<code>cs.ind</code>	indices (integer) of column numbers which are (like) coefficients and standard errors to be formatted together.

<code>tst.ind</code>	indices (integer) of column numbers for test statistics.
<code>zap.ind</code>	indices (integer) of column numbers which should be formatted by <code>zapsmall</code> , i.e., by “zapping” values close to 0.
<code>P.values</code>	logical or NULL; if TRUE, the last column of <code>x</code> is formatted by <code>format.pval</code> as P values. If <code>P.values = NULL</code> , the default, it is set to TRUE only if <code>options("show.coef.Pvalue")</code> is TRUE <i>and</i> <code>x</code> has at least 4 columns <i>and</i> the last column name of <code>x</code> starts with <code>"Pr("</code> .
<code>has.Pvalue</code>	logical; if TRUE, the last column of <code>x</code> contains P values; in that case, it is printed if and only if <code>P.values</code> (above) is true.
<code>eps.Pvalue</code>	number, ..
<code>na.print</code>	a character string to code NA values in printed output.
<code>...</code>	further arguments for <code>print</code> .

**Value**

Invisibly returns its argument, `x`.

**Author(s)**

Martin Maechler

**See Also**

`print.summary.lm`, `format.pval`, `format`.

**Examples**

```

cmat <- cbind(rnorm(3, 10), sqrt(rchisq(3, 12)))
cmat <- cbind(cmat, cmat[,1]/cmat[,2])
cmat <- cbind(cmat, 2*pnorm(-cmat[,3]))
colnames(cmat) <- c("Estimate", "Std.Err", "Z value", "Pr(>z)")
printCoefmat(cmat[,1:3])
printCoefmat(cmat)
options(show.coef.Pvalues = FALSE)
printCoefmat(cmat, digits=2)
printCoefmat(cmat, digits=2, P.values = TRUE)
options(show.coef.Pvalues = TRUE)# revert

```

**Description**

Investigates behavior of objective function near the solution represented by `fitted`.

See documentation on method functions for further details.

**Usage**

```
profile(fitted, ...)
```

**Arguments**

`fitted` the original fitted model object.  
`...` additional parameters. See documentation on individual methods.

**Value**

A list with an element for each parameter being profiled. See the individual methods for further details.

**See Also**

[profile.nls](#), [profile.glm](#) in package **MASS**, ...  
 For profiling code, see [Rprof](#).

---

`profile.nls` *Method for Profiling nls Objects*

---

**Description**

Investigates behavior of the log-likelihood function near the solution represented by `fitted`.

**Usage**

```
## S3 method for class 'nls':
profile(fitted, which = 1:npar, maxpts = 100, alphamax = 0.01,
        delta.t = cutoff/5, ...)
```

**Arguments**

`fitted` the original fitted model object.  
`which` the original model parameters which should be profiled. By default, all parameters are profiled.  
`maxpts` maximum number of points to be used for profiling each parameter.  
`alphamax` maximum significance level allowed for the profile t-statistics.  
`delta.t` suggested change on the scale of the profile t-statistics. Default value chosen to allow profiling at about 10 parameter values.  
`...` further arguments passed to or from other methods.

**Details**

The profile t-statistics is defined as the square root of change in sum-of-squares divided by residual standard error with an appropriate sign.

**Value**

A list with an element for each parameter being profiled. The elements are data-frames with two variables

`par.vals`        a matrix of parameter values for each fitted model.  
`tau`             The profile t-statistics.

**Author(s)**

Douglas M. Bates and Saikat DebRoy

**References**

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley (chapter 6)

**See Also**

[nls](#), [profile](#), [profiler.nls](#), [plot.profile.nls](#)

**Examples**

```
# obtain the fitted object
fml <- nls(demand ~ SSasymptOrig( Time, A, lrc ), data = BOD)
# get the profile for the fitted model
pr1 <- profile( fml )
# profiled values for the two parameters
pr1$A
pr1$lrc
```

---

`profiler`

*Constructor for Profiler Objects for Nonlinear Models*

---

**Description**

Create a profiler object for the model object `fitted`.

**Usage**

```
profiler(fitted, ...)
```

**Arguments**

`fitted`        the original fitted model object.  
`...`         Additional parameters. See documentation on individual methods.

**Value**

An object of class "profiler" which is a list with function elements

`getFittedPars()`

the parameters in `fitted`

`setDefault(varying, params)`

this is used for changing the default settings for profiling. In absence of both parameters, the default is set to the original fitted parameters with all parameters varying. The arguments are

`varying`: a logical, integer or character vector giving parameters to be varied.

`params`: the default value at which profiling is to take place.

`getProfile(varying, params)`

this can be used in conjunction with `setDefault` without any arguments. Alternatively, the parameters to be varied and the values for fixed parameters can be specified using the arguments. The arguments are

`varying`: a logical vector giving parameters to be varied. This can be omitted if `params` is a named list or numeric vector.

`params`: values for parameters to be held fixed.

It returns a list with elements

`parameters`: the parameter values for the profiled optimum.

`fstat`: a profile statistics. See individual methods for details.

`varying`: a logical vector indicating parameters which were varied.

**Author(s)**

Douglas M. Bates and Saikat DebRoy

**See Also**

[profiler.nls](#), [profile](#)

**Examples**

```
# see documentation on individual methods
```

---

`profiler.nls`

*Constructor for Profiler Objects from nls Objects*

---

**Description**

Create a profiler object for the model object `fitted` of class `nls`.

**Usage**

```
## S3 method for class 'nls':
profiler(fitted, ...)
```

**Arguments**

fitted            the original fitted model object of class `nls`.  
 ...              Additional parameters. None are used.

**Value**

An object of class `profiler.nls` which is a list with function elements

`getFittedModel()`  
                   the `nlsModel` object corresponding to `fitted`  
`getFittedPars()`  
                   See documentation for `profiler`  
`setDefault(varying, params)`  
                   See documentation for `profiler`  
`getProfile(varying, params)`  
                   In the returned list, `fstat` is the ratio of change in sum-of-squares and the residual standard error.  
                   For other details, see documentation for `profiler`

**Warning**

When using `setDefault` and `getProfile` together, the internal state of the fitted model may get changed. So after completing the profiling for a parameter, the internal states should be restored by a call to `setDefault` without any arguments. For example see below or the source for `profile.nls`.

**Author(s)**

Douglas M. Bates and Saikat DebRoy

**References**

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, Wiley

**See Also**

`nls`, `nlsModel`, `profiler`, `profile.nls`

**Examples**

```
## obtain the fitted object
fml <- nls(demand ~ SSasymOrig( Time, A, lrc ), data = BOD)
## get the profile for the fitted model
prof1 <- profiler( fml )
## profile with A fixed at 16.0
prof1$getProfile(c(FALSE, TRUE), 16.0)
## vary lrc
prof1$setDefault(varying = c(FALSE, TRUE))
## fix A at 14.0 and starting estimate of lrc at -0.2
prof1$setDefault(params = c(14.0, -0.2))
## and get the profile
```

```

prof1$getProfile()
## finally, set defaults back to original estimates
prof1$setDefault()

```

---

proj

*Projections of Models*


---

### Description

`proj` returns a matrix or list of matrices giving the projections of the data onto the terms of a linear model. It is most frequently used for [aov](#) models.

### Usage

```

proj(object, ...)

## S3 method for class 'aov':
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)

## S3 method for class 'aovlist':
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)

## Default S3 method:
proj(object, onedf = TRUE, ...)

## S3 method for class 'lm':
proj(object, onedf = FALSE, unweighted.scale = FALSE, ...)

```

### Arguments

<code>object</code>	An object of class "lm" or a class inheriting from it, or an object with a similar structure including in particular components <code>qr</code> and <code>effects</code> .
<code>onedf</code>	A logical flag. If <code>TRUE</code> , a projection is returned for all the columns of the model matrix. If <code>FALSE</code> , the single-column projections are collapsed by terms of the model (as represented in the analysis of variance table).
<code>unweighted.scale</code>	If the fit producing <code>object</code> used weights, this determines if the projections correspond to weighted or unweighted observations.
<code>...</code>	Swallow and ignore any other arguments.

### Details

A projection is given for each stratum of the object, so for `aov` models with an `Error` term the result is a list of projections.

**Value**

A projection matrix or (for multi-stratum objects) a list of projection matrices.

Each projection is a matrix with a row for each observations and either a column for each term (`onedf = FALSE`) or for each coefficient (`onedf = TRUE`). Projection matrices from the default method have orthogonal columns representing the projection of the response onto the column space of the Q matrix from the QR decomposition. The fitted values are the sum of the projections, and the sum of squares for each column is the reduction in sum of squares from fitting that column (after those to the left of it).

The methods for `lm` and `aov` models add a column to the projection matrix giving the residuals (the projection of the data onto the orthogonal complement of the model space).

Strictly, when `onedf = FALSE` the result is not a projection, but the columns represent sums of projections onto the columns of the model matrix corresponding to that term. In this case the matrix does not depend on the coding used.

**Author(s)**

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

**References**

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments*. Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[aov](#), [lm](#), [model.tables](#)

**Examples**

```
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
npk.aov <- aov(yield ~ block + N*P*K, npk)
proj(npk.aov)

## as a test, not particularly sensible
options(contrasts=c("contr.helmert", "contr.treatment"))
npk.aovE <- aov(yield ~ N*P*K + Error(block), npk)
proj(npk.aovE)
```

---

`prop.test`*Test for Equal or Given Proportions*

---

**Description**

`prop.test` can be used for testing the null that the proportions (probabilities of success) in several groups are the same, or that they equal certain given values.

**Usage**

```
prop.test(x, n, p = NULL,
          alternative = c("two.sided", "less", "greater"),
          conf.level = 0.95, correct = TRUE)
```

**Arguments**

<code>x</code>	a vector of counts of successes or a matrix with 2 columns giving the counts of successes and failures, respectively.
<code>n</code>	a vector of counts of trials; ignored if <code>x</code> is a matrix.
<code>p</code>	a vector of probabilities of success. The length of <code>p</code> must be the same as the number of groups specified by <code>x</code> , and its elements must be greater than 0 and less than 1.
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter. Only used for testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
<code>conf.level</code>	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Only used when testing the null that a single proportion equals a given value, or that two proportions are equal; ignored otherwise.
<code>correct</code>	a logical indicating whether Yates' continuity correction should be applied.

**Details**

Only groups with finite numbers of successes and failures are used. Counts of successes and failures must be nonnegative and hence not greater than the corresponding numbers of trials which must be positive. All finite counts should be integers.

If `p` is `NULL` and there is more than one group, the null tested is that the proportions in each group are the same. If there are two groups, the alternatives are that the probability of success in the first group is less than, not equal to, or greater than the probability of success in the second group, as specified by `alternative`. A confidence interval for the difference of proportions with confidence level as specified by `conf.level` and clipped to  $[-1, 1]$  is returned. Continuity correction is used only if it does not exceed the difference of the sample proportions in absolute value. Otherwise, if there are more than 2 groups, the alternative is always "two.sided", the returned confidence interval is `NULL`, and continuity correction is never used.

If there is only one group, then the null tested is that the underlying probability of success is  $p$ , or  $.5$  if  $p$  is not given. The alternative is that the probability of success is less than, not equal to, or greater than  $p$  or  $0.5$ , respectively, as specified by `alternative`. A confidence interval for the underlying proportion with confidence level as specified by `conf.level` and clipped to  $[0, 1]$  is returned. Continuity correction is used only if it does not exceed the difference between sample and null proportions in absolute value. The confidence interval is computed by inverting the score test.

Finally, if  $p$  is given and there are more than 2 groups, the null tested is that the underlying probabilities of success are those given by  $p$ . The alternative is always `"two.sided"`, the returned confidence interval is `NULL`, and continuity correction is never used.

### Value

A list with class `"htest"` containing the following components:

<code>statistic</code>	the value of Pearson's chi-squared test statistic.
<code>parameter</code>	the degrees of freedom of the approximate chi-squared distribution of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>estimate</code>	a vector with the sample proportions $x/n$ .
<code>conf.int</code>	a confidence interval for the true proportion if there is one group, or for the difference in proportions if there are 2 groups and $p$ is not given, or <code>NULL</code> otherwise. In the cases where it is not <code>NULL</code> , the returned confidence interval has an asymptotic confidence level as specified by <code>conf.level</code> , and is appropriate to the specified alternative hypothesis.
<code>null.value</code>	the value of $p$ if specified by the null, or <code>NULL</code> otherwise.
<code>alternative</code>	a character string describing the alternative.
<code>method</code>	a character string indicating the method used, and whether Yates' continuity correction was applied.
<code>data.name</code>	a character string giving the names of the data.

### See Also

[binom.test](#) for an *exact* test of a binomial hypothesis.

### Examples

```
heads <- rbinom(1, size=100, pr = .5)
prop.test(heads, 100)           # continuity correction TRUE by default
prop.test(heads, 100, correct = FALSE)

## Data from Fleiss (1981), p. 139.
## H0: The null hypothesis is that the four populations from which
##     the patients were drawn have the same true proportion of smokers.
## A:  The alternative is that this proportion is different in at
##     least one of the populations.

smokers <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
```

---

prop.trend.test      *Test for trend in proportions*

---

### Description

Performs chi-squared test for trend in proportions, i.e., a test asymptotically optimal for local alternatives where the log odds vary in proportion with `score`. By default, `score` is chosen as the group numbers.

### Usage

```
prop.trend.test(x, n, score = 1:length(x))
```

### Arguments

<code>x</code>	Number of events
<code>n</code>	Number of trials
<code>score</code>	Group score

### Value

An object of class "htest" with title, test statistic, p-value, etc.

### Note

This really should get integrated with `prop.test`

### Author(s)

Peter Dalgaard

### See Also

[prop.test](#)

### Examples

```
smokers <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
prop.trend.test(smokers, patients)
prop.trend.test(smokers, patients,c(0,0,0,1))
```

qqnorm

*Quantile-Quantile Plots***Description**

qqnorm is a generic functions the default method of which produces a normal QQ plot of the values in `y`. qqline adds a line to a normal quantile-quantile plot which passes through the first and third quartiles.

qqplot produces a QQ plot of two datasets.

Graphical parameters may be given as arguments to qqnorm, qqplot and qqline.

**Usage**

```
qqnorm(y, ...)
## Default S3 method:
qqnorm(y, ylim, main = "Normal Q-Q Plot",
        xlab = "Theoretical Quantiles",
        ylab = "Sample Quantiles", plot.it = TRUE, datax = FALSE,
        ...)
qqline(y, datax = FALSE, ...)
qqplot(x, y, plot.it = TRUE, xlab = deparse(substitute(x)),
        ylab = deparse(substitute(y)), ...)
```

**Arguments**

<code>x</code>	The first sample for qqplot.
<code>y</code>	The second or only data sample.
<code>xlab</code> , <code>ylab</code> , <code>main</code>	plot labels.
<code>plot.it</code>	logical. Should the result be plotted?
<code>datax</code>	logical. Should data values be on the x-axis?
<code>ylim</code> , ...	graphical parameters.

**Value**

For qqnorm and qqplot, a list with components

<code>x</code>	The x coordinates of the points that were/would be plotted
<code>y</code>	The original <code>y</code> vector, i.e., the corresponding y coordinates <i>including NAs</i> .

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[ppoints](#).

**Examples**

```
y <- rt(200, df = 5)
qqnorm(y); qqline(y, col = 2)
qqplot(y, rt(300, df = 5))

qqnorm(precip, ylab = "Precipitation [in/yr] for 70 US cities")
```

---

quade.test

*Quade Test*


---

**Description**

Performs a Quade test with unrepliated blocked data.

**Usage**

```
quade.test(y, ...)

## Default S3 method:
quade.test(y, groups, blocks, ...)

## S3 method for class 'formula':
quade.test(formula, data, subset, na.action, ...)
```

**Arguments**

<code>y</code>	either a numeric vector of data values, or a data matrix.
<code>groups</code>	a vector giving the group for the corresponding elements of <code>y</code> if this is a vector; ignored if <code>y</code> is a matrix. If not a factor object, it is coerced to one.
<code>blocks</code>	a vector giving the block for the corresponding elements of <code>y</code> if this is a vector; ignored if <code>y</code> is a matrix. If not a factor object, it is coerced to one.
<code>formula</code>	a formula of the form $a \sim b \mid c$ , where <code>a</code> , <code>b</code> and <code>c</code> give the data values and corresponding groups and blocks, respectively.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

## Details

quade.test can be used for analyzing unreplicated complete block designs (i.e., there is exactly one observation in `y` for each combination of levels of `groups` and `blocks`) where the normality assumption may be violated.

The null hypothesis is that apart from an effect of `blocks`, the location parameter of `y` is the same in each of the `groups`.

If `y` is a matrix, `groups` and `blocks` are obtained from the column and row indices, respectively. NA's are not allowed in `groups` or `blocks`; if `y` contains NA's, corresponding blocks are removed.

## Value

A list with class "htest" containing the following components:

<code>statistic</code>	the value of Quade's F statistic.
<code>parameter</code>	a vector with the numerator and denominator degrees of freedom of the approximate F distribution of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>method</code>	the character string "Quade test".
<code>data.name</code>	a character string giving the names of the data.

## References

D. Quade (1979), Using weighted rankings in the analysis of complete blocks with additive block effects. *Journal of the American Statistical Association*, **74**, 680–683.

William J. Conover (1999), *Practical nonparametric statistics*. New York: John Wiley & Sons. Pages 373–380.

## See Also

[friedman.test](#).

## Examples

```
## Conover (1999, p. 375f):
## Numbers of five brands of a new hand lotion sold in seven stores
## during one week.
y <- matrix(c( 5,  4,  7, 10, 12,
              1,  3,  1,  0,  2,
              16, 12, 22, 22, 35,
              5,  4,  3,  5,  4,
              10,  9,  7, 13, 10,
              19, 18, 28, 37, 58,
              10,  7,  6,  8,  7),
            nr = 7, byrow = TRUE,
            dimnames =
            list(Store = as.character(1:7),
                 Brand = LETTERS[1:5]))
y
quade.test(y)
```

---

quantile	<i>Sample Quantiles</i>
----------	-------------------------

---

**Description**

The generic function `quantile` produces sample quantiles corresponding to the given probabilities. The smallest observation corresponds to a probability of 0 and the largest to a probability of 1.

**Usage**

```
quantile(x, ...)

## Default S3 method:
quantile(x, probs = seq(0, 1, 0.25), na.rm = FALSE,
         names = TRUE, type = 7, ...)
```

**Arguments**

<code>x</code>	numeric vectors whose sample quantiles are wanted.
<code>probs</code>	numeric vector of probabilities with values in $[0, 1]$ .
<code>na.rm</code>	logical; if true, any <code>NA</code> and <code>NaN</code> 's are removed from <code>x</code> before the quantiles are computed.
<code>names</code>	logical; if true, the result has a <code>names</code> attribute. Set to <code>FALSE</code> for speedup with many <code>probs</code> .
<code>type</code>	an integer between 1 and 9 selecting one of the nine quantile algorithms detailed below to be used.
<code>...</code>	further arguments passed to or from other methods.

**Details**

A vector of length `length(probs)` is returned; if `names = TRUE`, it has a `names` attribute. `NA` and `NaN` values in `probs` are propagated to the result.

**Types**

`quantile` returns estimates of underlying distribution quantiles based on one or two order statistics from the supplied elements in `x` at probabilities in `probs`. One of the nine quantile algorithms discussed in Hyndman and Fan (1996), selected by `type`, is employed.

Sample quantiles of type  $i$  are defined by

$$Q_i(p) = (1 - \gamma)x_j + \gamma x_{j+1}$$

where  $1 \leq i \leq 9$ ,  $\frac{j-m}{n} \leq p < \frac{j-m+1}{n}$ ,  $x_j$  is the  $j$ th order statistic,  $n$  is the sample size, and  $m$  is a constant determined by the sample quantile type. Here  $\gamma$  depends on the fractional part of  $g = np + m - j$ .

For the continuous sample quantile types (4 through 9), the sample quantiles can be obtained by linear interpolation between the  $k$ th order statistic and  $p(k)$ :

$$p(k) = \frac{k - \alpha}{n - \alpha - \beta + 1}$$

where  $\alpha$  and  $\beta$  are constants determined by the type. Further,  $m = \alpha + p(1 - \alpha - \beta)$ , and  $\gamma = g$ .

### Discontinuous sample quantile types 1, 2, and 3

**Type 1** Inverse of empirical distribution function.

**Type 2** Similar to type 1 but with averaging at discontinuities.

**Type 3** SAS definition: nearest even order statistic.

### Continuous sample quantile types 4 through 9

**Type 4**  $p(k) = \frac{k}{n}$ . That is, linear interpolation of the empirical cdf.

**Type 5**  $p(k) = \frac{k-0.5}{n}$ . That is a piecewise linear function where the knots are the values midway through the steps of the empirical cdf. This is popular amongst hydrologists.

**Type 6**  $p(k) = \frac{k}{n+1}$ . Thus  $p(k) = E[F(x_k)]$ . This is used by Minitab and by SPSS.

**Type 7**  $p(k) = \frac{k-1}{n-1}$ . In this case,  $p(k) = \text{mode}[F(x_k)]$ . This is used by S.

**Type 8**  $p(k) = \frac{k-\frac{1}{3}}{n+\frac{1}{3}}$ . Then  $p(k) \approx \text{median}[F(x_k)]$ . The resulting quantile estimates are approximately median-unbiased regardless of the distribution of  $\mathbf{x}$ .

**Type 9**  $p(k) = \frac{k-\frac{3}{8}}{n+\frac{3}{4}}$ . The resulting quantile estimates are approximately unbiased if  $\mathbf{x}$  is normally distributed.

Hyndman and Fan (1996) recommend type 8. The default method is type 7, as used by S and by R < 2.0.0.

### Author(s)

of the version of R 2.0.0, Ivan Frohne and Rob J Hyndman.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Hyndman, R. J. and Fan, Y. (1996) Sample quantiles in statistical packages, *American Statistician*, **50**, 361–365.

### See Also

[ecdf](#) for empirical distributions of which `quantile` is the “inverse”; [boxplot.stats](#) and [fivenum](#) for computing “versions” of quartiles, etc.

**Examples**

```

quantile(x <- rnorm(1001))# Extremes & Quartiles by default
quantile(x, probs=c(.1,.5,1,2,5,10,50, NA)/100)

### Compare different types
p <- c(0.1,0.5,1,2,5,10,50)/100
res <- matrix(as.numeric(NA), 9, 7)
for(type in 1:9) res[type, ] <- y <- quantile(x, p, type=type)
dimnames(res) <- list(1:9, names(y))
round(res, 3)

```

---

read.ftable	<i>Manipulate Flat Contingency Tables</i>
-------------	---

---

**Description**

Read, write and coerce “flat” contingency tables.

**Usage**

```

read.ftable(file, sep = "", quote = "\"",
            row.var.names, col.vars, skip = 0)

write.ftable(x, file = "", quote = TRUE,
            digits = getOption("digits"))

## S3 method for class 'ftable':
as.table(x, ...)

```

**Arguments**

file	either a character string naming a file or a connection which the data are to be read from or written to. "" indicates input from the console for reading and output to the console for writing.
sep	the field separator string. Values on each line of the file are separated by this string.
quote	a character string giving the set of quoting characters for read.ftable; to disable quoting altogether, use quote="". For write.table, a logical indicating whether strings in the data will be surrounded by double quotes.
row.var.names	a character vector with the names of the row variables, in case these cannot be determined automatically.
col.vars	a list giving the names and levels of the column variables, in case these cannot be determined automatically.
skip	the number of lines of the data file to skip before beginning to read data.
x	an object of class "ftable".

`digits`        an integer giving the number of significant digits to use for (the cell entries of) `x`.

`...`            further arguments to be passed to or from methods.

### Details

`read.ftable` reads in a flat-like contingency table from a file. If the file contains the written representation of a flat table (more precisely, a header with all information on names and levels of column variables, followed by a line with the names of the row variables), no further arguments are needed. Similarly, flat tables with only one column variable the name of which is the only entry in the first line are handled automatically. Other variants can be dealt with by skipping all header information using `skip`, and providing the names of the row variables and the names and levels of the column variable using `row.var.names` and `col.vars`, respectively. See the examples below.

Note that flat tables are characterized by their “ragged” display of row (and maybe also column) labels. If the full grid of levels of the row variables is given, one should instead use `read.table` to read in the data, and create the contingency table from this using `xtabs`.

`write.ftable` writes a flat table to a file, which is useful for generating “pretty” ASCII representations of contingency tables.

`as.table.ftable` converts a contingency table in flat matrix form to one in standard array form. This is a method for the generic function `as.table`.

### References

Agresti, A. (1990) *Categorical data analysis*. New York: Wiley.

### See Also

[ftable](#) for more information on flat contingency tables.

### Examples

```
## Agresti (1990), page 157, Table 5.8.
## Not in ftable standard format, but o.k.
file <- tempfile()
cat("          Intercourse\n",
    "Race Gender    Yes No\n",
    "White Male      43 134\n",
    "    Female      26 149\n",
    "Black Male      29  23\n",
    "    Female      22  36\n",
    file = file)
file.show(file)
ft <- read.ftable(file)
ft
unlink(file)

## Agresti (1990), page 297, Table 8.16.
## Almost o.k., but misses the name of the row variable.
file <- tempfile()
```



**Examples**

```
hca <- hclust(dist(USArrests))
plot(hca)
rect.hclust(hca, k=3, border="red")
x <- rect.hclust(hca, h=50, which=c(2,7), border=3:4)
x
```

---

relevel

*Reorder Levels of Factor*

---

**Description**

The levels of a factor are re-ordered so that the level specified by `ref` is first and the others are moved down. This is useful for `contr.treatment` contrasts which take the first level as the reference.

**Usage**

```
relevel(x, ref, ...)
```

**Arguments**

<code>x</code>	An unordered factor.
<code>ref</code>	The reference level.
<code>...</code>	Additional arguments for future methods.

**Value**

A factor of the same length as `x`.

**See Also**

[factor](#), [contr.treatment](#)

**Examples**

```
warpbreaks$tension <- relevel(warpbreaks$tension, ref="M")
summary(lm(breaks ~ wool + tension, data=warpbreaks))
```

---

reorder	<i>Reorder a dendrogram</i>
---------	-----------------------------

---

### Description

There are many different orderings of a dendrogram that are consistent with the structure imposed. This function takes a dendrogram and a vector of values and reorders the dendrogram in the order of the supplied vector, maintaining the constraints on the dendrogram.

### Usage

```
reorder(x, ...)  
  
## S3 method for class 'dendrogram':  
reorder(x, wts, agгло.FUN = sum, ...)
```

### Arguments

<code>x</code>	the (dendrogram) object to be reordered
<code>wts</code>	numeric weights (arbitrary values) for reordering.
<code>agгло.FUN</code>	a function for weights agglomeration, see below.
<code>...</code>	additional arguments

### Details

Using the weights `wts`, the leaves of the dendrogram are reordered so as to be in an order as consistent as possible with the weights. At each node, the branches are ordered in increasing weights where the weight of a branch is defined as  $f(w_j)$  where  $f$  is `agгло.FUN` and  $w_j$  is the weight of the  $j$ -th sub branch).

### Value

From `reorder.dendrogram`, a dendrogram where each node has a further attribute `value` with its corresponding weight.

### Author(s)

R. Gentleman and M. Maechler

### See Also

[rev.dendrogram](#) which simply reverses the nodes' order; [heatmap](#), [cophenetic](#).

**Examples**

```

set.seed(123)
x <- rnorm(10)
hc <- hclust(dist(x))
dd <- as.dendrogram(hc)
dd.reorder <- reorder(dd, 10:1)
plot(dd, main = "random dendrogram `dd'")

op <- par(mfcol = 1:2)
plot(dd.reorder, main = "reorder(dd, 10:1)")
plot(reorder(dd,10:1, agglo.FUN= mean),
      main = "reorder(dd, 10:1, mean)")
par(op)

```

---

reorder.factor	<i>Reorder levels of a factor</i>
----------------	-----------------------------------

---

**Description**

Reorders the levels of a factor depending on values of a second variable, usually numeric.

**Usage**

```

## S3 method for class 'factor':
reorder(x, X, FUN = mean, ...,
        order = is.ordered(x))

```

**Arguments**

x	a factor (possibly ordered) whose levels will be reordered.
X	a vector of the same length as x, whose subset of values for each unique level of x determines the eventual order of that level.
FUN	a function whose first argument is a vector and returns a scalar, to be applied to each subset of X determined by the levels of x.
...	optional: extra arguments supplied to FUN
order	logical, whether return value will be an ordered factor rather than a factor.

**Value**

A factor or an ordered factor (depending on the value of `order`), with the order of the levels determined by `FUN` applied to `X` grouped by `x`. The levels are ordered such that the values returned by `FUN` are in increasing order.

Additionally, the values of `FUN` applied to the subsets of `X` (in the original order of the levels of `x`) is returned as the "scores" attribute.

**Author(s)**

Deepayan Sarkar <deepayan@stat.wisc.edu>

**Examples**

```
bymedian <- with(InsectSprays, reorder(spray, count, median))
boxplot(count ~ bymedian, data = InsectSprays,
        xlab = "Type of spray", ylab = "Insect count",
        main = "InsectSprays data", varwidth = TRUE,
        col = "lightgray")
```

---

replications	<i>Number of Replications of Terms</i>
--------------	--

---

**Description**

Returns a vector or a list of the number of replicates for each term in the formula.

**Usage**

```
replications(formula, data=NULL, na.action)
```

**Arguments**

<code>formula</code>	a formula or a terms object or a data frame.
<code>data</code>	a data frame used to find the objects in <code>formula</code> .
<code>na.action</code>	function for handling missing values. Defaults to a <code>na.action</code> attribute of <code>data</code> , then a setting of the option <code>na.action</code> , or <code>na.fail</code> if that is not set.

**Details**

If `formula` is a data frame and `data` is missing, `formula` is used for `data` with the formula `~ ..`

**Value**

A vector or list with one entry for each term in the formula giving the number(s) of replications for each level. If all levels are balanced (have the same number of replications) the result is a vector, otherwise it is a list with a component for each terms, as a vector, matrix or array as required.

A test for balance is `!is.list(replications(formula, data))`.

**Author(s)**

The design was inspired by the S function of the same name described in Chambers *et al.* (1992).

**References**

Chambers, J. M., Freeny, A and Heiberger, R. M. (1992) *Analysis of variance; designed experiments*. Chapter 5 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[model.tables](#)

**Examples**

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)
replications(~ . - yield, npk)
```

---

 reshape

*Reshape Grouped Data*


---

**Description**

This function reshapes a data frame between ‘wide’ format with repeated measurements in separate columns of the same record and ‘long’ format with the repeated measurements in separate records.

**Usage**

```
reshape(data, varying = NULL, v.names = NULL, timevar = "time",
        idvar = "id", ids = 1:NROW(data),
        times = seq(length = length(varying[[1]])),
        drop = NULL, direction, new.row.names = NULL,
        split = list(regexp="\\.\\.", include=FALSE))
```

**Arguments**

<code>data</code>	a data frame
<code>varying</code>	names of sets of variables in the wide format that correspond to single variables in long format (‘time-varying’). A list of vectors (or optionally a matrix for <code>direction="wide"</code> ). See below for more details and options.
<code>v.names</code>	names of variables in the long format that correspond to multiple variables in the wide format.

<code>timevar</code>	the variable in long format that differentiates multiple records from the same group or individual.
<code>idvar</code>	Names of one or more variables in long format that identify multiple records from the same group/individual. These variables may also be present in wide format
<code>ids</code>	the values to use for a newly created <code>idvar</code> variable in long format.
<code>times</code>	the values to use for a newly created <code>timevar</code> variable in long format.
<code>drop</code>	a vector of names of variables to drop before reshaping
<code>direction</code>	character string, either "wide" to reshape to wide format, or "long" to reshape to long format.
<code>new.row.names</code>	logical; if TRUE and <code>direction="wide"</code> , create new row names in long format from the values of the <code>id</code> and <code>time</code> variables.
<code>split</code>	information for guessing the <code>varying</code> , <code>v.names</code> , and <code>times</code> arguments. See below for details.

### Details

The arguments to this function are described in terms of longitudinal data, as that is the application motivating the functions. A 'wide' longitudinal dataset will have one record for each individual with some time-constant variables that occupy single columns and some time-varying variables that occupy a column for each time point. In 'long' format there will be multiple records for each individual, with some variables being constant across these records and others varying across the records. A 'long' format dataset also needs a 'time' variable identifying which time point each record comes from and an 'id' variable showing which records refer to the same person.

If the data frame resulted from a previous `reshape` then the operation can be reversed simply by `reshape(a)`. The `direction` argument is optional and the other arguments are stored as attributes on the data frame.

If `direction="long"` and no `varying` or `v.names` arguments are supplied it is assumed that all variables except `idvar` and `timevar` are time-varying. They are all expanded into multiple variables in wide format.

If `direction="wide"` the `varying` argument can be a vector of column names or column numbers (converted to column names). The function will attempt to guess the `v.names` and `times` from these names. The default is variable names like `x.1`, `x.2`, where `split=list(regex="\\.",include=FALSE)` to specifies to split at the dot and drop it from the name. To have alphabetic followed by numeric times use `split=list(regex="[A-Za-z][0-9]",include=TRUE)`. This splits between the alphabetic and numeric parts of the name and does not drop the regular expression.

### Value

The reshaped data frame with added attributes to simplify reshaping back to the original form.

### See Also

[stack](#), [aperm](#)

**Examples**

```

summary(Indometh)
wide <- reshape(Indometh, v.names="conc", idvar="Subject",
                timevar="time", direction="wide")

wide

reshape(wide, direction="long")
reshape(wide, idvar="Subject", varying=list(names(wide)[2:12]),
        v.names="conc", direction="long")

## times need not be numeric
df <- data.frame(id=rep(1:4,rep(2,4)), visit=I(rep(c("Before","After"),4)),
                x=rnorm(4), y=runif(4))

df
reshape(df, timevar="visit", idvar="id", direction="wide")
## warns that y is really varying
reshape(df, timevar="visit", idvar="id", direction="wide", v.names="x")

## unbalanced 'long' data leads to NA fill in 'wide' form
df2 <- df[1:7,]
df2
reshape(df2, timevar="visit", idvar="id", direction="wide")

## Alternative regular expressions for guessing names
df3 <- data.frame(id=1:4, age=c(40,50,60,50), dose1=c(1,2,1,2),
                 dose2=c(2,1,2,1), dose4=c(3,3,3,3))
reshape(df3, direction="long", varying=3:5,
        split=list(regexp="[a-z][0-9]", include=TRUE))

## an example that isn't longitudinal data
state.x77 <- as.data.frame(state.x77)
long <- reshape(state.x77, idvar="state", ids=row.names(state.x77),
               times=names(state.x77), timevar="Characteristic",
               varying=list(names(state.x77)), direction="long")

reshape(long, direction="wide")

reshape(long, direction="wide", new.row.names=unique(long$state))

## multiple id variables
df3 <- data.frame(school=rep(1:3,each=4), class=rep(9:10,6), time=rep(c(1,1,2,2),3),
                 score=rnorm(12))
wide <- reshape(df3, idvar=c("school","class"), direction="wide")
wide
## transform back
reshape(wide)

```

**Description**

`residuals` is a generic function which extracts model residuals from objects returned by modeling functions.

The abbreviated form `resid` is an alias for `residuals`. It is intended to encourage users to access object components through an accessor function rather than by directly referencing an object slot.

All object classes which are returned by model fitting functions should provide a `residuals` method. (Note that the method is ‘`residuals`’ and not ‘`resid`’.)

Methods can make use of `naresid` methods to compensate for the omission of missing values. The default method does.

**Usage**

```
residuals(object, ...)  
resid(object, ...)
```

**Arguments**

`object`            an object for which the extraction of model residuals is meaningful.  
`...`             other arguments.

**Value**

Residuals extracted from the object `object`.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

**See Also**

`coefficients`, `fitted.values`, `glm`, `lm`.

---

runmed

*Running Medians – Robust Scatter Plot Smoothing*

---

**Description**

Compute running medians of odd span. This is the “most robust” scatter plot smoothing possible. For efficiency (and historical reason), you can use one of two different algorithms giving identical results.

**Usage**

```
runmed(x, k, endrule = c("median", "keep", "constant"),  
      algorithm = NULL, print.level = 0)
```

**Arguments**

<code>x</code>	numeric vector, the “dependent” variable to be smoothed.
<code>k</code>	integer width of median window; must be odd. Turlach had a default of $k \leftarrow 1 + 2 * \min((n-1) \% \% 2, \text{ceiling}(0.1*n))$ . Use $k = 3$ for “minimal” robust smoothing eliminating isolated outliers.
<code>endrule</code>	character string indicating how the values at the beginning and the end (of the data) should be treated.  <b>"keep"</b> keeps the first and last $k_2$ values at both ends, where $k_2$ is the half-bandwidth $k_2 = k \% \% 2$ , i.e., $y[j] = x[j]$ for $j \in \{1, \dots, k_2; n - k_2 + 1, \dots, n\}$ ; <b>"constant"</b> copies $\text{median}(y[1:k_2])$ to the first values and analogously for the last ones making the smoothed ends <i>constant</i> ; <b>"median"</b> the default, smoothes the ends by using symmetrical medians of subsequently smaller bandwidth, but for the very first and last value where Tukey’s robust end-point rule is applied, see <a href="#">smoothEnds</a> .
<code>algorithm</code>	character string (partially matching "Turlach" or "Stuetzle") or the default NULL, specifying which algorithm should be applied. The default choice depends on $n = \text{length}(x)$ and $k$ where "Turlach" will be used for larger problems.
<code>print.level</code>	integer, indicating verbosity of algorithm; should rarely be changed by average users.

**Details**

Apart from the end values, the result  $y = \text{runmed}(x, k)$  simply has  $y[j] = \text{median}(x[(j-k_2):(j+k_2)])$  ( $k = 2*k_2+1$ ), computed very efficiently.

The two algorithms are internally entirely different:

**"Turlach"** is the Härdle-Steiger algorithm (see Ref.) as implemented by Berwin Turlach. A tree algorithm is used, ensuring performance  $O(n \log k)$  where  $n \leftarrow \text{length}(x)$  which is asymptotically optimal.

**"Stuetzle"** is the (older) Stuetzle-Friedman implementation which makes use of median *updating* when one observation enters and one leaves the smoothing window. While this performs as  $O(n \times k)$  which is slower asymptotically, it is considerably faster for small  $k$  or  $n$ .

**Value**

vector of smoothed values of the same length as  $x$  with an `attribute` `k` containing (the ‘oddified’)  $k$ .

**Author(s)**

Martin Maechler ([maechler@stat.math.ethz.ch](mailto:maechler@stat.math.ethz.ch)), based on Fortran code from Werner Stuetzle and S-plus and C code from Berwin Turlach.

## References

Härdle, W. and Steiger, W. (1995) [Algorithm AS 296] Optimal median smoothing, *Applied Statistics* **44**, 258–264.

Jerome H. Friedman and Werner Stuetzle (1982) *Smoothing of Scatterplots*; Report, Dep. Statistics, Stanford U., Project Orion 003.

Martin Maechler (2003) Fast Running Medians: Finite Sample and Asymptotic Optimality; working paper available from the author.

## See Also

[smoothEnds](#) which implements Tukey's end point rule and is called by default from `runmed(*, endrule = "median")`. [smooth](#) uses running medians of 3 for its compound smoothers.

## Examples

```
example(nhtemp)
myNHT <- as.vector(nhtemp)
myNHT[20] <- 2 * nhtemp[20]
plot(myNHT, type="b", ylim = c(48,60), main = "Running Medians Example")
lines(runmed(myNHT, 7), col = "red")

## special: multiple y values for one x
plot(cars, main = "'cars' data and runmed(dist, 3)")
lines(cars, col = "light gray", type = "c")
with(cars, lines(speed, runmed(dist, k = 3), col = 2))

## nice quadratic with a few outliers
y <- ys <- (-20:20)^2
y [c(1,10,21,41)] <- c(150, 30, 400, 450)
all(y == runmed(y, 1)) # 1-neighborhood <==> interpolation
plot(y) ## lines(y, lwd=.1, col="light gray")
lines(lowess(seq(y),y, f = .3), col = "brown")
lines(runmed(y, 7), lwd=2, col = "blue")
lines(runmed(y,11), lwd=2, col = "red")

## Lowess is not robust
y <- ys ; y[21] <- 6666 ; x <- seq(y)
col <- c("black", "brown", "blue")
plot(y, col=col[1])
lines(lowess(x,y, f = .3), col = col[2])
lines(runmed(y, 7), lwd=2, col = col[3])
legend(length(y),max(y), c("data", "lowess(y, f = 0.3)", "runmed(y, 7)"),
       xjust = 1, col = col, lty = c(0, 1,1), pch = c(1,NA,NA))
```

**Description**

Plot and add a smooth curve computed by loess to a scatter plot.

**Usage**

```
scatter.smooth(x, y, span = 2/3, degree = 1,  
              family = c("symmetric", "gaussian"),  
              xlab = deparse(substitute(x)), ylab = deparse(substitute(y)),  
              ylim = range(y, prediction$y), evaluation = 50, ...)
```

```
loess.smooth(x, y, span = 2/3, degree = 1,  
             family = c("symmetric", "gaussian"), evaluation = 50, ...)
```

**Arguments**

x	x coordinates for scatter plot.
y	y coordinates for scatter plot.
span	smoothness parameter for loess.
degree	degree of local polynomial used.
family	if "gaussian" fitting is by least-squares, and if family="symmetric" a re-descending M estimator is used.
xlab	label for x axis.
ylab	label for y axis.
ylim	the y limits of the plot.
evaluation	number of points at which to evaluate the smooth curve.
...	graphical parameters.

**Details**

loess.smooth is an auxiliary function which evaluates the loess smooth at evaluation equally spaced points covering the range of x.

**Value**

For scatter.smooth, none.

For loess.smooth, a list with two components, x (the grid of evaluation points) and y (the smoothed values at the grid points).

**See Also**

[loess](#)

**Examples**

```
attach(cars)  
scatter.smooth(speed, dist)  
detach()
```

---

`screepplot`*Screepplot of PCA Results*

---

**Description**

`screepplot` plots the variances against the number of the principal component. This is also the plot method for class "princomp".

**Usage**

```
screepplot(x, npcs = min(10, length(x$sdev)),
           type = c("barplot", "lines"),
           main = deparse(substitute(x)), ...)
```

**Arguments**

`x` an object of class "princomp", as from `princomp()`.  
`npcs` the number of principal components to be plotted.  
`type` the type of plot.  
`main, ...` graphics parameters.

**References**

Mardia, K. V., J. T. Kent and J. M. Bibby (1979). *Multivariate Analysis*, London: Academic Press.  
Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*, Springer-Verlag.

**See Also**

[princomp](#).

**Examples**

```
## The variances of the variables in the
## USArrests data vary by orders of magnitude, so scaling is appropriate
(pc.cr <- princomp(USArrests, cor = TRUE)) # inappropriate
screepplot(pc.cr)

fit <- princomp(covmat=Harman74.cor)
screepplot(fit)
screepplot(fit, npcs=24, type="lines")
```

---

sd	<i>Standard Deviation</i>
----	---------------------------

---

### Description

This function computes the standard deviation of the values in `x`. If `na.rm` is `TRUE` then missing values are removed before computation proceeds. If `x` is a matrix or a data frame, a vector of the standard deviation of the columns is returned.

### Usage

```
sd(x, na.rm = FALSE)
```

### Arguments

`x` a numeric vector, matrix or data frame.  
`na.rm` logical. Should missing values be removed?

### See Also

[var](#) for its square, and [mad](#), the most robust alternative.

### Examples

```
sd(1:2) ^ 2
```

---

se.contrast	<i>Standard Errors for Contrasts in Model Terms</i>
-------------	---

---

### Description

Returns the standard errors for one or more contrasts in an `aov` object.

### Usage

```
se.contrast(object, ...)  
## S3 method for class 'aov':  
se.contrast(object, contrast.obj,  
             coef = contr.helmert(ncol(contrast))[ , 1],  
             data = NULL, ...)
```

**Arguments**

<code>object</code>	A suitable fit, usually from <code>aov</code> .
<code>contrast.obj</code>	The contrasts for which standard errors are requested. This can be specified via a list or via a matrix. A single contrast can be specified by a list of logical vectors giving the cells to be contrasted. Multiple contrasts should be specified by a matrix, each column of which is a numerical contrast vector (summing to zero).
<code>coef</code>	used when <code>contrast.obj</code> is a list; it should be a vector of the same length as the list with zero sum. The default value is the first Helmert contrast, which contrasts the first and second cell means specified by the list.
<code>data</code>	The data frame used to evaluate <code>contrast.obj</code> .
<code>...</code>	further arguments passed to or from other methods.

**Details**

Contrasts are usually used to test if certain means are significantly different; it can be easier to use `se.contrast` than compute them directly from the coefficients.

In multistratum models, the contrasts can appear in more than one stratum; the contrast and standard error are computed in the lowest stratum and adjusted for efficiencies and comparisons between strata.

Suitable matrices for use with `coef` can be found by calling `contrasts` and indexing the columns by a factor.

**Value**

A vector giving the standard errors for each contrast.

**See Also**

[contrasts](#), [model.tables](#)

**Examples**

```
## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,
55.0, 62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)

npk <- data.frame(block = gl(6,4), N = factor(N), P = factor(P),
                  K = factor(K), yield = yield)
options(contrasts=c("contr.treatment", "contr.poly"))
npk.aov1 <- aov(yield ~ block + N + K, npk)
se.contrast(npk.aov1, list(N=="0", N=="1"), data=npk)
# or via a matrix
cont <- matrix(c(-1,1), 2, 1, dimnames=list(NULL, "N"))
se.contrast(npk.aov1, cont[N, , drop=FALSE]/12, data=npk)
```

```
## test a multi-stratum model
npk.aov2 <- aov(yield ~ N + K + Error(block/(N + K)), npk)
se.contrast(npk.aov2, list(N == "0", N == "1"))
```

---

selfStart

*Construct Self-starting Nonlinear Models*


---

### Description

Construct self-starting nonlinear models.

### Usage

```
selfStart(model, initial, parameters, template)
```

### Arguments

model	a function object defining a nonlinear model or a nonlinear formula object of the form <code>~expression</code> .
initial	a function object, taking three arguments: <code>mCall</code> , <code>data</code> , and <code>LHS</code> , representing, respectively, a matched call to the function <code>model</code> , a data frame in which to interpret the variables in <code>mCall</code> , and the expression from the left-hand side of the model formula in the call to <code>nls</code> . This function should return initial values for the parameters in <code>model</code> .
parameters	a character vector specifying the terms on the right hand side of <code>model</code> for which initial estimates should be calculated. Passed as the <code>namevec</code> argument to the <code>deriv</code> function.
template	an optional prototype for the calling sequence of the returned object, passed as the <code>function.arg</code> argument to the <code>deriv</code> function. By default, a template is generated with the covariates in <code>model</code> coming first and the parameters in <code>model</code> coming last in the calling sequence.

### Details

This function is generic; methods functions can be written to handle specific classes of objects.

### Value

a function object of class `"selfStart"`, for the `formula` method obtained by applying `deriv` to the right hand side of the `model` formula. An `initial` attribute (defined by the `initial` argument) is added to the function to calculate starting estimates for the parameters in the model automatically.

### Author(s)

Jose Pinheiro and Douglas Bates

**See Also**[nls](#)**Examples**

```
## self-starting logistic model

SSlogis <- selfStart(~ Asym/(1 + exp((xmid - x)/scal)),
  function(mCall, data, LHS)
  {
    xy <- sortedXyData(mCall[["x"]], LHS, data)
    if(nrow(xy) < 4) {
      stop("Too few distinct x values to fit a logistic")
    }
    z <- xy[["y"]]
    if (min(z) <= 0) { z <- z + 0.05 * max(z) } # avoid zeroes
    z <- z/(1.05 * max(z)) # scale to within unit height
    xy[["z"]] <- log(z/(1 - z)) # logit transformation
    aux <- coef(lm(x ~ z, xy))
    parameters(xy) <- list(xmid = aux[1], scal = aux[2])
    pars <- as.vector(coef(nls(y ~ 1/(1 + exp((xmid - x)/scal)),
      data = xy, algorithm = "plinear")))
    value <- c(pars[3], pars[1], pars[2])
    names(value) <- mCall[c("Asym", "xmid", "scal")]
    value
  }, c("Asym", "xmid", "scal"))

# 'first.order.log.model' is a function object defining a first order
# compartment model
# 'first.order.log.initial' is a function object which calculates initial
# values for the parameters in 'first.order.log.model'

# self-starting first order compartment model
## Not run:
SSfol <- selfStart(first.order.log.model, first.order.log.initial)
## End(Not run)
```

setNames

*Set the Names in an Object***Description**

This is a convenience function that sets the names on an object and returns the object. It is most useful at the end of a function definition where one is creating the object to be returned and would prefer not to store it under a name just so the names can be assigned.

**Usage**

```
setNames(object, nm)
```

**Arguments**

`object` an object for which a names attribute will be meaningful  
`nm` a character vector of names to assign to the object

**Value**

An object of the same sort as `object` with the new names assigned.

**Author(s)**

Douglas M. Bates and Saikat DebRoy

**See Also**

[clearNames](#)

**Examples**

```
setNames( 1:3, c("foo", "bar", "baz") )
# this is just a short form of
tmp <- 1:3
names(tmp) <- c("foo", "bar", "baz")
tmp
```

---

shapiro.test                      *Shapiro-Wilk Normality Test*

---

**Description**

Performs the Shapiro-Wilk test for normality.

**Usage**

```
shapiro.test(x)
```

**Arguments**

`x` a numeric vector of data values, the number of which must be between 3 and 5000. Missing values are allowed.

**Value**

A list with class "htest" containing the following components:

`statistic` the value of the Shapiro-Wilk statistic.  
`p.value` the p-value for the test.  
`method` the character string "Shapiro-Wilk normality test".  
`data.name` a character string giving the name(s) of the data.

## References

- Patrick Royston (1982) An Extension of Shapiro and Wilk's  $W$  Test for Normality to Large Samples. *Applied Statistics*, **31**, 115–124.
- Patrick Royston (1982) Algorithm AS 181: The  $W$  Test for Normality. *Applied Statistics*, **31**, 176–180.
- Patrick Royston (1995) A Remark on Algorithm AS 181: The  $W$  Test for Normality. *Applied Statistics*, **44**, 547–551.

## See Also

[qqnorm](#) for producing a normal quantile-quantile plot.

## Examples

```
shapiro.test(rnorm(100, mean = 5, sd = 3))
shapiro.test(runif(100, min = 2, max = 4))
```

---

SignRank

*Distribution of the Wilcoxon Signed Rank Statistic*

---

## Description

Density, distribution function, quantile function and random generation for the distribution of the Wilcoxon Signed Rank statistic obtained from a sample with size  $n$ .

## Usage

```
dsignrank(x, n, log = FALSE)
psignrank(q, n, lower.tail = TRUE, log.p = FALSE)
qsignrank(p, n, lower.tail = TRUE, log.p = FALSE)
rsignrank(nn, n)
```

## Arguments

- |                         |  |
|-------------------------|--|
| <code>x, q</code>       | vector of quantiles.   |
| <code>p</code>          | vector of probabilities.   |
| <code>nn</code>         | number of observations. If <code>length(nn) &gt; 1</code> , the length is taken to be the number required. |
| <code>n</code>          | number(s) of observations in the sample(s). A positive integer, or a vector of such integers.              |
| <code>log, log.p</code> | logical; if TRUE, probabilities <code>p</code> are given as $\log(p)$ .                                    |
| <code>lower.tail</code> | logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .                      |

**Details**

This distribution is obtained as follows. Let  $x$  be a sample of size  $n$  from a continuous distribution symmetric about the origin. Then the Wilcoxon signed rank statistic is the sum of the ranks of the absolute values  $x[i]$  for which  $x[i]$  is positive. This statistic takes values between 0 and  $n(n+1)/2$ , and its mean and variance are  $n(n+1)/4$  and  $n(n+1)(2n+1)/24$ , respectively.

If either of the first two arguments is a vector, the recycling rule is used to do the calculations for all combinations of the two up to the length of the longer vector.

**Value**

`dsignrank` gives the density, `psignrank` gives the distribution function, `qsignrank` gives the quantile function, and `rsignrank` generates random deviates.

**Author(s)**

Kurt Hornik

**See Also**

[wilcox.test](#) to calculate the statistic from data, find p values and so on.

[dwilcox](#) etc, for the distribution of *two-sample* Wilcoxon rank sum statistic.

**Examples**

```
par(mfrow=c(2,2))
for(n in c(4:5,10,40)) {
  x <- seq(0, n*(n+1)/2, length=501)
  plot(x, dsignrank(x,n=n), type='l', main=paste("dsignrank(x,n=",n,")"))
}
```

---

smooth

*Tukey's (Running Median) Smoothing*

---

**Description**

Tukey's smoothers, *3RS3R*, *3RSS*, *3R*, etc.

**Usage**

```
smooth(x, kind = c("3RS3R", "3RSS", "3RSR", "3R", "3", "S"),
       twiceit = FALSE, endrule = "Tukey", do.ends = FALSE)
```

**Arguments**

<code>x</code>	a vector or time series
<code>kind</code>	a character string indicating the kind of smoother required; defaults to "3RS3R".
<code>twiceit</code>	logical, indicating if the result should be “twiced”. Twicing a smoother $S(y)$ means $S(y) + S(y - S(y))$ , i.e., adding smoothed residuals to the smoothed values. This decreases bias (increasing variance).
<code>endrule</code>	a character string indicating the rule for smoothing at the boundary. Either "Tukey" (default) or "copy".
<code>do.ends</code>	logical, indicating if the 3-splitting of ties should also happen at the boundaries (“ends”). This is only used for <code>kind = "S"</code> .

**Details**

3 is Tukey’s short notation for running [medians](#) of length 3, 3R stands for **R**epeated 3 until convergence, and S for **S**plitting of horizontal stretches of length 2 or 3.

Hence, 3RS3R is a concatenation of 3R, S and 3R, 3RSS similarly, whereas 3RSR means first 3R and then (S and 3) **R**epeated until convergence – which can be bad.

**Value**

An object of class "tukeysmooth" (which has `print` and `summary` methods) and is a vector or time series containing the smoothed values with additional attributes.

**Note**

S and S-PLUS use a different (somewhat better) Tukey smoother in `smooth(*)`. Note that there are other smoothing methods which provide rather better results. These were designed for hand calculations and may be used mainly for didactical purposes.

Since R version 1.2, `smooth` *does* really implement Tukey’s end-point rule correctly (see argument `endrule`).

`kind = "3RSR"` has been the default till R-1.1, but it can have very bad properties, see the examples.

Note that repeated application of `smooth(*)` *does* smooth more, for the "3RS\*" kinds.

**References**

Tukey, J. W. (1977). *Exploratory Data Analysis*, Reading Massachusetts: Addison-Wesley.

**See Also**

[lowess](#); [loess](#), [supsmu](#) and [smooth.spline](#).

**Examples**

```
## see also demo(smooth) !

x1 <- c(4, 1, 3, 6, 6, 4, 1, 6, 2, 4, 2) # very artificial
(x3R <- smooth(x1, "3R")) # 2 iterations of "3"
smooth(x3R, kind = "S")

sm.3RS <- function(x, ...)
  smooth(smooth(x, "3R", ...), "S", ...)

y <- c(1,1, 19:1)
plot(y, main = "misbehaviour of \"3RSR\"", col.main = 3)
lines(sm.3RS(y))
lines(smooth(y))
lines(smooth(y, "3RSR"), col = 3, lwd = 2)# the horror

x <- c(8:10,10, 0,0, 9,9)
plot(x, main = "breakdown of 3R and S and hence 3RSS")
matlines(cbind(smooth(x,"3R"),smooth(x,"S"), smooth(x,"3RSS"),smooth(x)))

presidents[is.na(presidents)] <- 0 # silly
summary(sm3 <- smooth(presidents, "3R"))
summary(sm2 <- smooth(presidents,"3RSS"))
summary(sm <- smooth(presidents))

all.equal(c(sm2),c(smooth(smooth(sm3, "S"), "S"))) # 3RSS === 3R S S
all.equal(c(sm), c(smooth(smooth(sm3, "S"), "3R")))# 3RS3R === 3R S 3R

plot(presidents, main = "smooth(presidents0, *) : 3R and default 3RS3R")
lines(sm3,col = 3, lwd = 1.5)
lines(sm, col = 2, lwd = 1.25)
```

smooth.spline

*Fit a Smoothing Spline***Description**

Fits a cubic smoothing spline to the supplied data.

**Usage**

```
smooth.spline(x, y = NULL, w = NULL, df, spar = NULL,
              cv = FALSE, all.knots = FALSE, nknots = NULL,
              df.offset = 0, penalty = 1, control.spar = list())
```

**Arguments**

**x** a vector giving the values of the predictor variable, or a list or a two-column matrix specifying x and y.

<code>y</code>	responses. If <code>y</code> is missing, the responses are assumed to be specified by <code>x</code> .
<code>w</code>	optional vector of weights of the same length as <code>x</code> ; defaults to all 1.
<code>df</code>	the desired equivalent number of degrees of freedom (trace of the smoother matrix).
<code>spar</code>	smoothing parameter, typically (but not necessarily) in $(0, 1]$ . The coefficient $\lambda$ of the integral of the squared second derivative in the fit (penalized log likelihood) criterion is a monotone function of <code>spar</code> , see the details below.
<code>cv</code>	ordinary (TRUE) or “generalized” cross-validation (GCV) when FALSE.
<code>all.knots</code>	if TRUE, all distinct points in <code>x</code> are used as knots. If FALSE (default), a subset of <code>x[ ]</code> is used, specifically <code>x[ j ]</code> where the <code>nknots</code> indices are evenly spaced in $1:n$ , see also the next argument <code>nknots</code> .
<code>nknots</code>	integer giving the number of knots to use when <code>all.knots=FALSE</code> . Per default, this is less than $n$ , the number of unique <code>x</code> values for $n > 49$ .
<code>df.offset</code>	allows the degrees of freedom to be increased by <code>df.offset</code> in the GCV criterion.
<code>penalty</code>	the coefficient of the penalty for degrees of freedom in the GCV criterion.
<code>control.spar</code>	optional list with named components controlling the root finding when the smoothing parameter <code>spar</code> is computed, i.e., missing or NULL, see below. <b>Note</b> that this is partly <i>experimental</i> and may change with general <code>spar</code> computation improvements! <b>low:</b> lower bound for <code>spar</code> ; defaults to -1.5 (used to implicitly default to 0 in R versions earlier than 1.4). <b>high:</b> upper bound for <code>spar</code> ; defaults to +1.5. <b>tol:</b> the absolute precision ( <b>tolerance</b> ) used; defaults to 1e-4 (formerly 1e-3). <b>eps:</b> the relative precision used; defaults to 2e-8 (formerly 0.00244). <b>trace:</b> logical indicating if iterations should be traced. <b>maxit:</b> integer giving the maximal number of iterations; defaults to 500. Note that <code>spar</code> is only searched for in the interval $[low, high]$ .

## Details

The `x` vector should contain at least four distinct values. *Distinct* here means “distinct after rounding to 6 significant digits”, i.e., `x` will be transformed to `unique(sort(signif(x, 6)))`, and `y` and `w` are pooled accordingly.

The computational  $\lambda$  used (as a function of  $s = spar$ ) is  $\lambda = r * 256^{3s-1}$  where  $r = tr(X'WX)/tr(\Sigma)$ ,  $\Sigma$  is the matrix given by  $\Sigma_{ij} = \int B_i''(t)B_j''(t)dt$ ,  $X$  is given by  $X_{ij} = B_j(x_i)$ ,  $W$  is the diagonal matrix of weights (scaled such that its trace is  $n$ , the original number of observations) and  $B_k(\cdot)$  is the  $k$ -th B-spline.

Note that with these definitions,  $f_i = f(x_i)$ , and the B-spline basis representation  $f = Xc$  (i.e.,  $c$  is the vector of spline coefficients), the penalized log likelihood is  $L = (y - f)'W(y - f) + \lambda c'\Sigma c$ , and hence  $c$  is the solution of the (ridge regression)  $(X'WX + \lambda\Sigma)c = X'Wy$ .

If `spar` is missing or NULL, the value of `df` is used to determine the degree of smoothing. If both are missing, leave-one-out cross-validation (ordinary or “generalized” as determined by `cv`) is

used to determine  $\lambda$ . Note that from the above relation, `spar` is  $s = s_0 + 0.0601 * \log \lambda$ , which is intentionally *different* from the S-plus implementation of `smooth.spline` (where `spar` is proportional to  $\lambda$ ). In R's ( $\log \lambda$ ) scale, it makes more sense to vary `spar` linearly.

Note however that currently the results may become very unreliable for `spar` values smaller than about -1 or -2. The same may happen for values larger than 2 or so. Don't think of setting `spar` or the controls `low` and `high` outside such a safe range, unless you know what you are doing!

The "generalized" cross-validation method will work correctly when there are duplicated points in `x`. However, it is ambiguous what leave-one-out cross-validation means with duplicated points, and the internal code uses an approximation that involves leaving out groups of duplicated points. `cv=TRUE` is best avoided in that case.

### Value

An object of class "smooth.spline" with components

<code>x</code>	the <i>distinct</i> <code>x</code> values in increasing order, see the <b>Details</b> above.
<code>y</code>	the fitted values corresponding to <code>x</code> .
<code>w</code>	the weights used at the unique values of <code>x</code> .
<code>yin</code>	the <code>y</code> values used at the unique <code>y</code> values.
<code>lev</code>	leverages, the diagonal values of the smoother matrix.
<code>cv.crit</code>	cross-validation score, "generalized" or true, depending on <code>cv</code> .
<code>pen.crit</code>	penalized criterion
<code>crit</code>	the criterion value minimized in the underlying <code>.Fortran</code> routine 'sslvrg'.
<code>df</code>	equivalent degrees of freedom used. Note that (currently) this value may become quite unprecise when the true <code>df</code> is between and 1 and 2.
<code>spar</code>	the value of <code>spar</code> computed or given.
<code>lambda</code>	the value of $\lambda$ corresponding to <code>spar</code> , see the details above.
<code>iparms</code>	named integer(3) vector where <code>..\$ipars["iter"]</code> gives number of <code>spar</code> computing iterations used.
<code>fit</code>	list for use by <code>predict.smooth.spline</code> , with components <b>knot</b> : the knot sequence (including the repeated boundary knots). <b>nk</b> : number of coefficients or number of "proper" knots plus 2. <b>coef</b> : coefficients for the spline basis used. <b>min, range</b> : numbers giving the corresponding quantities of <code>x</code> .
<code>call</code>	the matched call.

### Note

The default `all.knots = FALSE` and `nknots = NULL` entails using only  $O(n^{0.2})$  knots instead of  $n$  for  $n > 49$ . This cuts speed and memory requirements, but not drastically anymore since R version 1.5.1 where it is only  $O(n_k) + O(n)$  where  $n_k$  is the number of knots. In this case where not all unique `x` values are used as knots, the result is not a smoothing spline in the strict sense, but very close unless a small smoothing parameter (or large `df`) is used.

**Author(s)**

R implementation by B. D. Ripley and Martin Maechler (`spar/lambda`, etc).

This function is based on code in the GAMFIT Fortran program by T. Hastie and R. Tibshirani (<http://lib.stat.cmu.edu/general/>), which makes use of spline code by Finbarr O'Sullivan. Its design parallels the `smooth.spline` function of Chambers & Hastie (1992).

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth & Brooks/Cole.

Green, P. J. and Silverman, B. W. (1994) *Nonparametric Regression and Generalized Linear Models: A Roughness Penalty Approach*. Chapman and Hall.

Hastie, T. J. and Tibshirani, R. J. (1990) *Generalized Additive Models*. Chapman and Hall.

**See Also**

[predict.smooth.spline](#) for evaluating the spline and its derivatives.

**Examples**

```
attach(cars)
plot(speed, dist, main = "data(cars) & smoothing splines")
cars.spl <- smooth.spline(speed, dist)
(cars.spl)
## This example has duplicate points, so avoid cv=TRUE

lines(cars.spl, col = "blue")
lines(smooth.spline(speed, dist, df=10), lty=2, col = "red")
legend(5,120,c(paste("default [C.V.] => df =",round(cars.spl$df,1)),
               "s( * , df = 10)"), col = c("blue","red"), lty = 1:2,
      bg='bisque')
detach()

##-- artificial example
y18 <- c(1:3,5,4,7:3,2*(2:5),rep(10,4))
xx <- seq(1,length(y18), len=201)
(s2 <- smooth.spline(y18)) # GCV
(s02 <- smooth.spline(y18, spar = 0.2))
plot(y18, main=deparse(s2$call), col.main=2)
lines(s2, col = "gray"); lines(predict(s2, xx), col = 2)
lines(predict(s02, xx), col = 3); mtext(deparse(s02$call), col = 3)

## The following shows the problematic behavior of 'spar' searching:
(s2 <- smooth.spline(y18, con=list(trace=TRUE,tol=1e-6, low= -1.5)))
(s2m <- smooth.spline(y18, cv=TRUE, con=list(trace=TRUE,tol=1e-6, low= -1.5)))
## both above do quite similarly (Df = 8.5 +- 0.2)
```

---

`smoothEnds`*End Points Smoothing (for Running Medians)*

---

**Description**

Smooth end points of a vector  $y$  using subsequently smaller medians and Tukey's end point rule at the very end. (of odd span),

**Usage**

```
smoothEnds(y, k = 3)
```

**Arguments**

<code>y</code>	dependent variable to be smoothed (vector).
<code>k</code>	width of largest median window; must be odd.

**Details**

`smoothEnds` is used to only do the "end point smoothing", i.e., change at most the observations closer to the beginning/end than half the window  $k$ . The first and last value are computed using "Tukey's end point rule", i.e.,  $sm[1] = \text{median}(y[1], sm[2], 3*sm[2] - 2*sm[3])$ .

**Value**

vector of smoothed values, the same length as  $y$ .

**Author(s)**

Martin Maechler

**References**

John W. Tukey (1977) *Exploratory Data Analysis*, Addison.

Velleman, P.F., and Hoaglin, D.C. (1981) *ABC of EDA (Applications, Basics, and Computing of Exploratory Data Analysis)*; Duxbury.

**See Also**

[runmed](#)(\*, end.rule = "median") which calls `smoothEnds()`.

**Examples**

```

y <- ys <- (-20:20)^2
y [c(1,10,21,41)] <- c(100, 30, 400, 470)
s7k <- runmed(y,7, end = "keep")
s7. <- runmed(y,7, end = "const")
s7m <- runmed(y,7)
col3 <- c("midnightblue","blue","steelblue")
plot(y, main = "Running Medians -- runmed(*, k=7, end.rule = X)")
lines(ys, col = "light gray")
matlines(cbind(s7k,s7.,s7m), lwd= 1.5, lty = 1, col = col3)
legend(1,470, paste("end.rule",c("keep","constant","median"),sep=" = "),
      col = col3, lwd = 1.5, lty = 1)

stopifnot(identical(s7m, smoothEnds(s7k, 7)))

```

---

sortedXyData                      *Create a sortedXyData object*

---

**Description**

This is a constructor function for the class of `sortedXyData` objects. These objects are mostly used in the `initial` function for a self-starting nonlinear regression model, which will be of the `selfStart` class.

**Usage**

```
sortedXyData(x, y, data)
```

**Arguments**

<code>x</code>	a numeric vector or an expression that will evaluate in <code>data</code> to a numeric vector
<code>y</code>	a numeric vector or an expression that will evaluate in <code>data</code> to a numeric vector
<code>data</code>	an optional data frame in which to evaluate expressions for <code>x</code> and <code>y</code> , if they are given as expressions

**Value**

A `sortedXyData` object. This is a data frame with exactly two numeric columns, named `x` and `y`. The rows are sorted so the `x` column is in increasing order. Duplicate `x` values are eliminated by averaging the corresponding `y` values.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[selfStart](#), [NLSstClosestX](#), [NLSstLfAsymptote](#), [NLSstRtAsymptote](#)

**Examples**

```
DNase.2 <- DNase[ DNase$Run == "2", ]
sortedXyData( expression(log(conc)), expression(density), DNase.2 )
```

spec.ar

*Estimate Spectral Density of a Time Series from AR Fit***Description**

Fits an AR model to `x` (or uses the existing fit) and computes (and by default plots) the spectral density of the fitted model.

**Usage**

```
spec.ar(x, n.freq, order = NULL, plot = TRUE, na.action = na.fail,
        method = "yule-walker", ...)
```

**Arguments**

<code>x</code>	A univariate (not yet:or multivariate) time series or the result of a fit by <a href="#">ar</a> .
<code>n.freq</code>	The number of points at which to plot.
<code>order</code>	The order of the AR model to be fitted. If omitted, the order is chosen by AIC.
<code>plot</code>	Plot the periodogram?
<code>na.action</code>	NA action function.
<code>method</code>	method for ar fit.
<code>...</code>	Graphical arguments passed to <a href="#">plot.spec</a> .

**Value**

An object of class "spec". The result is returned invisibly if `plot` is true.

**Warning**

Some authors, for example Thomson (1990), warn strongly that AR spectra can be misleading.

**Note**

The multivariate case is not yet implemented.

**References**

Thompson, D.J. (1990) Time series analysis of Holocene climate data. *Phil. Trans. Roy. Soc. A* **330**, 601–616.

Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer. (Especially page 402.)

**See Also**

[ar](#), [spectrum](#).

**Examples**

```
spec.ar(lh)

spec.ar(ldeaths)
spec.ar(ldeaths, method="burg")
```

---

spec.pgram	<i>Estimate Spectral Density of a Time Series by a Smoothed Periodogram</i>
------------	---

---

**Description**

`spec.pgram` calculates the periodogram using a fast Fourier transform, and optionally smooths the result with a series of modified Daniell smoothers (moving averages giving half weight to the end values).

**Usage**

```
spec.pgram(x, spans = NULL, kernel, taper = 0.1,
           pad = 0, fast = TRUE, demean = FALSE, detrend = TRUE,
           plot = TRUE, na.action = na.fail, ...)
```

**Arguments**

<code>x</code>	univariate or multivariate time series.
<code>spans</code>	vector of odd integers giving the widths of modified Daniell smoothers to be used to smooth the periodogram.
<code>kernel</code>	alternatively, a kernel smoother of class "tskernel".
<code>taper</code>	proportion of data to taper. A split cosine bell taper is applied to this proportion of the data at the beginning and end of the series.
<code>pad</code>	proportion of data to pad. Zeros are added to the end of the series to increase its length by the proportion <code>pad</code> .
<code>fast</code>	logical; if TRUE, pad the series to a highly composite length.
<code>demean</code>	logical. If TRUE, subtract the mean of the series.
<code>detrend</code>	logical. If TRUE, remove a linear trend from the series. This will also remove the mean.
<code>plot</code>	plot the periodogram?
<code>na.action</code>	NA action function.
<code>...</code>	graphical arguments passed to <code>plot.spec</code> .

**Details**

The raw periodogram is not a consistent estimator of the spectral density, but adjacent values are asymptotically independent. Hence a consistent estimator can be derived by smoothing the raw periodogram, assuming that the spectral density is smooth.

The series will be automatically padded with zeros until the series length is a highly composite number in order to help the Fast Fourier Transform. This is controlled by the `fast` and not the `pad` argument.

The periodogram at zero is in theory zero as the mean of the series is removed (but this may be affected by tapering): it is replaced by an interpolation of adjacent values during smoothing, and no value is returned for that frequency.

**Value**

A list object of class "spec" (see [spectrum](#)) with the following additional components:

<code>kernel</code>	The <code>kernel</code> argument, or the kernel constructed from <code>spans</code> .
<code>df</code>	The distribution of the spectral density estimate can be approximated by a chi square distribution with <code>df</code> degrees of freedom.
<code>bandwidth</code>	The equivalent bandwidth of the kernel smoother as defined by Bloomfield (1976, page 201).
<code>taper</code>	The value of the <code>taper</code> argument.
<code>pad</code>	The value of the <code>pad</code> argument.
<code>detrend</code>	The value of the <code>detrend</code> argument.
<code>demean</code>	The value of the <code>demean</code> argument.

The result is returned invisibly if `plot` is true.

**Author(s)**

Originally Martyn Plummer; kernel smoothing by Adrian Trapletti, synthesis by B.D. Ripley

**References**

- Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction*. Wiley.
- Brockwell, P.J. and Davis, R.A. (1991) *Time Series: Theory and Methods*. Second edition. Springer.
- Venables, W.N. and Ripley, B.D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer. (Especially pp. 392–7.)

**See Also**

[spectrum](#), [spec.taper](#), [plot.spec](#), [fft](#)

**Examples**

```
## Examples from Venables & Ripley
spectrum(ldeaths)
spectrum(ldeaths, spans = c(3,5))
spectrum(ldeaths, spans = c(5,7))
spectrum(mdeaths, spans = c(3,3))
spectrum(fdeaths, spans = c(3,3))

## bivariate example
mfdeaths.spc <- spec.pgram(ts.union(mdeaths, fdeaths), spans = c(3,3))
# plots marginal spectra: now plot coherency and phase
plot(mfdeaths.spc, plot.type = "coherency")
plot(mfdeaths.spc, plot.type = "phase")

## now impose a lack of alignment
mfdeaths.spc <- spec.pgram(ts.intersect(mdeaths, lag(fdeaths, 4)),
  spans = c(3,3), plot = FALSE)
plot(mfdeaths.spc, plot.type = "coherency")
plot(mfdeaths.spc, plot.type = "phase")

stocks.spc <- spectrum(EuStockMarkets, kernel("daniell", c(30,50)),
  plot = FALSE)
plot(stocks.spc, plot.type = "marginal") # the default type
plot(stocks.spc, plot.type = "coherency")
plot(stocks.spc, plot.type = "phase")

sales.spc <- spectrum(ts.union(BJsales, BJsales.lead),
  kernel("modified.daniell", c(5,7)))
plot(sales.spc, plot.type = "coherency")
plot(sales.spc, plot.type = "phase")
```

---

spec.taper

*Taper a Time Series by a Cosine Bell*


---

**Description**

Apply a cosine-bell taper to a time series.

**Usage**

```
spec.taper(x, p = 0.1)
```

**Arguments**

**x** A univariate or multivariate time series

**p** The total proportion to be tapered, either a scalar or a vector of the length of the number of series.

**Details**

The cosine-bell taper is applied to the first and last  $p[i]/2$  observations of time series  $x[, i]$ .

**Value**

A new time series object.

**Note**

From package **MASS**.

**Author(s)**

Kurt Hornik, B.D. Ripley

**See Also**

[spec.pgram](#), [cpgram](#)

---

spectrum

*Spectral Density Estimation*

---

**Description**

The `spectrum` function estimates the spectral density of a time series.

**Usage**

```
spectrum(x, method = c("pgram", "ar"), plot = TRUE,
         na.action = na.fail, ...)
```

**Arguments**

<code>x</code>	A univariate or multivariate time series.
<code>method</code>	String specifying the method used to estimate the spectral density. Allowed methods are "pgram" (the default) and "ar".
<code>plot</code>	logical. If TRUE then the spectral density is plotted.
<code>na.action</code>	NA action function.
<code>...</code>	Further arguments to specific spec methods or <code>plot.spec</code> .

**Details**

`spectrum` is a wrapper function which calls the methods `spec.pgram` and `spec.ar`.

The spectrum here is defined with scaling  $1/\text{frequency}(x)$ , following S-PLUS. This makes the spectral density a density over the range  $(-\text{frequency}(x)/2, +\text{frequency}(x)/2]$ , whereas a more common scaling is  $2\pi$  and range  $(-0.5, 0.5]$  (e.g., Bloomfield) or 1 and range  $(-\pi, \pi]$ .

If available, a confidence interval will be plotted by `plot.spec`: this is asymmetric, and the width of the centre mark indicates the equivalent bandwidth.

**Value**

An object of class "spec", which is a list containing at least the following components:

<code>freq</code>	vector of frequencies at which the spectral density is estimated. (Possibly approximate Fourier frequencies.) The units are the reciprocal of cycles per unit time (and not per observation spacing): see Details below.
<code>spec</code>	Vector (for univariate series) or matrix (for multivariate series) of estimates of the spectral density at frequencies corresponding to <code>freq</code> .
<code>coh</code>	NULL for univariate series. For multivariate time series, a matrix containing the <i>squared</i> coherency between different series. Column $i + (j - 1) * (j - 2)/2$ of <code>coh</code> contains the squared coherency between columns $i$ and $j$ of <code>x</code> , where $i < j$ .
<code>phase</code>	NULL for univariate series. For multivariate time series a matrix containing the cross-spectrum phase between different series. The format is the same as <code>coh</code> .
<code>series</code>	The name of the time series.
<code>snames</code>	For multivariate input, the names of the component series.
<code>method</code>	The method used to calculate the spectrum.

The result is returned invisibly if `plot` is true.

**Note**

The default plot for objects of class "spec" is quite complex, including an error bar and default title, subtitle and axis labels. The defaults can all be overridden by supplying the appropriate graphical parameters.

**Author(s)**

Martyn Plummer, B.D. Ripley

**References**

- Bloomfield, P. (1976) *Fourier Analysis of Time Series: An Introduction*. Wiley.
- Brockwell, P. J. and Davis, R. A. (1991) *Time Series: Theory and Methods*. Second edition. Springer.
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S-PLUS*. Fourth edition. Springer. (Especially pages 392–7.)

**See Also**

[spec.ar](#), [spec.pgram](#); [plot.spec](#).

**Examples**

```
## Examples from Venables & Ripley
## spec.pgram
par(mfrow=c(2,2))
spectrum(lh)
spectrum(lh, spans=3)
spectrum(lh, spans=c(3,3))
spectrum(lh, spans=c(3,5))

spectrum(ldeaths)
spectrum(ldeaths, spans=c(3,3))
spectrum(ldeaths, spans=c(3,5))
spectrum(ldeaths, spans=c(5,7))
spectrum(ldeaths, spans=c(5,7), log="dB", ci=0.8)

# for multivariate examples see the help for spec.pgram

## spec.ar
spectrum(lh, method="ar")
spectrum(ldeaths, method="ar")
```

---

splinefun

*Interpolating Splines*


---

**Description**

Perform cubic spline interpolation of given data points, returning either a list of points obtained by the interpolation or a function performing the interpolation.

**Usage**

```
splinefun(x, y = NULL, method = "fmm")

spline(x, y = NULL, n = 3*length(x), method = "fmm",
       xmin = min(x), xmax = max(x))
```

**Arguments**

<code>x, y</code>	vectors giving the coordinates of the points to be interpolated. Alternatively a single plotting structure can be specified: see <a href="#">xy.coords</a> .
<code>method</code>	specifies the type of spline to be used. Possible values are "fmm", "natural" and "periodic".
<code>n</code>	interpolation takes place at <code>n</code> equally spaced points spanning the interval <code>[xmin, xmax]</code> .

xmin            left-hand endpoint of the interpolation interval.  
 xmax            right-hand endpoint of the interpolation interval.

### Details

If `method = "fmm"`, the spline used is that of Forsythe, Malcolm and Moler (an exact cubic is fitted through the four points at each end of the data, and this is used to determine the end conditions). Natural splines are used when `method = "natural"`, and periodic splines when `method = "periodic"`.

These interpolation splines can also be used for extrapolation, that is prediction at points outside the range of `x`. Extrapolation makes little sense for `method = "fmm"`; for natural splines it is linear using the slope of the interpolating curve at the nearest data point.

### Value

`spline` returns a list containing components `x` and `y` which give the ordinates where interpolation took place and the interpolated values.

`splinefun` returns a function which will perform cubic spline interpolation of the given data points. This is often more useful than `spline`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

Forsythe, G. E., Malcolm, M. A. and Moler, C. B. (1977) *Computer Methods for Mathematical Computations*.

### See Also

[approx](#) and [approxfun](#) for constant and linear interpolation.

Package **splines**, especially [interpSpline](#) and [periodicSpline](#) for interpolation splines. That package also generates spline bases that can be used for regression splines.

[smooth.spline](#) in package **modreg** for smoothing splines.

### Examples

```
op <- par(mfrow = c(2,1), mgp = c(2,.8,0), mar = .1+c(3,3,3,1))
n <- 9
x <- 1:n
y <- rnorm(n)
plot(x, y, main = paste("spline[fun](.) through", n, "points"))
lines(spline(x, y))
lines(spline(x, y, n = 201), col = 2)

y <- (x-6)^2
plot(x, y, main = "spline(.) -- 3 methods")
lines(spline(x, y, n = 201), col = 2)
lines(spline(x, y, n = 201, method = "natural"), col = 3)
lines(spline(x, y, n = 201, method = "periodic"), col = 4)
```

```

legend(6,25, c("fmm","natural","periodic"), col=2:4, lty=1)

f <- splinefun(x, y)
ls(envir = environment(f))
splinecoef <- eval(expression(z), envir = environment(f))
curve(f(x), 1, 10, col = "green", lwd = 1.5)
points(splinecoef, col = "purple", cex = 2)
par(op)

```

---

SSasymp

*Asymptotic Regression Model*


---

### Description

This `selfStart` model evaluates the asymptotic regression function and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Asym`, `R0`, and `lrc` for a given set of data.

### Usage

```
SSasymp(input, Asym, R0, lrc)
```

### Arguments

<code>input</code>	a numeric vector of values at which to evaluate the model.
<code>Asym</code>	a numeric parameter representing the horizontal asymptote on the right side (very large values of <code>input</code> ).
<code>R0</code>	a numeric parameter representing the response when <code>input</code> is zero.
<code>lrc</code>	a numeric parameter representing the natural logarithm of the rate constant.

### Value

a numeric vector of the same length as `input`. It is the value of the expression  $Asym + (R0 - Asym) * \exp(-\exp(lrc) * input)$ . If all of the arguments `Asym`, `R0`, and `lrc` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

### Author(s)

Jose Pinheiro and Douglas Bates

### See Also

[nls](#), [selfStart](#)

**Examples**

```

Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
SSasymp( Lob.329$age, 100, -8.5, -3.2 ) # response only
Asym <- 100
resp0 <- -8.5
lrc <- -3.2
SSasymp( Lob.329$age, Asym, resp0, lrc ) # response and gradient
getInitial(height ~ SSasymp( age, Asym, resp0, lrc), data = Lob.329)
## Initial values are in fact the converged values
fml <- nls(height ~ SSasymp( age, Asym, resp0, lrc), data = Lob.329)
summary(fml)

```

---

SSasympOff

*Asymptotic Regression Model with an Offset*


---

**Description**

This `selfStart` model evaluates an alternative parameterization of the asymptotic regression function and the gradient with respect to those parameters. It has an `initial` attribute that creates initial estimates of the parameters `Asym`, `lrc`, and `c0`.

**Usage**

```
SSasympOff(input, Asym, lrc, c0)
```

**Arguments**

<code>input</code>	a numeric vector of values at which to evaluate the model.
<code>Asym</code>	a numeric parameter representing the horizontal asymptote on the right side (very large values of <code>input</code> ).
<code>lrc</code>	a numeric parameter representing the natural logarithm of the rate constant.
<code>c0</code>	a numeric parameter representing the <code>input</code> for which the response is zero.

**Value**

a numeric vector of the same length as `input`. It is the value of the expression  $Asym * (1 - \exp(-\exp(lrc) * (input - c0)))$ . If all of the arguments `Asym`, `lrc`, and `c0` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [selfStart](#)

**Examples**

```

CO2.Qn1 <- CO2[CO2$Plant == "Qn1", ]
SSasympOff( CO2.Qn1$conc, 32, -4, 43 ) # response only
Asym <- 32; lrc <- -4; c0 <- 43
SSasympOff( CO2.Qn1$conc, Asym, lrc, c0 ) # response and gradient
getInitial(uptake ~ SSasymp( conc, Asym, lrc, c0), data = CO2.Qn1)
## Initial values are in fact the converged values
fml <- nls(uptake ~ SSasymp( conc, Asym, lrc, c0), data = CO2.Qn1)
summary(fml)

```

---

SSasympOrig

*Asymptotic Regression Model through the Origin*


---

**Description**

This `selfStart` model evaluates the asymptotic regression function through the origin and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Asym` and `lrc` for a given set of data.

**Usage**

```
SSasympOrig(input, Asym, lrc)
```

**Arguments**

<code>input</code>	a numeric vector of values at which to evaluate the model.
<code>Asym</code>	a numeric parameter representing the horizontal asymptote.
<code>lrc</code>	a numeric parameter representing the natural logarithm of the rate constant.

**Value**

a numeric vector of the same length as `input`. It is the value of the expression  $Asym * (1 - \exp(-\exp(lrc) * input))$ . If all of the arguments `Asym` and `lrc` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [selfStart](#)

**Examples**

```
Lob.329 <- Loblolly[ Loblolly$Seed == "329", ]
SSasymOrig( Lob.329$age, 100, -3.2 ) # response only
Asym <- 100; lrc <- -3.2
SSasymOrig( Lob.329$age, Asym, lrc ) # response and gradient
getInitial(height ~ SSasymOrig(age, Asym, lrc), data = Lob.329)
## Initial values are in fact the converged values
fm1 <- nls(height ~ SSasymOrig( age, Asym, lrc), data = Lob.329)
summary(fm1)
```

---

SSbiexp

*Biexponential model*


---

**Description**

This `selfStart` model evaluates the biexponential model function and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `A1`, `lrc1`, `A2`, and `lrc2`.

**Usage**

```
SSbiexp(input, A1, lrc1, A2, lrc2)
```

**Arguments**

<code>input</code>	a numeric vector of values at which to evaluate the model.
<code>A1</code>	a numeric parameter representing the multiplier of the first exponential.
<code>lrc1</code>	a numeric parameter representing the natural logarithm of the rate constant of the first exponential.
<code>A2</code>	a numeric parameter representing the multiplier of the second exponential.
<code>lrc2</code>	a numeric parameter representing the natural logarithm of the rate constant of the second exponential.

**Value**

a numeric vector of the same length as `input`. It is the value of the expression  $A1 \cdot \exp(-\exp(lrc1) \cdot \text{input}) + A2 \cdot \exp(-\exp(lrc2) \cdot \text{input})$ . If all of the arguments `A1`, `lrc1`, `A2`, and `lrc2` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [selfStart](#)

**Examples**

```

Indo.1 <- Indometh[Indometh$Subject == 1, ]
SSbiexp( Indo.1$time, 3, 1, 0.6, -1.3 ) # response only
A1 <- 3; lrc1 <- 1; A2 <- 0.6; lrc2 <- -1.3
SSbiexp( Indo.1$time, A1, lrc1, A2, lrc2 ) # response and gradient
getInitial(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = Indo.1)
## Initial values are in fact the converged values
fm1 <- nls(conc ~ SSbiexp(time, A1, lrc1, A2, lrc2), data = Indo.1)
summary(fm1)

```

---

SSfol

*First-order Compartment Model*


---

**Description**

This selfStart model evaluates the first-order compartment function and its gradient. It has an initial attribute that creates initial estimates of the parameters lKe, lKa, and lCl.

**Usage**

```
SSfol(Dose, input, lKe, lKa, lCl)
```

**Arguments**

Dose	a numeric value representing the initial dose.
input	a numeric vector at which to evaluate the model.
lKe	a numeric parameter representing the natural logarithm of the elimination rate constant.
lKa	a numeric parameter representing the natural logarithm of the absorption rate constant.
lCl	a numeric parameter representing the natural logarithm of the clearance.

**Value**

a numeric vector of the same length as input, which is the value of the expression  $\text{Dose} * \exp(lKe + lKa - lCl) * (\exp(-\exp(lKe) * \text{input}) - \exp(-\exp(lKa) * \text{input})) / (\exp(lKa) - \exp(lKe))$ .

If all of the arguments lKe, lKa, and lCl are names of objects, the gradient matrix with respect to these names is attached as an attribute named gradient.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [selfStart](#)

**Examples**

```
Theoph.1 <- Theoph[ Theoph$Subject == 1, ]
SSfol( Theoph.1$Dose, Theoph.1$Time, -2.5, 0.5, -3 ) # response only
lKe <- -2.5; lKa <- 0.5; lCl <- -3
SSfol( Theoph.1$Dose, Theoph.1$Time, lKe, lKa, lCl ) # response and gradient
getInitial(conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data = Theoph.1)
## Initial values are in fact the converged values
fml <- nls(conc ~ SSfol(Dose, Time, lKe, lKa, lCl), data = Theoph.1)
summary(fml)
```

---

SSfpl

*Four-parameter Logistic Model*


---

**Description**

This `selfStart` model evaluates the four-parameter logistic function and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `A`, `B`, `xmid`, and `scal` for a given set of data.

**Usage**

```
SSfpl(input, A, B, xmid, scal)
```

**Arguments**

<code>input</code>	a numeric vector of values at which to evaluate the model.
<code>A</code>	a numeric parameter representing the horizontal asymptote on the left side (very small values of <code>input</code> ).
<code>B</code>	a numeric parameter representing the horizontal asymptote on the right side (very large values of <code>input</code> ).
<code>xmid</code>	a numeric parameter representing the <code>input</code> value at the inflection point of the curve. The value of <code>SSfpl</code> will be midway between <code>A</code> and <code>B</code> at <code>xmid</code> .
<code>scal</code>	a numeric scale parameter on the <code>input</code> axis.

**Value**

a numeric vector of the same length as `input`. It is the value of the expression  $A + (B - A) / (1 + \exp((xmid - input) / scal))$ . If all of the arguments `A`, `B`, `xmid`, and `scal` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [selfStart](#)

**Examples**

```
Chick.1 <- ChickWeight[ChickWeight$Chick == 1, ]
SSfpl( Chick.1$Time, 13, 368, 14, 6 ) # response only
A <- 13; B <- 368; xmid <- 14; scal <- 6
SSfpl( Chick.1$Time, A, B, xmid, scal ) # response and gradient
getInitial(weight ~ SSfpl(Time, A, B, xmid, scal), data = Chick.1)
## Initial values are in fact the converged values
fml <- nls(weight ~ SSfpl(Time, A, B, xmid, scal), data = Chick.1)
summary(fml)
```

---

SSgompertz

*Gompertz Growth Model*

---

**Description**

This `selfStart` model evaluates the Gompertz growth model and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `Asym`, `b2`, and `b3`.

**Usage**

```
SSgompertz(x, Asym, b2, b3)
```

**Arguments**

<code>x</code>	a numeric vector of values at which to evaluate the model.
<code>Asym</code>	a numeric parameter representing the asymptote.
<code>b2</code>	a numeric parameter related to the value of the function at $x = 0$
<code>b3</code>	a numeric parameter related to the scale the $x$ axis.

**Value**

a numeric vector of the same length as `input`. It is the value of the expression `Asym*exp(-b2*b3^x)`. If all of the arguments `Asym`, `b2`, and `b3` are names of objects the gradient matrix with respect to these names is attached as an attribute named `gradient`.

**Author(s)**

Douglas Bates

**See Also**

[nls](#), [selfStart](#)

**Examples**

```
DNase.1 <- subset(DNase, Run == 1)
SSlogis(log(DNase.1$conc), 4.5, 2.3, 0.7) # response only
Asym <- 4.5; b2 <- 2.3; b3 <- 0.7
SSgompertz(log(DNase.1$conc), Asym, b2, b3 ) # response and gradient
getInitial(density ~ SSgompertz(log(conc), Asym, b2, b3),
           data = DNase.1)
## Initial values are in fact the converged values
fml <- nls(density ~ SSgompertz(log(conc), Asym, b2, b3),
          data = DNase.1)
summary(fml)
```

---

SSlogis

*Logistic Model*

---

**Description**

This `selfStart` model evaluates the logistic function and its gradient. It has an `initial` attribute that creates initial estimates of the parameters `Asym`, `xmid`, and `scal`.

**Usage**

```
SSlogis(input, Asym, xmid, scal)
```

**Arguments**

<code>input</code>	a numeric vector of values at which to evaluate the model.
<code>Asym</code>	a numeric parameter representing the asymptote.
<code>xmid</code>	a numeric parameter representing the <code>x</code> value at the inflection point of the curve. The value of <code>SSlogis</code> will be <code>Asym/2</code> at <code>xmid</code> .
<code>scal</code>	a numeric scale parameter on the input axis.

**Value**

a numeric vector of the same length as `input`. It is the value of the expression  $Asym / (1 + \exp((xmid - input) / scal))$ . If all of the arguments `Asym`, `xmid`, and `scal` are names of objects the gradient matrix with respect to these names is attached as an attribute named `gradient`.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [selfStart](#)

**Examples**

```
Chick.1 <- ChickWeight[ChickWeight$Chick == 1, ]
SSlogis( Chick.1$Time, 368, 14, 6 ) # response only
Asym <- 368; xmid <- 14; scal <- 6
SSlogis( Chick.1$Time, Asym, xmid, scal ) # response and gradient
getInitial(weight ~ SSlogis(Time, Asym, xmid, scal), data = Chick.1)
## Initial values are in fact the converged values
fml <- nls(weight ~ SSlogis(Time, Asym, xmid, scal), data = Chick.1)
summary(fml)
```

---

SSmicmen

*Michaelis-Menten Model*


---

**Description**

This `selfStart` model evaluates the Michaelis-Menten model and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters  $V_m$  and  $K$ .

**Usage**

```
SSmicmen(input, Vm, K)
```

**Arguments**

<code>input</code>	a numeric vector of values at which to evaluate the model.
<code>Vm</code>	a numeric parameter representing the maximum value of the response.
<code>K</code>	a numeric parameter representing the <code>input</code> value at which half the maximum response is attained. In the field of enzyme kinetics this is called the Michaelis parameter.

**Value**

a numeric vector of the same length as `input`. It is the value of the expression  $V_m * input / (K + input)$ . If both the arguments `Vm` and `K` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

**Author(s)**

Jose Pinheiro and Douglas Bates

**See Also**

[nls](#), [selfStart](#)

**Examples**

```

PurTrt <- Puromycin[ Puromycin$state == "treated", ]
SSmicmen( PurTrt$conc, 200, 0.05 ) # response only
Vm <- 200; K <- 0.05
SSmicmen( PurTrt$conc, Vm, K ) # response and gradient
getInitial(rate ~ SSmicmen(conc, Vm, K), data = PurTrt)
## Initial values are in fact the converged values
fm1 <- nls(rate ~ SSmicmen(conc, Vm, K), data = PurTrt)
summary( fm1 )
## Alternative call using the subset argument
fm2 <- nls(rate ~ SSmicmen(conc, Vm, K), data = Puromycin,
            subset = state == "treated")
summary(fm2)

```

---

SSweibull

*Weibull growth curve model*


---

**Description**

This `selfStart` model evaluates the Weibull model for growth curve data and its gradient. It has an `initial` attribute that will evaluate initial estimates of the parameters `Asym`, `Drop`, `lrc`, and `pwr` for a given set of data.

**Usage**

```
SSweibull(x, Asym, Drop, lrc, pwr)
```

**Arguments**

<code>x</code>	a numeric vector of values at which to evaluate the model.
<code>Asym</code>	a numeric parameter representing the horizontal asymptote on the right side (very small values of <code>x</code> ).
<code>Drop</code>	a numeric parameter representing the change from <code>Asym</code> to the $y$ intercept.
<code>lrc</code>	a numeric parameter representing the natural logarithm of the rate constant.
<code>pwr</code>	a numeric parameter representing the power to which <code>x</code> is raised.

**Details**

This model is a generalization of the [SSasymp](#) model in that it reduces to `SSasymp` when `pwr` is unity.

**Value**

a numeric vector of the same length as `x`. It is the value of the expression  $Asym - Drop * \exp(-\exp(lrc) * x^{pwr})$ . If all of the arguments `Asym`, `Drop`, `lrc`, and `pwr` are names of objects, the gradient matrix with respect to these names is attached as an attribute named `gradient`.

**Author(s)**

Douglas Bates

**References**Ratkowsky, David A. (1983), *Nonlinear Regression Modeling*, Dekker. (section 4.4.5)**See Also**[nls](#), [selfStart](#), [SSasymp](#)**Examples**

```
Chick.6 <- subset(ChickWeight, (Chick == 6) & (Time > 0))
SSweibull(Chick.6$Time, 160, 115, -5.5, 2.5 ) # response only
Asym <- 160; Drop <- 115; lrc <- -5.5; pwr <- 2.5
SSweibull(Chick.6$Time, Asym, Drop, lrc, pwr) # response and gradient
getInitial(weight ~ SSweibull(Time, Asym, Drop, lrc, pwr), data = Chick.6)
## Initial values are in fact the converged values
fml <- nls(weight ~ SSweibull(Time, Asym, Drop, lrc, pwr), data = Chick.6)
summary(fml)
```

---

start

---

*Encode the Terminal Times of Time Series*


---

**Description**

Extract and encode the times the first and last observations were taken. Provided only for compatibility with S version 2.

**Usage**

```
start(x, ...)
end(x, ...)
```

**Arguments**

**x** a univariate or multivariate time-series, or a vector or matrix.  
**...** extra arguments for future methods.

**Details**

These are generic functions, which will use the `tsp` attribute of `x` if it exists. Their default methods decode the start time from the original time units, so that for a monthly series 1995.5 is represented as `c(1995, 7)`. For a series of frequency `f`, time `n+i/f` is presented as `c(n, i+1)` (even for `i = 0` and `f = 1`).

**Warning**

The representation used by `start` and `end` has no meaning unless the frequency is supplied.

**See Also**

[ts](#), [time](#), [tsp](#).

---

 stat.anova

*GLM Anova Statistics*


---

**Description**

This is a utility function, used in `lm` and `glm` methods for `anova(..., test != NULL)` and should not be used by the average user.

**Usage**

```
stat.anova(table, test = c("Chisq", "F", "Cp"), scale, df.scale, n)
```

**Arguments**

<code>table</code>	numeric matrix as results from <code>anova.glm(..., test=NULL)</code> .
<code>test</code>	a character string, matching one of "Chisq", "F" or "Cp".
<code>scale</code>	a weighted residual sum of squares.
<code>df.scale</code>	degrees of freedom corresponding to scale.
<code>n</code>	number of observations.

**Value**

A matrix which is the original `table`, augmented by a column of test statistics, depending on the `test` argument.

**References**

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models*. Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[anova.lm](#), [anova.glm](#).

**Examples**

```
##-- Continued from '?glm':

print(ag <- anova(glm.D93))
stat.anova(ag$table, test = "Cp",
           scale = sum(resid(glm.D93, "pearson")^2)/4, df = 4, n = 9)
```

---

step	<i>Choose a model by AIC in a Stepwise Algorithm</i>
------	--

---

**Description**

Select a formula-based model by AIC.

**Usage**

```
step(object, scope, scale = 0,
      direction = c("both", "backward", "forward"),
      trace = 1, keep = NULL, steps = 1000, k = 2, ...)
```

**Arguments**

object	an object representing a model of an appropriate class (mainly "lm" and "glm"). This is used as the initial model in the stepwise search.
scope	defines the range of models examined in the stepwise search. This should be either a single formula, or a list containing components <code>upper</code> and <code>lower</code> , both formulae. See the details for how to specify the formulae and how they are used.
scale	used in the definition of the AIC statistic for selecting the models, currently only for <code>lm</code> , <code>ao</code> and <code>glm</code> models.
direction	the mode of stepwise search, can be one of "both", "backward", or "forward", with a default of "both". If the <code>scope</code> argument is missing the default for <code>direction</code> is "backward".
trace	if positive, information is printed during the running of <code>step</code> . Larger values may give more detailed information.
keep	a filter function whose input is a fitted model object and the associated AIC statistic, and whose output is arbitrary. Typically <code>keep</code> will select a subset of the components of the object and return them. The default is not to keep anything.
steps	the maximum number of steps to be considered. The default is 1000 (essentially as many as required). It is typically used to stop the process early.
k	the multiple of the number of degrees of freedom used for the penalty. Only <code>k = 2</code> gives the genuine AIC: <code>k = log(n)</code> is sometimes referred to as BIC or SBC.
...	any additional arguments to <code>extractAIC</code> .

**Details**

`step` uses `add1` and `drop1` repeatedly; it will work for any method for which they work, and that is determined by having a valid method for `extractAIC`. When the additive constant can be chosen so that AIC is equal to Mallows'  $C_p$ , this is done and the tables are labelled appropriately.

The set of models searched is determined by the `scope` argument. The right-hand-side of its lower component is always included in the model, and right-hand-side of the model is included in the upper component. If `scope` is a single formula, it specifies the upper component, and the lower model is empty. If `scope` is missing, the initial model is used as the upper model.

Models specified by `scope` can be templates to update `object` as used by `update.formula`.

There is a potential problem in using `glm` fits with a variable `scale`, as in that case the deviance is not simply related to the maximized log-likelihood. The "glm" method for function `extractAIC` makes the appropriate adjustment for a `gaussian` family, but may need to be amended for other cases. (The `binomial` and `poisson` families have fixed `scale` by default and do not correspond to a particular maximum-likelihood problem for variable `scale`.)

### Value

the stepwise-selected model is returned, with up to two additional components. There is an "anova" component corresponding to the steps taken in the search, as well as a "keep" component if the `keep=` argument was supplied in the call. The "Resid. Dev" column of the analysis of deviance table refers to a constant minus twice the maximized log likelihood: it will be a deviance only in cases where a saturated model is well-defined (thus excluding `lm`, `aov` and `survreg` fits, for example).

### Warning

The model fitting must apply the models to the same dataset. This may be a problem if there are missing values and R's default of `na.action = na.omit` is used. We suggest you remove the missing values first.

### Note

This function differs considerably from the function in S, which uses a number of approximations and does not compute the correct AIC.

This is a minimal implementation. Use `stepAIC` in package **MASS** for a wider range of object classes.

### Author(s)

B. D. Ripley: `step` is a slightly simplified version of `stepAIC` in package **MASS** (Venables & Ripley, 2002 and earlier editions).

The idea of a `step` function follows that described in Hastie & Pregibon (1992); but the implementation in R is more general.

### References

Hastie, T. J. and Pregibon, D. (1992) *Generalized linear models*. Chapter 6 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. New York: Springer (4th ed).

**See Also**

[stepAIC](#) in [MASS](#), [add1](#), [drop1](#)

**Examples**

```
example(lm)
step(lm.D9)

summary(lm1 <- lm(Fertility ~ ., data = swiss))
slm1 <- step(lm1)
summary(slm1)
slm1$anova
```

---

stepfun

*Step Function Class*

---

**Description**

Given the vectors  $(x_1, \dots, x_n)$  and  $(y_0, y_1, \dots, y_n)$  (one value more!), `stepfun(x, y, ...)` returns an interpolating “step” function, say `fn`. I.e.,  $fn(t) = c_i$  (constant) for  $t \in (x_i, x_{i+1})$  and at the abscissa values, if (by default) `right = FALSE`,  $fn(x_i) = y_i$  and for `right = TRUE`,  $fn(x_i) = y_{i-1}$ , for  $i = 1, \dots, n$ .

The value of the constant  $c_i$  above depends on the “continuity” parameter `f`. For the default, `right = FALSE`, `f = 0`, `fn` is a “cadlag” function, i.e., continuous at right, limit (“the point”) at left. In general,  $c_i$  is interpolated in between the neighbouring  $y$  values,  $c_i = (1 - f)y_i + f \cdot y_{i+1}$ . Therefore, for non-0 values of `f`, `fn` may no longer be a proper step function, since it can be discontinuous from both sides, unless `right = TRUE`, `f = 1` which is right-continuous.

**Usage**

```
stepfun(x, y, f = as.numeric(right), ties = "ordered", right = FALSE)

is.stepfun(x)
knots(Fn, ...)
as.stepfun(x, ...)
```

## S3 method for class 'stepfun':  
`print(x, digits = getOption("digits") - 2, ...)`

## S3 method for class 'stepfun':  
`summary(object, ...)`

**Arguments**

`x` numeric vector giving the knots or jump locations of the step function for `stepfun()`. For the other functions, `x` is as object below.

<code>y</code>	numeric vector one longer than <code>x</code> , giving the heights of the function values <i>between</i> the <code>x</code> values.
<code>f</code>	a number between 0 and 1, indicating how interpolation outside the given <code>x</code> values should happen. See <a href="#">approxfun</a> .
<code>ties</code>	Handling of tied <code>x</code> values. Either a function or the string "ordered". See <a href="#">approxfun</a> .
<code>right</code>	logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa.
<code>Fn, object</code>	an R object inheriting from "stepfun".
<code>digits</code>	number of significant digits to use, see <a href="#">print</a> .
<code>...</code>	potentially further arguments (required by the generic).

### Value

A function of class "stepfun", say `fn`. There are methods available for summarizing ("`summary(.)`"), representing ("`print(.)`") and plotting ("`plot(.)`", see [plot.stepfun](#)) "stepfun" objects.

The [environment](#) of `fn` contains all the information needed;

<code>"x" , "y"</code>	the original arguments
<code>"n"</code>	number of knots (x values)
<code>"f"</code>	continuity parameter
<code>"yleft" , "yright"</code>	the function values <i>outside</i> the knots;
<code>"method"</code>	(always == "constant", from <a href="#">approxfun(.)</a> ).

normal-bracket109bracket-normal The knots are also available by [knots\(fn\)](#).

### Author(s)

Martin Maechler, [maechler@stat.math.ethz.ch](mailto:maechler@stat.math.ethz.ch) with some basic code from Thomas Lumley.

### See Also

[ecdf](#) for empirical distribution functions as special step functions and [plot.stepfun](#) for *plotting* step functions.

[approxfun](#) and [splinefun](#).

### Examples

```

y0 <- c(1,2,4,3)
sfun0 <- stepfun(1:3, y0, f = 0)
sfun.2 <- stepfun(1:3, y0, f = .2)
sfun1 <- stepfun(1:3, y0, f = 1)
sfun1c <- stepfun(1:3, y0, right=TRUE)# hence f=1
sfun0
summary(sfun0)

```

```
summary(sfun.2)

## look at the internal structure:
unclass(sfun0)
ls(envir = environment(sfun0))

x0 <- seq(0.5,3.5, by = 0.25)
rbind(x=x0, f.f0 = sfun0(x0), f.f02= sfun.2(x0),
      f.f1 = sfun1(x0), f.f1c = sfun1c(x0))
## Identities :
stopifnot(identical(y0[-1], sfun0 (1:3)),# right = FALSE
          identical(y0[-4], sfun1c(1:3)))# right = TRUE
```

---

 stl

*Seasonal Decomposition of Time Series by Loess*


---

### Description

Decompose a time series into seasonal, trend and irregular components using loess, acronym STL.

### Usage

```
stl(x, s.window, s.degree = 0,
    t.window = NULL, t.degree = 1,
    l.window = nextodd(period), l.degree = t.degree,
    s.jump = ceiling(s.window/10),
    t.jump = ceiling(t.window/10),
    l.jump = ceiling(l.window/10),
    robust = FALSE,
    inner = if(robust) 1 else 2,
    outer = if(robust) 15 else 0,
    na.action = na.fail)
```

### Arguments

x	univariate time series to be decomposed. This should be an object of class "ts" with a frequency greater than one.
s.window	either the character string "periodic" or the span (in lags) of the loess window for seasonal extraction, which should be odd. This has no default.
s.degree	degree of locally-fitted polynomial in seasonal extraction. Should be zero or one.
t.window	the span (in lags) of the loess window for trend extraction, which should be odd. If NULL, the default, <code>nextodd(ceiling((1.5*period) / (1-(1.5/s.window))))</code> , is taken.
t.degree	degree of locally-fitted polynomial in trend extraction. Should be zero or one.

<code>l.window</code>	the span (in lags) of the loess window of the low-pass filter used for each subseries. Defaults to the smallest odd integer greater than or equal to <code>frequency(x)</code> which is recommended since it prevents competition between the trend and seasonal components. If not an odd integer its given value is increased to the next odd one.
<code>l.degree</code>	degree of locally-fitted polynomial for the subseries low-pass filter. Must be 0 or 1.
<code>s.jump</code> , <code>t.jump</code> , <code>l.jump</code>	integers at least one to increase speed of the respective smoother. Linear interpolation happens between every <code>*.jumpth</code> value.
<code>robust</code>	logical indicating if robust fitting be used in the <code>loess</code> procedure.
<code>inner</code>	integer; the number of 'inner' (backfitting) iterations; usually very few (2) iterations suffice.
<code>outer</code>	integer; the number of 'outer' robustness iterations.
<code>na.action</code>	action on missing values.

### Details

The seasonal component is found by *loess* smoothing the seasonal sub-series (the series of all January values, ...); if `s.window = "periodic"` smoothing is effectively replaced by taking the mean. The seasonal values are removed, and the remainder smoothed to find the trend. The overall level is removed from the seasonal component and added to the trend component. This process is iterated a few times. The `remainder` component is the residuals from the seasonal plus trend fit.

Several methods for the resulting class "`stl`" objects, see, [plot.stl](#).

### Value

`stl` returns an object of class "`stl`" with components

<code>time.series</code>	a multiple time series with columns <code>seasonal</code> , <code>trend</code> and <code>remainder</code> .
<code>weights</code>	the final robust weights (all one if fitting is not done robustly).
<code>call</code>	the matched call.
<code>win</code>	integer (length 3 vector) with the spans used for the " <code>s</code> ", " <code>t</code> ", and " <code>l</code> " smoothers.
<code>deg</code>	integer (length 3) vector with the polynomial degrees for these smoothers.
<code>jump</code>	integer (length 3) vector with the "jumps" (skips) used for these smoothers.
<code>ni</code>	number of inner iterations
<code>no</code>	number of outer robustness iterations

### Note

This is similar to but not identical to the `stl` function in S-PLUS. The `remainder` component given by S-PLUS is the sum of the `trend` and `remainder` series from this function.

**Author(s)**

B.D. Ripley; Fortran code by Cleveland *et al.* (1990) from 'netlib'.

**References**

R. B. Cleveland, W. S. Cleveland, J.E. McRae, and I. Terpenning (1990) STL: A Seasonal-Trend Decomposition Procedure Based on Loess. *Journal of Official Statistics*, **6**, 3–73.

**See Also**

`plot.stl` for stl methods; `loess` in package `stats` (which is not actually used in stl).

**Examples**

```
plot(stl(nottem, "per"))
plot(stl(nottem, s.win = 4, t.win = 50, t.jump = 1))

plot(stllc <- stl(log(co2), s.window=21))
summary(stllc)
## linear trend, strict period.
plot(stl(log(co2), s.window="per", t.window=1000))

## Two STL plotted side by side :
      stmd <- stl(mdeaths, s.w = "per") # un-robust
summary(stmR <- stl(mdeaths, s.w = "per", robust = TRUE))
op <- par(mar = c(0, 4, 0, 3), oma = c(5, 0, 4, 0), mfcol = c(4, 2))
plot(stmd, set.pars=NULL, labels = NULL,
      main = "stl(mdeaths, s.w = \"per\", robust = FALSE / TRUE)")
plot(stmR, set.pars=NULL)
# mark the 'outliers' :
(iO <- which(stmR $ weights < 1e-8)) # 10 were considered outliers
sts <- stmR$time.series
points(time(sts)[iO], .8* sts[,"remainder"][iO], pch = 4, col = "red")
par(op)# reset
```

---

stlmethods

*Methods for STL Objects*


---

**Description**

Methods for objects of class `stl`, typically the result of `stl`. The `plot` method does a multiple figure plot with some flexibility.

**Usage**

```
## S3 method for class 'stl':
plot(x, labels = colnames(X),
      set.pars = list(mar = c(0, 6, 0, 6), oma = c(6, 0, 4, 0),
                     tck = -0.01, mfrow = c(nplot, 1)),
      main = NULL, range.bars = TRUE, ...)
```

**Arguments**

<code>x</code>	<code>stl</code> object.
<code>labels</code>	character of length 4 giving the names of the component time-series.
<code>set.pars</code>	settings for <code>par( . )</code> when setting up the plot.
<code>main</code>	plot main title.
<code>range.bars</code>	logical indicating if each plot should have a bar at its right side which are of equal heights in user coordinates.
<code>...</code>	further arguments passed to or from other methods.

**See Also**

`plot.ts` and `stl`, particularly for examples.

---

 StructTS

*Fit Structural Time Series*


---

**Description**

Fit a structural model for a time series by maximum likelihood.

**Usage**

```
StructTS(x, type = c("level", "trend", "BSM"), init = NULL,
         fixed = NULL, optim.control = NULL)
```

**Arguments**

<code>x</code>	a univariate numeric time series. Missing values are allowed.
<code>type</code>	the class of structural model. If omitted, a BSM is used for a time series with <code>frequency(x) &gt; 1</code> , and a local trend model otherwise.
<code>init</code>	initial values of the variance parameters.
<code>fixed</code>	optional numeric vector of the same length as the total number of parameters. If supplied, only non-NA entries in <code>fixed</code> will be varied. Probably most useful for setting variances to zero.
<code>optim.control</code>	List of control parameters for <code>optim</code> . Method "L-BFGS-B" is used.

## Details

*Structural time series* models are (linear Gaussian) state-space models for (univariate) time series based on a decomposition of the series into a number of components. They are specified by a set of error variances, some of which may be zero.

The simplest model is the *local level* model specified by `type = "level"`. This has an underlying level  $\mu_t$  which evolves by

$$\mu_{t+1} = \mu_t + \xi_t, \quad \xi_t \sim N(0, \sigma_\xi^2)$$

The observations are

$$x_t = \mu_t + \epsilon_t, \quad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

There are two parameters,  $\sigma_\xi^2$  and  $\sigma_\epsilon^2$ . It is an ARIMA(0,1,1) model, but with restrictions on the parameter set.

The *local linear trend model*, `type = "trend"`, has the same measurement equation, but with a time-varying slope in the dynamics for  $\mu_t$ , given by

$$\begin{aligned} \mu_{t+1} &= \mu_t + \nu_t + \xi_t, & \xi_t &\sim N(0, \sigma_\xi^2) \\ \nu_{t+1} &= \nu_t + \zeta_t, & \zeta_t &\sim N(0, \sigma_\zeta^2) \end{aligned}$$

with three variance parameters. It is not uncommon to find  $\sigma_\zeta^2 = 0$  (which reduces to the local level model) or  $\sigma_\xi^2 = 0$ , which ensures a smooth trend. This is a restricted ARIMA(0,2,2) model.

The *basic structural model*, `type = "BSM"`, is a local trend model with an additional seasonal component. Thus the measurement equation is

$$x_t = \mu_t + \gamma_t + \epsilon_t, \quad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

where  $\gamma_t$  is a seasonal component with dynamics

$$\gamma_{t+1} = -\gamma_t + \dots + \gamma_{t-s+2} + \omega_t, \quad \omega_t \sim N(0, \sigma_\omega^2)$$

The boundary case  $\sigma_\omega^2 = 0$  corresponds to a deterministic (but arbitrary) seasonal pattern. (This is sometimes known as the ‘dummy variable’ version of the BSM.)

## Value

A list of class "StructTS" with components:

<code>coef</code>	the estimated variances of the components.
<code>loglik</code>	the maximized log-likelihood. Note that as all these models are non-stationary this includes a diffuse prior for some observations and hence is not comparable with <a href="#">arima</a> nor different types of structural models.
<code>data</code>	the time series <code>x</code> .
<code>residuals</code>	the standardized residuals.
<code>fitted</code>	a multiple time series with one component for the level, slope and seasonal components, estimated contemporaneously (that is at time $t$ and not at the end of the series).
<code>call</code>	the matched call.

series	the name of the series <code>x</code> .
convergence	the value returned by <code>optim</code> .
model, model0	Lists representing the Kalman Filter used in the fitting. See <code>KalmanLike</code> . model0 is the initial state of the filter, model its final state.
xtsp	the <code>tsp</code> attributes of <code>x</code> .

### Note

Optimization of structural models is a lot harder than many of the references admit. For example, the `AirPassengers` data are considered in Brockwell & Davis (1996): their solution appears to be a local maximum, but nowhere near as good a fit as that produced by `StructTS`. It is quite common to find fits with one or more variances zero, and this can include  $\sigma_\epsilon^2$ .

### References

- Brockwell, P. J. & Davis, R. A. (1996). *Introduction to Time Series and Forecasting*. Springer, New York. Sections 8.2 and 8.5.
- Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press.
- Harvey, A. C. (1989) *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press.
- Harvey, A. C. (1993) *Time Series Models*. 2nd Edition, Harvester Wheatsheaf.

### See Also

`KalmanLike`, `tsSmooth`.

### Examples

```
## see also JohnsonJohnson, Nile and AirPassengers

trees <- window(treering, start=0)
(fit <- StructTS(trees, type = "level"))
plot(trees)
lines(fitted(fit), col = "green")
tsdiag(fit)

(fit <- StructTS(log10(UKgas), type = "BSM"))
par(mfrow = c(4, 1))
plot(log10(UKgas))
plot(cbind(fitted(fit), resid=resid(fit)), main = "UK gas consumption")
```

---

`summary.aov`*Summarize an Analysis of Variance Model*

---

## Description

Summarize an analysis of variance model.

## Usage

```
## S3 method for class 'aov':
summary(object, intercept = FALSE, split,
        expand.split = TRUE, keep.zero.df = TRUE, ...)

## S3 method for class 'aovlist':
summary(object, ...)
```

## Arguments

<code>object</code>	An object of class "aov" or "aovlist".
<code>intercept</code>	logical: should intercept terms be included?
<code>split</code>	an optional named list, with names corresponding to terms in the model. Each component is itself a list with integer components giving contrasts whose contributions are to be summed.
<code>expand.split</code>	logical: should the split apply also to interactions involving the factor?
<code>keep.zero.df</code>	logical: should terms with no degrees of freedom be included?
<code>...</code>	Arguments to be passed to or from other methods, for <code>summary.aovlist</code> including those for <code>summary.aov</code> .

## Value

An object of class `c("summary.aov", "listof")` or `"summary.aovlist"` respectively.

For a fits with a single stratum the result will be a list of ANOVA tables, one for each response (even if there is only one response): the tables are of class "anova" inheriting from class "data.frame".

For multistratum fits the return value is a list of summaries, one for each stratum.

## Note

The use of `expand.split = TRUE` is little tested: it is always possible to set it to `FALSE` and specify exactly all the splits required.

## See Also

[aov](#), [summary](#), [model.tables](#), [TukeyHSD](#)

**Examples**

```

## From Venables and Ripley (2002) p.165.
N <- c(0,1,0,1,1,1,0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,1,0,0)
P <- c(1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,1,1,0,0,1,0,1,1,0)
K <- c(1,0,0,1,0,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0)
yield <- c(49.5,62.8,46.8,57.0,59.8,58.5,55.5,56.0,62.8,55.8,69.5,55.0,
          62.0,48.8,45.5,44.2,52.0,51.5,49.8,48.8,57.2,59.0,53.2,56.0)
npk <- data.frame(block=gl(6,4), N=factor(N), P=factor(P),
                  K=factor(K), yield=yield)

( npk.aov <- aov(yield ~ block + N*P*K, npk) )
summary(npk.aov)
coefficients(npk.aov)

# Cochran and Cox (1957, p.164)
# 3x3 factorial with ordered factors, each is average of 12.
CC <- data.frame(
  y = c(449, 413, 326, 409, 358, 291, 341, 278, 312)/12,
  P = ordered(gl(3, 3)), N = ordered(gl(3, 1, 9))
)
CC.aov <- aov(y ~ N * P, data = CC , weights = rep(12, 9))
summary(CC.aov)

# Split both main effects into linear and quadratic parts.
summary(CC.aov, split = list(N = list(L = 1, Q = 2),
                             P = list(L = 1, Q = 2)))

# Split only the interaction
summary(CC.aov, split = list("N:P" = list(L.L = 1, Q = 2:4)))

# split on just one var
summary(CC.aov, split = list(P = list(lin = 1, quad = 2)))
summary(CC.aov, split = list(P = list(lin = 1, quad = 2)),
        expand.split=FALSE)

```

summary.glm

*Summarizing Generalized Linear Model Fits***Description**

These functions are all [methods](#) for class `glm` or `summary.glm` objects.

**Usage**

```

## S3 method for class 'glm':
summary(object, dispersion = NULL, correlation = FALSE,
        symbolic.cor = FALSE, ...)

## S3 method for class 'summary.glm':

```

```
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

### Arguments

<code>object</code>	an object of class "glm", usually, a result of a call to <a href="#">glm</a> .
<code>x</code>	an object of class "summary.glm", usually, a result of a call to <code>summary.glm</code> .
<code>dispersion</code>	the dispersion parameter for the fitting family. By default it is obtained from <code>object</code> .
<code>correlation</code>	logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.
<code>digits</code>	the number of significant digits to use when printing.
<code>symbolic.cor</code>	logical. If TRUE, print the correlations in a symbolic form (see <a href="#">symnum</a> ) rather than as numbers.
<code>signif.stars</code>	logical. If TRUE, "significance stars" are printed for each coefficient.
<code>...</code>	further arguments passed to or from other methods.

### Details

`print.summary.glm` tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives "significance stars" if `signif.stars` is TRUE.

Aliased coefficients are omitted in the returned object but (as from R 1.8.0) restored by the `print` method.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print `summary(object)$correlation` directly.

### Value

`summary.glm` returns an object of class "summary.glm", a list with components

<code>call</code>	the component from <code>object</code> .
<code>family</code>	the component from <code>object</code> .
<code>deviance</code>	the component from <code>object</code> .
<code>contrasts</code>	the component from <code>object</code> .
<code>df.residual</code>	the component from <code>object</code> .
<code>null.deviance</code>	the component from <code>object</code> .
<code>df.null</code>	the component from <code>object</code> .
<code>deviance.resid</code>	the deviance residuals: see <a href="#">residuals.glm</a> .
<code>coefficients</code>	the matrix of coefficients, standard errors, z-values and p-values. Aliased coefficients are omitted.

**aliased**            named logical vector showing if the original coefficients are aliased.  
**dispersion**       either the supplied argument or the estimated dispersion if the latter is NULL  
**df**                 a 3-vector of the rank of the model and the number of residual degrees of freedom, plus number of non-aliased coefficients.  
**cov.unscaled**    the unscaled (dispersion = 1) estimated covariance matrix of the estimated coefficients.  
**cov.scaled**       ditto, scaled by dispersion.  
**correlation**      (only if correlation is true.) The estimated correlations of the estimated coefficients.  
**symbolic.cor**    (only if correlation is true.) The value of the argument symbolic.cor.

### See Also

[glm](#), [summary](#).

### Examples

```
## --- Continuing the Example from '?glm':
summary(glm.D93)
```

---

summary.lm

*Summarizing Linear Model Fits*

---

### Description

summary method for class "lm".

### Usage

```
## S3 method for class 'lm':
summary(object, correlation = FALSE, symbolic.cor = FALSE, ...)

## S3 method for class 'summary.lm':
print(x, digits = max(3, getOption("digits") - 3),
      symbolic.cor = x$symbolic.cor,
      signif.stars = getOption("show.signif.stars"), ...)
```

### Arguments

**object**            an object of class "lm", usually, a result of a call to [lm](#).  
**x**                  an object of class "summary.lm", usually, a result of a call to `summary.lm`.  
**correlation**      logical; if TRUE, the correlation matrix of the estimated parameters is returned and printed.  
**digits**            the number of significant digits to use when printing.

`symbolic.cor` logical. If TRUE, print the correlations in a symbolic form (see `symnum`) rather than as numbers.

`signif.stars` logical. If TRUE, “significance stars” are printed for each coefficient.

... further arguments passed to or from other methods.

### Details

`print.summary.lm` tries to be smart about formatting the coefficients, standard errors, etc. and additionally gives “significance stars” if `signif.stars` is TRUE.

Correlations are printed to two decimal places (or symbolically): to see the actual correlations print `summary(object)$correlation` directly.

### Value

The function `summary.lm` computes and returns a list of summary statistics of the fitted linear model given in `object`, using the components (list elements) “`call`” and “`terms`” from its argument, plus

`residuals` the *weighted* residuals, the usual residuals rescaled by the square root of the weights specified in the call to `lm`.

`coefficients` a  $p \times 4$  matrix with columns for the estimated coefficient, its standard error, t-statistic and corresponding (two-sided) p-value. Aliased coefficients are omitted.

`aliased` named logical vector showing if the original coefficients are aliased.

`sigma` the square root of the estimated variance of the random error

$$\hat{\sigma}^2 = \frac{1}{n-p} \sum_i R_i^2,$$

where  $R_i$  is the  $i$ -th residual, `residuals[i]`.

`df` degrees of freedom, a 3-vector  $(p, n-p, p^*)$ , the last being the number of non-aliased coefficients.

`fstatistic` (for models including non-intercept terms) a 3-vector with the value of the F-statistic with its numerator and denominator degrees of freedom.

`r.squared`  $R^2$ , the “fraction of variance explained by the model”,

$$R^2 = 1 - \frac{\sum_i R_i^2}{\sum_i (y_i - y^*)^2},$$

where  $y^*$  is the mean of  $y_i$  if there is an intercept and zero otherwise.

`adj.r.squared` the above  $R^2$  statistic “*adjusted*”, penalizing for higher  $p$ .

`cov.unscaled` a  $p \times p$  matrix of (unscaled) covariances of the  $\hat{\beta}_j$ ,  $j = 1, \dots, p$ .

`correlation` the correlation matrix corresponding to the above `cov.unscaled`, if `correlation = TRUE` is specified.

`symbolic.cor` (only if `correlation` is true.) The value of the argument `symbolic.cor`.

**See Also**

The model fitting function [lm](#), [summary](#).

Function [coef](#) will extract the matrix of coefficients with standard errors, t-statistics and p-values.

**Examples**

```
##-- Continuing the lm(.) example:
coef(lm.D90)# the bare coefficients
sld90 <- summary(lm.D90 <- lm(weight ~ group -1))# omitting intercept
sld90
coef(sld90)# much more
```

---

summary.manova

*Summary Method for Multivariate Analysis of Variance*


---

**Description**

A summary method for class "manova".

**Usage**

```
## S3 method for class 'manova':
summary(object,
        test = c("Pillai", "Wilks", "Hotelling-Lawley", "Roy"),
        intercept = FALSE, ...)
```

**Arguments**

object	An object of class "manova" or an aov object with multiple responses.
test	The name of the test statistic to be used. Partial matching is used so the name can be abbreviated.
intercept	logical. If TRUE, the intercept term is included in the table.
...	further arguments passed to or from other methods.

**Details**

The `summary.manova` method uses a multivariate test statistic for the summary table. Wilks' statistic is most popular in the literature, but the default Pillai-Bartlett statistic is recommended by Hand and Taylor (1987).

**Value**

A list with components

SS	A named list of sums of squares and product matrices.
Eigenvalues	A matrix of eigenvalues.
stats	A matrix of the statistics, approximate F value, degrees of freedom and P value.

**References**

- Krzanowski, W. J. (1988) *Principles of Multivariate Analysis. A User's Perspective*. Oxford.
- Hand, D. J. and Taylor, C. C. (1987) *Multivariate Analysis of Variance and Repeated Measures*. Chapman and Hall.

**See Also**

[manova](#), [aov](#)

**Examples**

```
## Example on producing plastic film from Krzanowski (1998, p. 381)
tear <- c(6.5, 6.2, 5.8, 6.5, 6.5, 6.9, 7.2, 6.9, 6.1, 6.3,
         6.7, 6.6, 7.2, 7.1, 6.8, 7.1, 7.0, 7.2, 7.5, 7.6)
gloss <- c(9.5, 9.9, 9.6, 9.6, 9.2, 9.1, 10.0, 9.9, 9.5, 9.4,
          9.1, 9.3, 8.3, 8.4, 8.5, 9.2, 8.8, 9.7, 10.1, 9.2)
opacity <- c(4.4, 6.4, 3.0, 4.1, 0.8, 5.7, 2.0, 3.9, 1.9, 5.7,
            2.8, 4.1, 3.8, 1.6, 3.4, 8.4, 5.2, 6.9, 2.7, 1.9)
Y <- cbind(tear, gloss, opacity)
rate <- factor(gl(2,10), labels=c("Low", "High"))
additive <- factor(gl(2, 5, len=20), labels=c("Low", "High"))

fit <- manova(Y ~ rate * additive)
summary.aov(fit) # univariate ANOVA tables
summary(fit, test="Wilks") # ANOVA table of Wilks' lambda
```

---

summary.princomp    *Summary method for Principal Components Analysis*

---

**Description**

The [summary](#) method for class "princomp".

**Usage**

```
## S3 method for class 'princomp':
summary(object, loadings = FALSE, cutoff = 0.1, ...)

## S3 method for class 'summary.princomp':
print(x, digits = 3, loadings = x$print.loadings,
      cutoff = x$cutoff, ...)
```

**Arguments**

object	an object of class "princomp", as from princomp( ).
loadings	logical. Should loadings be included?
cutoff	numeric. Loadings below this cutoff in absolute value are shown as blank in the output.

`x` an object of class "summary.princomp".  
`digits` the number of significant digits to be used in listing loadings.  
`...` arguments to be passed to or from other methods.

**Value**

object with additional components `cutoff` and `print.loadings`.

**See Also**

[princomp](#)

**Examples**

```
summary(pc.cr <- princomp(USArrests, cor=TRUE))
print(summary(princomp(USArrests, cor=TRUE),
  loadings = TRUE, cutoff = 0.2), digits = 2)
```

---

supsmu

*Friedman's SuperSmoother*


---

**Description**

Smooth the (x, y) values by Friedman's "super smoother".

**Usage**

```
supsmu(x, y, wt, span = "cv", periodic = FALSE, bass = 0)
```

**Arguments**

`x` x values for smoothing  
`y` y values for smoothing  
`wt` case weights, by default all equal  
`span` the fraction of the observations in the span of the running lines smoother, or "cv" to choose this by leave-one-out cross-validation.  
`periodic` if TRUE, the x values are assumed to be in [0, 1] and of period 1.  
`bass` controls the smoothness of the fitted curve. Values of up to 10 indicate increasing smoothness.

**Details**

supsmu is a running lines smoother which chooses between three spans for the lines. The running lines smoothers are symmetric, with  $k/2$  data points each side of the predicted point, and values of  $k$  as  $0.5 * n$ ,  $0.2 * n$  and  $0.05 * n$ , where  $n$  is the number of data points. If `span` is specified, a single smoother with span  $span * n$  is used.

The best of the three smoothers is chosen by cross-validation for each prediction. The best spans are then smoothed by a running lines smoother and the final prediction chosen by linear interpolation.

The FORTRAN code says: “For small samples ( $n < 40$ ) or if there are substantial serial correlations between observations close in  $x$ -value, then a prespecified fixed span smoother ( $span > 0$ ) should be used. Reasonable span values are 0.2 to 0.4.”

**Value**

A list with components

- `x`                    the input values in increasing order with duplicates removed.
- `y`                    the corresponding  $y$  values on the fitted curve.

**References**

Friedman, J. H. (1984) SMART User’s Guide. Laboratory for Computational Statistics, Stanford University Technical Report No. 1.

Friedman, J. H. (1984) A variable span scatterplot smoother. Laboratory for Computational Statistics, Stanford University Technical Report No. 5.

**See Also**

[ppr](#)

**Examples**

```
with(cars, {
  plot(speed, dist)
  lines(supsmu(speed, dist))
  lines(supsmu(speed, dist, bass = 7), lty = 2)
})
```

---

symnum

*Symbolic Number Coding*

---

**Description**

Symbolically encode a given numeric or logical vector or array.

**Usage**

```

symnum(x, cutpoints=c(0.3, 0.6, 0.8, 0.9, 0.95),
       symbols=c(" ", ".", ",", "+", "*", "B"),
       legend = length(symbols) >= 3,
       na = "?", eps = 1e-5, corr = missing(cutpoints),
       show.max = if(corr) "1", show.min = NULL,
       abbr.colnames = has.colnames,
       lower.triangular = corr && is.numeric(x) && is.matrix(x),
       diag.lower.tri = corr && !is.null(show.max))

```

**Arguments**

**x** numeric or logical vector or array.

**cutpoints** numeric vector whose values  $\text{cutpoints}[j] = c_j$  (after augmentation, see `corr` below) are used for intervals.

**symbols** character vector, one shorter than (the *augmented*, see `corr` below) `cutpoints`.  $\text{symbols}[j] = s_j$  are used as “code” for the (half open) interval  $(c_j, c_{j+1}]$ .  
For logical argument `x`, the default is `c(" . , " | ")` (graphical 0 / 1 s).

**legend** logical indicating if a "legend" attribute is desired.

**na** character or logical. How **NA**s are coded. If `na == FALSE`, **NA**s are coded invisibly, including the "legend" attribute below, which otherwise mentions **NA** coding.

**eps** absolute precision to be used at left and right boundary.

**corr** logical. If `TRUE`, `x` contains correlations. The cutpoints are augmented by 0 and 1 and `abs(x)` is coded.

**show.max** if `TRUE`, or of mode character, the maximal cutpoint is coded especially.

**show.min** if `TRUE`, or of mode character, the minimal cutpoint is coded especially.

**abbr.colnames** logical, integer or `NULL` indicating how column names should be abbreviated (if they are); if `NULL` (or `FALSE` and `x` has no column names), the column names will all be empty, i.e., `" "`; otherwise if `abbr.colnames` is false, they are left unchanged. If `TRUE` or integer, existing column names will be abbreviated to `abbreviate(*, minlength = abbr.colnames)`.

**lower.triangular** logical. If `TRUE` and `x` is a matrix, only the *lower triangular* part of the matrix is coded as non-blank.

**diag.lower.tri** logical. If `lower.triangular` and this are `TRUE`, the *diagonal* part of the matrix is shown.

**Value**

An atomic character object of class `noquote` and the same dimensions as `x`.

If legend (TRUE by default when there more than 2 classes), it has an attribute "legend" containing a legend of the returned character codes, in the form

$$c_1 s_1 c_2 s_2 \dots s_n c_{n+1}$$

where  $c_j = \text{cutpoints}[j]$  and  $s_j = \text{symbols}[j]$ .

### Author(s)

Martin Maechler <maechler@stat.math.ethz.ch>

### See Also

[as.character](#)

### Examples

```
ii <- 0:8; names(ii) <- ii
symnum(ii, cut= 2*(0:4), sym = c(".", "-", "+", "$"))
symnum(ii, cut= 2*(0:4), sym = c(".", "-", "+", "$"), show.max=TRUE)

symnum(1:12 %% 3 == 0)# use for logical

##-- Symbolic correlation matrices:
symnum(cor(attitude), diag = FALSE)
symnum(cor(attitude), abbr.= NULL)
symnum(cor(attitude), abbr.= FALSE)
symnum(cor(attitude), abbr.= 2)

symnum(cor(rbind(1, rnorm(25), rnorm(25)^2)))
symnum(cor(matrix(rexp(30, 1), 5, 18))) # <<-- PATTERN ! --
symnum(cm1 <- cor(matrix(rnorm(90) , 5, 18))) # < White Noise SMALL n
symnum(cm1, diag=FALSE)
symnum(cm2 <- cor(matrix(rnorm(900), 50, 18))) # < White Noise "BIG" n
symnum(cm2, lower=FALSE)

## NA's:
Cm <- cor(matrix(rnorm(60), 10, 6)); Cm[c(3,6), 2] <- NA
symnum(Cm, show.max=NULL)

## Graphical P-values (aka "significance stars"):
pval <- rev(sort(c(outer(1:6, 10^-(1:3)))))
symp <- symnum(pval, corr=FALSE,
               cutpoints = c(0, .001, .01, .05, .1, 1),
               symbols = c("****", "***", "**", ".", " "))
noquote(cbind(P.val = format(pval), Signif= symp))
```

---

t.test

*Student's t-Test*


---

### Description

Performs one and two sample t-tests on vectors of data.

### Usage

```
t.test(x, ...)

## Default S3 method:
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)

## S3 method for class 'formula':
t.test(formula, data, subset, na.action, ...)
```

### Arguments

x	a numeric vector of data values.
y	an optional numeric vector data values.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
mu	a number indicating the true value of the mean (or difference in means if you are performing a two sample test).
paired	a logical indicating whether you want a paired t-test.
var.equal	a logical variable indicating whether to treat the two variances as being equal. If TRUE then the pooled variance is used to estimate the variance otherwise the Welch (or Satterthwaite) approximation to the degrees of freedom is used.
conf.level	confidence level of the interval.
formula	a formula of the form lhs ~ rhs where lhs is a numeric variable giving the data values and rhs a factor with two levels giving the corresponding groups.
data	an optional data frame containing the variables in the model formula.
subset	an optional vector specifying a subset of observations to be used.
na.action	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
...	further arguments to be passed to or from methods.

**Details**

The formula interface is only applicable for the 2-sample tests.

If `paired` is `TRUE` then both `x` and `y` must be specified and they must be the same length. Missing values are removed (in pairs if `paired` is `TRUE`). If `var.equal` is `TRUE` then the pooled estimate of the variance is used. By default, if `var.equal` is `FALSE` then the variance is estimated separately for both groups and the Welch modification to the degrees of freedom is used.

If the input data are effectively constant (compared to the larger of the two means) an error is generated.

**Value**

A list with class "htest" containing the following components:

<code>statistic</code>	the value of the t-statistic.
<code>parameter</code>	the degrees of freedom for the t-statistic.
<code>p.value</code>	the p-value for the test.
<code>conf.int</code>	a confidence interval for the mean appropriate to the specified alternative hypothesis.
<code>estimate</code>	the estimated mean or difference in means depending on whether it was a one-sample test or a two-sample test.
<code>null.value</code>	the specified hypothesized value of the mean or mean difference depending on whether it was a one-sample test or a two-sample test.
<code>alternative</code>	a character string describing the alternative hypothesis.
<code>method</code>	a character string indicating what type of t-test was performed.
<code>data.name</code>	a character string giving the name(s) of the data.

**See Also**

[prop.test](#)

**Examples**

```
t.test(1:10,y=c(7:20))      # P = .00001855
t.test(1:10,y=c(7:20, 200)) # P = .1245    -- NOT significant anymore

## Classical example: Student's sleep data
plot(extra ~ group, data = sleep)
## Traditional interface
with(sleep, t.test(extra[group == 1], extra[group == 2]))
## Formula interface
t.test(extra ~ group, data = sleep)
```

**Description**

Density, distribution function, quantile function and random generation for the *t* distribution with *df* degrees of freedom (and optional noncentrality parameter *ncp*).

**Usage**

```
dt(x, df, ncp=0, log = FALSE)
pt(q, df, ncp=0, lower.tail = TRUE, log.p = FALSE)
qt(p, df, lower.tail = TRUE, log.p = FALSE)
rt(n, df)
```

**Arguments**

*x*, *q*            vector of quantiles.  
*p*                vector of probabilities.  
*n*                number of observations. If `length(n) > 1`, the length is taken to be the number required.  
*df*               degrees of freedom ( $> 0$ , maybe non-integer).  
*ncp*              non-centrality parameter  $\delta$ ; currently for `pt()` and `dt()`, only for `ncp <= 37.62`.  
*log*, *log.p*      logical; if TRUE, probabilities *p* are given as `log(p)`.  
*lower.tail*      logical; if TRUE (default), probabilities are  $P[X \leq x]$ , otherwise,  $P[X > x]$ .

**Details**

The *t* distribution with *df* =  $\nu$  degrees of freedom has density

$$f(x) = \frac{\Gamma((\nu + 1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}(1 + x^2/\nu)^{-(\nu+1)/2}$$

for all real *x*. It has mean 0 (for  $\nu > 1$ ) and variance  $\frac{\nu}{\nu-2}$  (for  $\nu > 2$ ).

The general *non-central t* with parameters  $(\nu, \delta) = (\text{df}, \text{ncp})$  is defined as the distribution of  $T_\nu(\delta) := \frac{U+\delta}{\chi_\nu/\sqrt{\nu}}$  where *U* and  $\chi_\nu$  are independent random variables,  $U \sim \mathcal{N}(0, 1)$ , and  $\chi_\nu^2$  is chi-squared, see [pchisq](#).

The most used applications are power calculations for *t*-tests:

Let  $T = \frac{\bar{X}-\mu_0}{S/\sqrt{n}}$  where  $\bar{X}$  is the [mean](#) and *S* the sample standard deviation ([sd](#)) of  $X_1, X_2, \dots, X_n$  which are i.i.d.  $N(\mu, \sigma^2)$ . Then *T* is distributed as non-centrally *t* with *df* =  $n - 1$  degrees of freedom and non-centrality parameter  $\text{ncp} = (\mu - \mu_0)\sqrt{n}/\sigma$ .

**Value**

dt gives the density, pt gives the distribution function, qt gives the quantile function, and rt generates random deviates.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole. (except non-central versions.)

Lenth, R. V. (1989). *Algorithm AS 243* — Cumulative distribution function of the non-central  $t$  distribution, *Appl. Statist.* **38**, 185–189.

**See Also**

[df](#) for the F distribution.

**Examples**

```
1 - pt(1:5, df = 1)
qt(.975, df = c(1:10,20,50,100,1000))

tt <- seq(0,10, len=21)
ncp <- seq(0,6, len=31)
ptn <- outer(tt,ncp, function(t,d) pt(t, df = 3, ncp=d))
image(tt,ncp,ptn, zlim=c(0,1),main=t.tit <- "Non-central t - Probabilities")
persp(tt,ncp,ptn, zlim=0:1, r=2, phi=20, theta=200, main=t.tit,
       xlab = "t", ylab = "noncentrality parameter", zlab = "Pr(T <= t)")

op <- par(yaxs="i")
plot(function(x) dt(x, df = 3, ncp = 2), -3, 11, ylim = c(0, 0.32),
      main="Non-central t - Density")
par(op)
```

---

termplot

*Plot regression terms*


---

**Description**

Plots regression terms against their predictors, optionally with standard errors and partial residuals added.

**Usage**

```
termplot(model, data=NULL, envir=environment(formula(model)),
         partial.resid=FALSE, rug=FALSE,
         terms=NULL, se=FALSE, xlabs=NULL, ylabs=NULL, main = NULL,
         col.term = 2, lwd.term = 1.5,
         col.se = "orange", lty.se = 2, lwd.se = 1,
         col.res = "gray", cex = 1, pch = par("pch"),
```

```
col.smth = "darkred", lty.smth=2, span.smth=2/3,
ask = interactive() && nb.fig < n.tms && .Device != "postscript",
use.factor.levels=TRUE, smooth=NULL,
...)
```

## Arguments

model	fitted model object
data	data frame in which variables in model can be found
envir	environment in which variables in model can be found
partial.resid	logical; should partial residuals be plotted?
rug	add <a href="#">rugplots</a> (jittered 1-d histograms) to the axes?
terms	which terms to plot (default NULL means all terms)
se	plot pointwise standard errors?
xlabs	vector of labels for the x axes
ylabs	vector of labels for the y axes
main	logical, or vector of main titles; if TRUE, the model's call is taken as main title, NULL or FALSE mean no titles.
col.term, lwd.term	color and line width for the "term curve", see <a href="#">lines</a> .
col.se, lty.se, lwd.se	color, line type and line width for the "twice-standard-error curve" when se = TRUE.
col.res, cex, pch	color, plotting character expansion and type for partial residuals, when partial.resid = TRUE, see <a href="#">points</a> .
ask	logical; if TRUE, the user is <i>asked</i> before each plot, see <a href="#">par</a> (ask= . ).
use.factor.levels	Should x-axis ticks use factor levels or numbers for factor terms?
smooth	NULL or a function with the same arguments as <a href="#">panel.smooth</a> to draw a smooth through the partial residuals for non-factor terms
lty.smth, col.smth, span.smth	Passed to smooth
...	other graphical parameters

## Details

The model object must have a `predict` method that accepts `type=terms`, eg [glm](#) in the **base** package, [coxph](#) and [survreg](#) in the **survival** package.

For the `partial.resid=TRUE` option it must have a `residuals` method that accepts `type="partial"`, which [lm](#) and [glm](#) do.

The data argument should rarely be needed, but in some cases `termplot` may be unable to reconstruct the original data frame. Using `na.action=na.exclude` makes these problems less likely.

Nothing sensible happens for interaction terms.

**See Also**

For (generalized) linear models, [plot.lm](#) and [predict.glm](#).

**Examples**

```
had.splines <- "package:splines" %in% search()
if(!had.splines) rs <- require(splines)
x <- 1:100
z <- factor(rep(LETTERS[1:4],25))
y <- rnorm(100,sin(x/10)+as.numeric(z))
model <- glm(y ~ ns(x,6) + z)

par(mfrow=c(2,2)) ## 2 x 2 plots for same model :
termpplot(model, main = paste("termpplot( ", deparse(model$call), " ...)") )
termpplot(model, rug=TRUE)
termpplot(model, partial=TRUE, se = TRUE, main = TRUE)
termpplot(model, partial=TRUE, smooth=panel.smooth,span.smth=1/4)
if(!had.splines && rs) detach("package:splines")
```

---

terms

---

*Model Terms*


---

**Description**

The function `terms` is a generic function which can be used to extract *terms* objects from various kinds of R data objects.

**Usage**

```
terms(x, ...)
```

**Arguments**

`x` object used to select a method to dispatch.  
`...` further arguments passed to or from other methods.

**Details**

There are methods for classes "aovlist", and "terms" "formula" (see [terms.formula](#)): the default method just extracts the `terms` component of the object (if any).

**Value**

An object of class `c("terms", "formula")` which contains the *terms* representation of a symbolic model. See [terms.object](#) for its structure.

**References**

Chambers, J. M. and Hastie, T. J. (1992) *Statistical models*. Chapter 2 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[terms.object](#), [terms.formula](#), [lm](#), [glm](#), [formula](#).

---

terms.formula      *Construct a terms Object from a Formula*

---

**Description**

This function takes a formula and some optional arguments and constructs a terms object. The terms object can then be used to construct a [model.matrix](#).

**Usage**

```
## S3 method for class 'formula':
terms(x, specials = NULL, abb = NULL, data = NULL, neg.out = TRUE,
      keep.order = FALSE, simplify = FALSE, ...)
```

**Arguments**

x	a formula.
specials	which functions in the formula should be marked as special in the terms object.
abb	Not implemented in R.
data	a data frame from which the meaning of the special symbol . can be inferred. It is unused if there is no . in the formula.
neg.out	Not implemented in R.
keep.order	a logical value indicating whether the terms should keep their positions. If FALSE the terms are reordered so that main effects come first, followed by the interactions, all second-order, all third-order and so on. Effects of a given order are kept in the order specified.
simplify	should the formula be expanded and simplified, the pre-1.7.0 behaviour?
...	further arguments passed to or from other methods.

**Details**

Not all of the options work in the same way that they do in S and not all are implemented.

**Value**

A [terms.object](#) object is returned. The object itself is the re-ordered (unless `keep.order = TRUE`) formula. In all cases variables within an interaction term in the formula are re-ordered by the ordering of the "variables" attribute, which is the order in which the variables occur in the formula.

**See Also**

[terms](#), [terms.object](#)

---

terms.object	<i>Description of Terms Objects</i>
--------------	-------------------------------------

---

**Description**

An object of class [terms](#) holds information about a model. Usually the model was specified in terms of a [formula](#) and that formula was used to determine the terms object.

**Value**

The object itself is simply the formula supplied to the call of [terms.formula](#). The object has a number of attributes and they are used to construct the model frame:

factors	A matrix of variables by terms showing which variables appear in which terms. The entries are 0 if the variable does not occur in the term, 1 if it does occur and should be coded by contrasts, and 2 if it occurs and should be coded via dummy variables for all levels (as when an intercept or lower-order term is missing). If there are no terms other than an intercept and offsets, this is <code>numeric(0)</code> .
term.labels	A character vector containing the labels for each of the terms in the model, except for offsets. Non-syntactic names will be quoted by backticks.
variables	A call to <code>list</code> of the variables in the model.
intercept	Either 0, indicating no intercept is to be fit, or 1 indicating that an intercept is to be fit.
order	A vector of the same length as <code>term.labels</code> indicating the order of interaction for each term
response	The index of the variable (in <code>variables</code> ) of the response (the left hand side of the formula). Zero, if there is no response.
offset	If the model contains <a href="#">offset</a> terms there is an <code>offset</code> attribute indicating which variable(s) are offsets
specials	If the <code>specials</code> argument was given to <a href="#">terms.formula</a> there is a <code>specials</code> attribute, a list of vectors indicating the terms that contain these special functions.
dataClasses	optional. A named character vector giving the classes (as given by <code>.MFclass</code> ) of the variables used in a fit.

The object has class `c("terms", "formula")`.

**Note**

These objects are different from those found in S. In particular there is no `formula` attribute, instead the object is itself a formula. Thus, the mode of a terms object is different as well.

Examples of the `specials` argument can be seen in the [aov](#) and [coxph](#) functions.

**See Also**

[terms](#), [formula](#).

---

time *Sampling Times of Time Series*

---

**Description**

`time` creates the vector of times at which a time series was sampled.

`cycle` gives the positions in the cycle of each observation.

`frequency` returns the number of samples per unit time and `deltat` the time interval between observations (see [ts](#)).

**Usage**

```
time(x, ...)  
## Default S3 method:  
time(x, offset=0, ...)  
  
cycle(x, ...)  
frequency(x, ...)  
deltat(x, ...)
```

**Arguments**

<code>x</code>	a univariate or multivariate time-series, or a vector or matrix.
<code>offset</code>	can be used to indicate when sampling took place in the time unit. 0 (the default) indicates the start of the unit, 0.5 the middle and 1 the end of the interval.
<code>...</code>	extra arguments for future methods.

**Details**

These are all generic functions, which will use the `tsp` attribute of `x` if it exists. `time` and `cycle` have methods for class `ts` that coerce the result to that class.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[ts](#), [start](#), [tsp](#), [window](#).

[date](#) for clock time, [system.time](#) for CPU usage.

**Examples**

```
cycle(presidents)
# a simple series plot: c() makes the x and y arguments into vectors
plot(c(time(presidents)), c(presidents), type="l")
```

toeplitz

*Form Symmetric Toeplitz Matrix***Description**

Forms a symmetric Toeplitz matrix given its first row.

**Usage**

```
toeplitz(x)
```

**Arguments**

`x` the first row to form the Toeplitz matrix.

**Value**

The Toeplitz matrix.

**Author(s)**

A. Trapletti

**Examples**

```
x <- 1:5
toeplitz(x)
```

ts

*Time-Series Objects***Description**

The function `ts` is used to create time-series objects.

`as.ts` and `is.ts` coerce an object to a time-series and test whether an object is a time series.

**Usage**

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
    deltat = 1, ts.eps = getOption("ts.eps"), class = , names = )
as.ts(x)
is.ts(x)
```

**Arguments**

<code>data</code>	a numeric vector or matrix of the observed time-series values. A data frame will be coerced to a numeric matrix via <code>data.matrix</code> .
<code>start</code>	the time of the first observation. Either a single number or a vector of two integers, which specify a natural time unit and a (1-based) number of samples into the time unit. See the examples for the use of the second form.
<code>end</code>	the time of the last observation, specified in the same way as <code>start</code> .
<code>frequency</code>	the number of observations per unit of time.
<code>deltat</code>	the fraction of the sampling period between successive observations; e.g., 1/12 for monthly data. Only one of <code>frequency</code> or <code>deltat</code> should be provided.
<code>ts.eps</code>	time series comparison tolerance. Frequencies are considered equal if their absolute difference is less than <code>ts.eps</code> .
<code>class</code>	class to be given to the result, or none if <code>NULL</code> or <code>"none"</code> . The default is <code>"ts"</code> for a single series, <code>c("mts", "ts")</code> for multiple series.
<code>names</code>	a character vector of names for the series in a multiple series: defaults to the colnames of <code>data</code> , or <code>Series 1, Series 2, ...</code>
<code>x</code>	an arbitrary R object.

**Details**

The function `ts` is used to create time-series objects. These are vector or matrices with class of `"ts"` (and additional attributes) which represent data which has been sampled at equispaced points in time. In the matrix case, each column of the matrix `data` is assumed to contain a single (univariate) time series. Time series must have an least one observation, and although they need not be numeric there is very limited support for non-numeric series.

Class `"ts"` has a number of methods. In particular arithmetic will attempt to align time axes, and subsetting to extract subsets of series can be used (e.g., `EuStockMarkets[, "DAX"]`). However, subsetting the first (or only) dimension will return a matrix or vector, as will matrix subsetting. There is a method for `t` that transposes the series as a matrix (a one-column matrix if a vector) and hence returns a result that does not inherit from class `"ts"`.

The value of argument `frequency` is used when the series is sampled an integral number of times in each unit time interval. For example, one could use a value of 7 for `frequency` when the data are sampled daily, and the natural time period is a week, or 12 when the data are sampled monthly and the natural time period is a year. Values of 4 and 12 are assumed in (e.g.) `print` methods to imply a quarterly and monthly series respectively.

`as.ts` will use the `tsp` attribute of the object if it has one to set the start and end times and frequency.

`is.ts` tests if an object is a time series. It is generic: you can write methods to handle specific classes of objects, see [InternalMethods](#).

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

`ts`, `frequency`, `start`, `end`, `time`, `window`; `print.ts`, the print method for time series objects; `plot.ts`, the plot method for time series objects.

**Examples**

```
ts(1:10, frequency = 4, start = c(1959, 2)) # 2nd Quarter of 1959
print( ts(1:10, freq = 7, start = c(12, 2)), calendar = TRUE) # print.ts(.)
## Using July 1954 as start date:
gnp <- ts(cumsum(1 + round(rnorm(100), 2)),
          start = c(1954, 7), frequency = 12)
plot(gnp) # using 'plot.ts' for time-series plot

## Multivariate
z <- ts(matrix(rnorm(300), 100, 3), start=c(1961, 1), frequency=12)
class(z)
plot(z)
plot(z, plot.type="single", lty=1:3)

## A phase plot:
plot(nhtemp, c(nhtemp[-1], NA), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## a clearer way to do this would be
## Not run:
plot(nhtemp, lag(nhtemp, 1), cex = .8, col="blue",
     main = "Lag plot of New Haven temperatures")
## End(Not run)
```

---

ts-methods

*Methods for Time Series Objects*


---

**Description**

Methods for objects of class "ts", typically the result of `ts`.

**Usage**

```
## S3 method for class 'ts':
diff(x, lag = 1, differences = 1, ...)

## S3 method for class 'ts':
na.omit(object, ...)
```

**Arguments**

`x` an object of class "ts" containing the values to be differenced.  
`lag` an integer indicating which lag to use.  
`differences` an integer indicating the order of the difference.

object            a univariate or multivariate time series.  
...                further arguments to be passed to or from methods.

### Details

The `na.omit` method omits initial and final segments with missing values in one or more of the series. ‘Internal’ missing values will lead to failure.

### Value

For the `na.omit` method, a time series without missing values. The class of `object` will be preserved.

### See Also

[diff](#); [na.omit](#), [na.fail](#), [na.contiguous](#).

---

ts.plot

*Plot Multiple Time Series*

---

### Description

Plot several time series on a common plot. Unlike [plot.ts](#) the series can have a different time bases, but they should have the same frequency.

### Usage

```
ts.plot(..., gpars = list())
```

### Arguments

...                one or more univariate or multivariate time series.  
gpars              list of named graphics parameters to be passed to the plotting functions. Those commonly used can be supplied directly in ...

### Value

None.

### Note

Although this can be used for a single time series, `plot` is easier to use and is preferred.

### See Also

[plot.ts](#)

**Examples**

```
ts.plot(ldeaths, mdeaths, fdeaths,
        gpars=list(xlab="year", ylab="deaths", lty=c(1:3)))
```

---

<code>ts.union</code>	<i>Bind Two or More Time Series</i>
-----------------------	-------------------------------------

---

**Description**

Bind time series which have a common frequency. `ts.union` pads with NAs to the total time coverage, `ts.intersect` restricts to the time covered by all the series.

**Usage**

```
ts.intersect(..., dframe = FALSE)
ts.union(..., dframe = FALSE)
```

**Arguments**

`...` two or more univariate or multivariate time series, or objects which can coerced to time series.

`dframe` logical; if TRUE return the result as a data frame.

**Details**

As a special case, `...` can contain vectors or matrices of the same length as the combined time series of the time series present, as well as those of a single row.

**Value**

A time series object if `dframe` is FALSE, otherwise a data frame.

**See Also**

[cbind](#).

**Examples**

```
ts.union(mdeaths, fdeaths)
cbind(mdeaths, fdeaths) # same as the previous line
ts.intersect(window(mdeaths, 1976), window(fdeaths, 1974, 1978))

sales1 <- ts.union(BJsales, lead = BJsales.lead)
ts.intersect(sales1, lead3 = lag(BJsales.lead, -3))
```

**Description**

A generic function to plot time-series diagnostics.

**Usage**

```
tsdiag(object, gof.lag, ...)
```

**Arguments**

<code>object</code>	a fitted time-series model
<code>gof.lag</code>	the maximum number of lags for a Portmanteau goodness-of-fit test
<code>...</code>	further arguments to be passed to particular methods

**Details**

This is a generic function. It will generally plot the residuals, often standadized, the autocorrelation function of the residuals, and the p-values of a Portmanteau test for all lags up to `gof.lag`.

The methods for [arima](#) and [StructTS](#) objects plots residuals scaled by the estimate of their (individual) variance, and use the Ljung–Box version of the portmanteau test.

**Value**

None. Diagnostics are plotted.

**See Also**

[arima](#), [StructTS](#), [Box.test](#)

**Examples**

```
## Not run: fit <- arima(lh, c(1,0,0))
tsdiag(fit)

## see also examples(arima)

(fit <- StructTS(log10(JohnsonJohnson), type="BSM"))
tsdiag(fit)
## End(Not run)
```

---

`tsp`*Tsp Attribute of Time-Series-like Objects*

---

### Description

`tsp` returns the `tsp` attribute (or `NULL`). It is included for compatibility with S version 2. `tsp<-` sets the `tsp` attribute. `hasTsp` ensures `x` has a `tsp` attribute, by adding one if needed.

### Usage

```
tsp(x)
tsp(x) <- value
hasTsp(x)
```

### Arguments

`x` a vector or matrix or univariate or multivariate time-series.  
`value` a numeric vector of length 3 or `NULL`.

### Details

The `tsp` attribute was previously described here as `c(start(x), end(x), frequency(x))`, but this is incorrect. It gives the start time *in time units*, the end time and the frequency.

Assignments are checked for consistency.

Assigning `NULL` which removes the `tsp` attribute *and* any "ts" class of `x`.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

[ts](#), [time](#), [start](#).

---

tsSmooth	<i>Use Fixed-Interval Smoothing on Time Series</i>
----------	--

---

### Description

Performs fixed-interval smoothing on a univariate time series via a state-space model. Fixed-interval smoothing gives the best estimate of the state at each time point based on the whole observed series.

### Usage

```
tsSmooth(object, ...)
```

### Arguments

object	a time-series fit. Currently only class " <a href="#">StructTS</a> " is supported
...	possible arguments for future methods.

### Value

A time series, with as many dimensions as the state space and results at each time point of the original series. (For seasonal models, only the current seasonal component is returned.)

### Author(s)

B. D. Ripley

### References

Durbin, J. and Koopman, S. J. (2001) *Time Series Analysis by State Space Methods*. Oxford University Press.

### See Also

[KalmanSmooth](#), [StructTS](#).

For examples consult [AirPassengers](#), [JohnsonJohnson](#) and [Nile](#).

---

 Tukey

*The Studentized Range Distribution*


---

### Description

Functions on the distribution of the studentized range,  $R/s$ , where  $R$  is the range of a standard normal sample of size  $n$  and  $s^2$  is independently distributed as chi-squared with  $df$  degrees of freedom, see [pchisq](#).

### Usage

```
ptukey(q, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
qtukey(p, nmeans, df, nranges = 1, lower.tail = TRUE, log.p = FALSE)
```

### Arguments

<code>q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>nmeans</code>	sample size for range (same for each group).
<code>df</code>	degrees of freedom for $s$ (see below).
<code>nranges</code>	number of <i>groups</i> whose <b>maximum</b> range is considered.
<code>log.p</code>	logical; if TRUE, probabilities $p$ are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

### Details

If  $n_g = \text{nranges}$  is greater than one,  $R$  is the *maximum* of  $n_g$  groups of  $n\text{means}$  observations each.

### Value

`ptukey` gives the distribution function and `qtukey` its inverse, the quantile function.

### Note

A Legendre 16-point formula is used for the integral of `ptukey`. The computations are relatively expensive, especially for `qtukey` which uses a simple secant method for finding the inverse of `ptukey`. `qtukey` will be accurate to the 4th decimal place.

### References

Copenhaver, Margaret Diponzio and Holland, Burt S. (1988) Multiple comparisons of simple effects in the two-way analysis of variance with fixed effects. *Journal of Statistical Computation and Simulation*, **30**, 1–15.

**See Also**

[pnorm](#) and [qnorm](#) for the corresponding functions for the normal distribution.

**Examples**

```
if(interactive())
  curve(ptukey(x, nm=6, df=5), from=-1, to=8, n=101)
(ptt <- ptukey(0:10, 2, df= 5))
(qtt <- qtkey(.95, 2, df= 2:11))
## The precision may be not much more than about 8 digits:
summary(abs(.95 - ptukey(qtt,2, df = 2:11)))
```

---

 TukeyHSD

---

*Compute Tukey Honest Significant Differences*


---

**Description**

Create a set of confidence intervals on the differences between the means of the levels of a factor with the specified family-wise probability of coverage. The intervals are based on the Studentized range statistic, Tukey's 'Honest Significant Difference' method. There is a `plot` method.

**Usage**

```
TukeyHSD(x, which, ordered = FALSE, conf.level = 0.95, ...)
```

**Arguments**

<code>x</code>	A fitted model object, usually an <a href="#">aov</a> fit.
<code>which</code>	A list of terms in the fitted model for which the intervals should be calculated. Defaults to all the terms.
<code>ordered</code>	A logical value indicating if the levels of the factor should be ordered according to increasing average in the sample before taking differences. If <code>ordered</code> is true then the calculated differences in the means will all be positive. The significant differences will be those for which the <code>lwr</code> end point is positive.
<code>conf.level</code>	A numeric value between zero and one giving the family-wise confidence level to use.
<code>...</code>	Optional additional arguments. None are used at present.

**Details**

When comparing the means for the levels of a factor in an analysis of variance, a simple comparison using t-tests will inflate the probability of declaring a significant difference when it is not in fact present. This because the intervals are calculated with a given coverage probability for each interval but the interpretation of the coverage is usually with respect to the entire family of intervals.

John Tukey introduced intervals based on the range of the sample means rather than the individual differences. The intervals returned by this function are based on this Studentized range statistics.

Technically the intervals constructed in this way would only apply to balanced designs where there are the same number of observations made at each level of the factor. This function incorporates an adjustment for sample size that produces sensible intervals for mildly unbalanced designs.

If `which` specifies non-factor terms these will be dropped with a warning: if no terms are left this is an error.

### Value

A list with one component for each term requested in `which`. Each component is a matrix with columns `diff` giving the difference in the observed means, `lwr` giving the lower end point of the interval, and `upr` giving the upper end point.

### Author(s)

Douglas Bates

### References

Miller, R. G. (1981) *Simultaneous Statistical Inference*. Springer.

Yandell, B. S. (1997) *Practical Data Analysis for Designed Experiments*. Chapman & Hall.

### See Also

[aov](#), [qtukey](#), [model.tables](#)

### Examples

```
summary(fml <- aov(breaks ~ wool + tension, data = warpbreaks))
TukeyHSD(fml, "tension", ordered = TRUE)
plot(TukeyHSD(fml, "tension"))
```

---

Uniform

*The Uniform Distribution*

---

### Description

These functions provide information about the uniform distribution on the interval from `min` to `max`. `dunif` gives the density, `pnif` gives the distribution function `qunif` gives the quantile function and `runif` generates random deviates.

### Usage

```
dunif(x, min=0, max=1, log = FALSE)
pnif(q, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
runif(n, min=0, max=1)
```

**Arguments**

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>min, max</code>	lower and upper limits of the distribution.
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

If `min` or `max` are not specified they assume the default values of 0 and 1 respectively.

The uniform distribution has density

$$f(x) = \frac{1}{max - min}$$

for  $min \leq x \leq max$ .

For the case of  $u := min == max$ , the limit case of  $X \equiv u$  is assumed.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[.Random.seed](#) about random number generation, [rnorm](#), etc for other distributions.

**Examples**

```
u <- runif(20)

## The following relations always hold :
punif(u) == u
dunif(u) == 1

var(runif(10000))#- ~ = 1/12 = .08333
```

uniroot

*One Dimensional Root (Zero) Finding***Description**

The function `uniroot` searches the interval from `lower` to `upper` for a root (i.e., zero) of the function `f` with respect to its first argument.

**Usage**

```
uniroot(f, interval, lower = min(interval), upper = max(interval),
        tol = .Machine$double.eps^0.25, maxiter = 1000, ...)
```

**Arguments**

<code>f</code>	the function for which the root is sought.
<code>interval</code>	a vector containing the end-points of the interval to be searched for the root.
<code>lower</code>	the lower end point of the interval to be searched.
<code>upper</code>	the upper end point of the interval to be searched.
<code>tol</code>	the desired accuracy (convergence tolerance).
<code>maxiter</code>	the maximum number of iterations.
<code>...</code>	additional arguments to <code>f</code> .

**Details**

Either `interval` or both `lower` and `upper` must be specified. The function uses Fortran sub-routine "zeroin" (from Netlib) based on algorithms given in the reference below.

If the algorithm does not converge in `maxiter` steps, a warning is printed and the current approximation is returned.

**Value**

A list with four components: `root` and `f.root` give the location of the root and the value of the function evaluated at that point. `iter` and `estim.prec` give the number of iterations used and an approximate estimated precision for `root`.

**References**

Brent, R. (1973) *Algorithms for Minimization without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall.

**See Also**

[polyroot](#) for all complex roots of a polynomial; [optimize](#), [nlm](#).

**Examples**

```
f <- function (x,a) x - a
str(xmin <- uniroot(f, c(0, 1), tol = 0.0001, a = 1/3))
str(uniroot(function(x) x*(x^2-1) + .5, low = -2, up = 2, tol = 0.0001),
     dig = 10)
str(uniroot(function(x) x*(x^2-1) + .5, low = -2, up = 2, tol = 1e-10),
     dig = 10)

## Find the smallest value x for which exp(x) > 0 (numerically):
r <- uniroot(function(x) 1e80*exp(x) -1e-300,, -1000,0, tol=1e-20)
str(r, digits= 15)##> around -745.1332191

exp(r$r)          # = 0, but not for r$r * 0.999...
minexp <- r$r * (1 - .Machine$double.eps)
exp(minexp)       # typically denormalized
```

update

*Update and Re-fit a Model Call***Description**

`update` will update and (by default) re-fit a model. It does this by extracting the call stored in the object, updating the call and (by default) evaluating that call. Sometimes it is useful to call `update` with only one argument, for example if the data frame has been corrected.

**Usage**

```
update(object, ...)

## Default S3 method:
update(object, formula., ..., evaluate = TRUE)
```

**Arguments**

<code>object</code>	An existing fit from a model function such as <code>lm</code> , <code>glm</code> and many others.
<code>formula.</code>	Changes to the formula – see <code>update.formula</code> for details.
<code>...</code>	Additional arguments to the call, or arguments with changed values. Use <code>name=NULL</code> to remove the argument name.
<code>evaluate</code>	If true evaluate the new call else return the call.

**Value**

If `evaluate = TRUE` the fitted object, otherwise the updated call.

**References**

Chambers, J. M. (1992) *Linear models*. Chapter 4 of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

[update.formula](#)

**Examples**

```
oldcon <- options(contrasts = c("contr.treatment", "contr.poly"))
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D9
summary(lm.D90 <- update(lm.D9, . ~ . - 1))
options(contrasts = c("contr.helmert", "contr.poly"))
update(lm.D9)
options(oldcon)
```

---

update.formula      *Model Updating*

---

**Description**

`update.formula` is used to update model formulae. This typically involves adding or dropping terms, but updates can be more general.

**Usage**

```
## S3 method for class 'formula':
update(old, new, ...)
```

**Arguments**

<code>old</code>	a model formula to be updated.
<code>new</code>	a formula giving a template which specifies how to update.
<code>...</code>	further arguments passed to or from other methods.

**Details**

The function works by first identifying the *left-hand side* and *right-hand side* of the `old` formula. It then examines the new formula and substitutes the *lhs* of the `old` formula for any occurrence of "." on the left of `new`, and substitutes the *rhs* of the `old` formula for any occurrence of "." on the right of `new`.

**Value**

The updated formula is returned.

**See Also**

[terms](#), [model.matrix](#).

**Examples**

```
update(y ~ x,      ~ . + x2) #> y ~ x + x2
update(y ~ x, log(.) ~ . ) #> log(y) ~ x
```

---

<code>var.test</code>	<i>F Test to Compare Two Variances</i>
-----------------------	--

---

**Description**

Performs an F test to compare the variances of two samples from normal populations.

**Usage**

```
var.test(x, ...)
```

## Default S3 method:

```
var.test(x, y, ratio = 1,
         alternative = c("two.sided", "less", "greater"),
         conf.level = 0.95, ...)
```

## S3 method for class 'formula':

```
var.test(formula, data, subset, na.action, ...)
```

**Arguments**

<code>x, y</code>	numeric vectors of data values, or fitted linear model objects (inheriting from class "lm").
<code>ratio</code>	the hypothesized ratio of the population variances of <code>x</code> and <code>y</code> .
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
<code>conf.level</code>	confidence level for the returned confidence interval.
<code>formula</code>	a formula of the form <code>lhs ~ rhs</code> where <code>lhs</code> is a numeric variable giving the data values and <code>rhs</code> a factor with two levels giving the corresponding groups.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

**Details**

The null hypothesis is that the ratio of the variances of the populations from which  $x$  and  $y$  were drawn, or in the data to which the linear models  $x$  and  $y$  were fitted, is equal to `ratio`.

**Value**

A list with class "htest" containing the following components:

<code>statistic</code>	the value of the F test statistic.
<code>parameter</code>	the degrees of the freedom of the F distribution of the test statistic.
<code>p.value</code>	the p-value of the test.
<code>conf.int</code>	a confidence interval for the ratio of the population variances.
<code>estimate</code>	the ratio of the sample variances of $x$ and $y$ .
<code>null.value</code>	the ratio of population variances under the null.
<code>alternative</code>	a character string describing the alternative hypothesis.
<code>method</code>	the character string "F test to compare two variances".
<code>data.name</code>	a character string giving the names of the data.

**See Also**

[bartlett.test](#) for testing homogeneity of variances in more than two samples from normal distributions; [ansari.test](#) and [mood.test](#) for two rank based (nonparametric) two-sample tests for difference in scale.

**Examples**

```
x <- rnorm(50, mean = 0, sd = 2)
y <- rnorm(30, mean = 1, sd = 1)
var.test(x, y) # Do x and y have the same variance?
var.test(lm(x ~ 1), lm(y ~ 1)) # The same.
```

---

varimax

*Rotation Methods for Factor Analysis*


---

**Description**

These functions 'rotate' loading matrices in factor analysis.

**Usage**

```
varimax(x, normalize = TRUE, eps = 1e-5)
promax(x, m = 4)
```

**Arguments**

<code>x</code>	A loadings matrix, with $p$ rows and $k < p$ columns
<code>m</code>	The power used the target for <code>promax</code> . Values of 2 to 4 are recommended.
<code>normalize</code>	logical. Should Kaiser normalization be performed? If so the rows of <code>x</code> are re-scaled to unit length before rotation, and scaled back afterwards.
<code>eps</code>	The tolerance for stopping: the relative change in the sum of singular values.

**Details**

These seek a ‘rotation’ of the factors `x %*% T` that aims to clarify the structure of the loadings matrix. The matrix `T` is a rotation (possibly with reflection) for `varimax`, but a general linear transformation for `promax`, with the variance of the factors being preserved.

**Value**

A list with components

<code>loadings</code>	The ‘rotated’ loadings matrix, <code>x %*% rotmat</code> , of class "loadings".
<code>rotmat</code>	The ‘rotation’ matrix.

**References**

- Hendrickson, A. E. and White, P. O. (1964) Promax: a quick method for rotation to orthogonal oblique structure. *British Journal of Statistical Psychology*, **17**, 65–70.
- Horst, P. (1965) *Factor Analysis of Data Matrices*. Holt, Rinehart and Winston. Chapter 10.
- Kaiser, H. F. (1958) The varimax criterion for analytic rotation in factor analysis. *Psychometrika* **23**, 187–200.
- Lawley, D. N. and Maxwell, A. E. (1971) *Factor Analysis as a Statistical Method*. Second edition. Butterworths.

**See Also**

[factanal](#), [Harman74.cor](#).

**Examples**

```
## varimax with normalize = TRUE is the default
fa <- factanal( ~., 2, data = swiss)
varimax(loadings(fa), normalize = FALSE)
promax(loadings(fa))
```

---

 vcov

*Calculate Variance-Covariance Matrix for a Fitted Model Object*


---

**Description**

Returns the variance-covariance matrix of the main parameters of a fitted model object.

**Usage**

```
vcov(object, ...)
```

**Arguments**

<code>object</code>	a fitted model object.
<code>...</code>	additional arguments for method functions. For the <code>glm</code> method this can be used to pass a dispersion parameter.

**Details**

This is a generic function. Functions with names beginning in `vcov.` will be methods for this function. Classes with methods for this function include: `lm`, `glm`, `nls`, `lme`, `gls`, `coxph` and `survreg` (the last two in package **survival**).

**Value**

A matrix of the estimated covariances between the parameter estimates in the linear or non-linear predictor of the model.

---

 Weibull

*The Weibull Distribution*


---

**Description**

Density, distribution function, quantile function and random generation for the Weibull distribution with parameters `shape` and `scale`.

**Usage**

```
dweibull(x, shape, scale = 1, log = FALSE)
pweibull(q, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
qweibull(p, shape, scale = 1, lower.tail = TRUE, log.p = FALSE)
rweibull(n, shape, scale = 1)
```

**Arguments**

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) &gt; 1</code> , the length is taken to be the number required.
<code>shape, scale</code>	shape and scale parameters, the latter defaulting to 1.
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

**Details**

The Weibull distribution with shape parameter  $a$  and scale parameter  $\sigma$  has density given by

$$f(x) = (a/\sigma)(x/\sigma)^{a-1} \exp(-(x/\sigma)^a)$$

for  $x > 0$ . The cumulative is  $F(x) = 1 - \exp(-(x/\sigma)^a)$ , the mean is  $E(X) = \sigma\Gamma(1 + 1/a)$ , and the  $Var(X) = \sigma^2(\Gamma(1 + 2/a) - (\Gamma(1 + 1/a))^2)$ .

**Value**

`dweibull` gives the density, `pweibull` gives the distribution function, `qweibull` gives the quantile function, and `rweibull` generates random deviates.

**Note**

The cumulative hazard  $H(t) = -\log(1 - F(t))$  is `-pweibull(t, a, b, lower = FALSE, log = TRUE)` which is just  $H(t) = (t/b)^a$ .

**See Also**

[dexp](#) for the Exponential which is a special case of a Weibull distribution.

**Examples**

```
x <- c(0,rlnorm(50))
all.equal(dweibull(x, shape = 1), dexp(x))
all.equal(pweibull(x, shape = 1, scale = pi), pexp(x, rate = 1/pi))
## Cumulative hazard H():
all.equal(pweibull(x, 2.5, pi, lower=FALSE, log=TRUE), -(x/pi)^2.5, tol=1e-15)
all.equal(qweibull(x/11, shape = 1, scale = pi), qexp(x/11, rate = 1/pi))
```

---

weighted.mean	<i>Weighted Arithmetic Mean</i>
---------------	---------------------------------

---

**Description**

Compute a weighted mean of a numeric vector.

**Usage**

```
weighted.mean(x, w, na.rm = FALSE)
```

**Arguments**

<code>x</code>	a numeric vector containing the values whose mean is to be computed.
<code>w</code>	a vector of weights the same length as <code>x</code> giving the weights to use for each element of <code>x</code> .
<code>na.rm</code>	a logical value indicating whether NA values in <code>x</code> should be stripped before the computation proceeds.

**Details**

If `w` is missing then all elements of `x` are given the same weight.

Missing values in `w` are not handled.

**See Also**

[mean](#)

**Examples**

```
## GPA from Siegel 1994
wt <- c(5, 5, 4, 1)/15
x <- c(3.7, 3.3, 3.5, 2.8)
xm <- weighted.mean(x, wt)
```

---

weighted.residuals	<i>Compute Weighted Residuals</i>
--------------------	-----------------------------------

---

**Description**

Computed weighted residuals from a linear model fit.

**Usage**

```
weighted.residuals(obj, drop0 = TRUE)
```

**Arguments**

`obj` R object, typically of class `lm` or `glm`.  
`drop0` logical. If `TRUE`, drop all cases with `weights == 0`.

**Details**

Weighted residuals are the usual residuals  $R_i$ , multiplied by  $\sqrt{w_i}$ , where  $w_i$  are the `weights` as specified in `lm`'s call.

Dropping cases with weights zero is compatible with `influence` and related functions.

**Value**

Numeric vector of length  $n'$ , where  $n'$  is the number of non-0 weights (`drop0 = TRUE`) or the number of observations, otherwise.

**See Also**

`residuals`, `lm.influence`, etc.

**Examples**

```
example("lm")
all.equal(weighted.residuals(lm.D9),
          residuals(lm.D9))
x <- 1:10
w <- 0:9
y <- rnorm(x)
weighted.residuals(lmxy <- lm(y ~ x, weights = w))
weighted.residuals(lmxy, drop0 = FALSE)
```

---

wilcox.test

*Wilcoxon Rank Sum and Signed Rank Tests*

---

**Description**

Performs one and two sample Wilcoxon tests on vectors of data; the latter is also known as ‘Mann-Whitney’ test.

**Usage**

```
wilcox.test(x, ...)

## Default S3 method:
wilcox.test(x, y = NULL,
            alternative = c("two.sided", "less", "greater"),
            mu = 0, paired = FALSE, exact = NULL, correct = TRUE,
            conf.int = FALSE, conf.level = 0.95, ...)
```

```
## S3 method for class 'formula':
wilcox.test(formula, data, subset, na.action, ...)
```

### Arguments

<code>x</code>	numeric vector of data values.
<code>y</code>	an optional numeric vector of data values.
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
<code>mu</code>	a number specifying an optional location parameter.
<code>paired</code>	a logical indicating whether you want a paired test.
<code>exact</code>	a logical indicating whether an exact p-value should be computed.
<code>correct</code>	a logical indicating whether to apply continuity correction in the normal approximation for the p-value.
<code>conf.int</code>	a logical indicating whether a confidence interval should be computed.
<code>conf.level</code>	confidence level of the interval.
<code>formula</code>	a formula of the form <code>lhs ~ rhs</code> where <code>lhs</code> is a numeric variable giving the data values and <code>rhs</code> a factor with two levels giving the corresponding groups.
<code>data</code>	an optional data frame containing the variables in the model formula.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs. Defaults to <code>getOption("na.action")</code> .
<code>...</code>	further arguments to be passed to or from methods.

### Details

The formula interface is only applicable for the 2-sample tests.

If only `x` is given, or if both `x` and `y` are given and `paired` is `TRUE`, a Wilcoxon signed rank test of the null that the distribution of `x` (in the one sample case) or of `x-y` (in the paired two sample case) is symmetric about `mu` is performed.

Otherwise, if both `x` and `y` are given and `paired` is `FALSE`, a Wilcoxon rank sum test (equivalent to the Mann-Whitney test) is carried out. In this case, the null hypothesis is that the location of the distributions of `x` and `y` differ by `mu`.

By default (if `exact` is not specified), an exact p-value is computed if the samples contain less than 50 finite values and there are no ties. Otherwise, a normal approximation is used.

Optionally (if argument `conf.int` is true), a nonparametric confidence interval and an estimator for the pseudomedian (one-sample case) or for the difference of the location parameters `x-y` is computed. (The pseudomedian of a distribution  $F$  is the median of the distribution of  $(u + v)/2$ , where  $u$  and  $v$  are independent, each with distribution  $F$ . If  $F$  is symmetric, then the pseudomedian and median coincide. See Hollander & Wolfe (1973), page 34.) If exact p-values are available, an exact confidence interval is obtained by the algorithm described in Bauer (1972), and the Hodges-Lehmann estimator is employed. Otherwise, the returned confidence interval and point estimate are based on normal approximations.

**Value**

A list with class "htest" containing the following components:

statistic	the value of the test statistic with a name describing it.
parameter	the parameter(s) for the exact distribution of the test statistic.
p.value	the p-value for the test.
null.value	the location parameter mu.
alternative	a character string describing the alternative hypothesis.
method	the type of test applied.
data.name	a character string giving the names of the data.
conf.int	a confidence interval for the location parameter. (Only present if argument <code>conf.int = TRUE</code> .)
estimate	an estimate of the location parameter. (Only present if argument <code>conf.int = TRUE</code> .)

**References**

Myles Hollander & Douglas A. Wolfe (1973), *Nonparametric statistical inference*. New York: John Wiley & Sons. Pages 27–33 (one-sample), 68–75 (two-sample).

David F. Bauer (1972), Constructing confidence sets using rank statistics. *Journal of the American Statistical Association* **67**, 687–690.

**See Also**

[kruskal.test](#) for testing homogeneity in location parameters in the case of two or more samples; [t.test](#) for a parametric alternative under normality assumptions.

**Examples**

```
## One-sample test.
## Hollander & Wolfe (1973), 29f.
## Hamilton depression scale factor measurements in 9 patients with
## mixed anxiety and depression, taken at the first (x) and second
## (y) visit after initiation of a therapy (administration of a
## tranquilizer).
x <- c(1.83, 0.50, 1.62, 2.48, 1.68, 1.88, 1.55, 3.06, 1.30)
y <- c(0.878, 0.647, 0.598, 2.05, 1.06, 1.29, 1.06, 3.14, 1.29)
wilcox.test(x, y, paired = TRUE, alternative = "greater")
wilcox.test(y - x, alternative = "less") # The same.
wilcox.test(y - x, alternative = "less",
            exact = FALSE, correct = FALSE) # H&W large sample
                                           # approximation

## Two-sample test.
## Hollander & Wolfe (1973), 69f.
## Permeability constants of the human chorioamnion (a placental
## membrane) at term (x) and between 12 to 26 weeks gestational
## age (y). The alternative of interest is greater permeability
```

```
## of the human chorioamnion for the term pregnancy.
x <- c(0.80, 0.83, 1.89, 1.04, 1.45, 1.38, 1.91, 1.64, 0.73, 1.46)
y <- c(1.15, 0.88, 0.90, 0.74, 1.21)
wilcox.test(x, y, alternative = "g")          # greater
wilcox.test(x, y, alternative = "greater",
            exact = FALSE, correct = FALSE) # H&W large sample
                                           # approximation

wilcox.test(rnorm(10), rnorm(10, 2), conf.int = TRUE)

## Formula interface.
boxplot(Ozone ~ Month, data = airquality)
wilcox.test(Ozone ~ Month, data = airquality,
            subset = Month %in% c(5, 8))
```

---

Wilcoxon

*Distribution of the Wilcoxon Rank Sum Statistic*


---

## Description

Density, distribution function, quantile function and random generation for the distribution of the Wilcoxon rank sum statistic obtained from samples with size  $m$  and  $n$ , respectively.

## Usage

```
dwilcox(x, m, n, log = FALSE)
pwilcox(q, m, n, lower.tail = TRUE, log.p = FALSE)
qwilcox(p, m, n, lower.tail = TRUE, log.p = FALSE)
rwilcox(nn, m, n)
```

## Arguments

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>nn</code>	number of observations. If <code>length(nn) &gt; 1</code> , the length is taken to be the number required.
<code>m, n</code>	numbers of observations in the first and second sample, respectively. Can be vectors of positive integers.
<code>log, log.p</code>	logical; if TRUE, probabilities $p$ are given as $\log(p)$ .
<code>lower.tail</code>	logical; if TRUE (default), probabilities are $P[X \leq x]$ , otherwise, $P[X > x]$ .

## Details

This distribution is obtained as follows. Let  $x$  and  $y$  be two random, independent samples of size  $m$  and  $n$ . Then the Wilcoxon rank sum statistic is the number of all pairs  $(x[i], y[j])$  for which  $y[j]$  is not greater than  $x[i]$ . This statistic takes values between 0 and  $m * n$ , and its mean and variance are  $m * n / 2$  and  $m * n * (m + n + 1) / 12$ , respectively.

If any of the first three arguments are vectors, the recycling rule is used to do the calculations for all combinations of the three up to the length of the longest vector.

**Value**

`dwilcox` gives the density, `pwilcox` gives the distribution function, `qwilcox` gives the quantile function, and `rwilcox` generates random deviates.

**Note**

S-PLUS uses a different (but equivalent) definition of the Wilcoxon statistic.

**Author(s)**

Kurt Hornik

**See Also**

[wilcox.test](#) to calculate the statistic from data, find p values and so on.

[dsignrank](#) etc, for the distribution of the *one-sample* Wilcoxon signed rank statistic.

**Examples**

```
x <- -1:(4*6 + 1)
fx <- dwilcox(x, 4, 6)
Fx <- pwilcox(x, 4, 6)

layout(rbind(1,2),width=1,heights=c(3,2))
plot(x, fx,type='h', col="violet",
      main= "Probabilities (density) of Wilcoxon-Statist.(n=6,m=4)")
plot(x, Fx,type="s", col="blue",
      main= "Distribution of Wilcoxon-Statist.(n=6,m=4)")
abline(h=0:1, col="gray20",lty=2)
layout(1)# set back

N <- 200
hist(U <- rwilcox(N, m=4,n=6), breaks=0:25 - 1/2, border="red", col="pink",
      sub = paste("N =",N))
mtext("N * f(x), f() = true \"density\"", side=3, col="blue")
lines(x, N*fx, type='h', col='blue', lwd=2)
points(x, N*fx, cex=2)

## Better is a Quantile-Quantile Plot
qqplot(U, qw <- qwilcox((1:N - 1/2)/N, m=4,n=6),
        main = paste("Q-Q-Plot of empirical and theoretical quantiles",
                     "Wilcoxon Statistic, (m=4, n=6)",sep="\n"))
n <- as.numeric(names(print(tU <- table(U))))
text(n+.2, n+.5, labels=tU, col="red")
```

---

window	<i>Time Windows</i>
--------	---------------------

---

### Description

`window` is a generic function which extracts the subset of the object `x` observed between the times `start` and `end`. If a frequency is specified, the series is then re-sampled at the new frequency.

### Usage

```

window(x, ...)
window(x, start, end, frequency, deltat, ...) <- value

## S3 method for class 'ts':
window(x, ...)

## Default S3 method:
window(x, start = NULL, end = NULL,
       frequency = NULL, deltat = NULL, extend = FALSE, ...)
```

### Arguments

<code>x</code>	a time-series (or other object if not replacing values).
<code>start</code>	the start time of the period of interest.
<code>end</code>	the end time of the period of interest.
<code>frequency, deltat</code>	the new frequency can be specified by either (or both if they are consistent).
<code>extend</code>	logical. If true, the <code>start</code> and <code>end</code> values are allowed to extend the series. If false, attempts to extend the series give a warning and are ignored.
<code>...</code>	further arguments passed to or from other methods.
<code>value</code>	replacement values.

### Details

The start and end times can be specified as for `ts`. If there is no observation at the new `start` or `end`, the immediately following (`start`) or preceding (`end`) observation time is used.

The replacement function only has a method for `ts` objects, and is allowed to extend the series, with a warning.

### Value

The value depends on the method. `window.default` will return a vector or matrix with an appropriate `tsp` attribute.

`window.ts` differs from `window.default` only in ensuring the result is a `ts` object.

If `extend = TRUE` the series will be padded with NAs if needed.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[time](#), [ts](#).

**Examples**

```
window(presidents, 1960, c(1969,4)) # values in the 1960's
window(presidents, deltat=1) # All Qtrls
window(presidents, start=c(1945,3), deltat=1) # All Qtr3s
window(presidents, 1944, c(1979,2), extend=TRUE)

pres <- window(presidents, 1945, c(1949,4)) # values in the 1940's
window(pres, 1945.25, 1945.50) <- c(60, 70)
window(pres, 1944, 1944.75) <- 0 # will generate a warning
window(pres, c(1945,4), c(1949,4), freq=1) <- 85:89
pres
```

---

xtabs

*Cross Tabulation*


---

**Description**

Create a contingency table from cross-classifying factors, usually contained in a data frame, using a formula interface.

**Usage**

```
xtabs(formula = ~., data = parent.frame(), subset, na.action,
      exclude = c(NA, NaN), drop.unused.levels = FALSE)
```

**Arguments**

<code>formula</code>	a formula object with the cross-classifying variables, separated by +, on the right hand side. Interactions are not allowed. On the left hand side, one may optionally give a vector or a matrix of counts; in the latter case, the columns are interpreted as corresponding to the levels of a variable. This is useful if the data has already been tabulated, see the examples below.
<code>data</code>	a data frame, list or environment containing the variables to be cross-tabulated.
<code>subset</code>	an optional vector specifying a subset of observations to be used.
<code>na.action</code>	a function which indicates what should happen when the data contain NAs.
<code>exclude</code>	a vector of values to be excluded when forming the set of levels of the classifying factors.

`drop.unused.levels`

a logical indicating whether to drop unused levels in the classifying factors. If this is `FALSE` and there are unused levels, the table will contain zero marginals, and a subsequent chi-squared test for independence of the factors will not work.

### Details

There is a `summary` method for contingency table objects created by `table` or `xtabs`, which gives basic information and performs a chi-squared test for independence of factors (note that the function `chisq.test` in package `ctest` currently only handles 2-d tables).

If a left hand side is given in `formula`, its entries are simply summed over the cells corresponding to the right hand side; this also works if the lhs does not give counts.

### Value

A contingency table in array representation of class `c("xtabs", "table")`, with a `"call"` attribute storing the matched call.

### See Also

`table` for “traditional” cross-tabulation, and `as.data.frame.table` which is the inverse operation of `xtabs` (see the DF example below).

### Examples

```
## 'esoph' has the frequencies of cases and controls for all levels of
## the variables 'agegp', 'alcgp', and 'tobgp'.
xtabs(cbind(ncases, ncontrols) ~ ., data = esoph)
## Output is not really helpful ... flat tables are better:
ftable(xtabs(cbind(ncases, ncontrols) ~ ., data = esoph))
## In particular if we have fewer factors ...
ftable(xtabs(cbind(ncases, ncontrols) ~ agegp, data = esoph))

## This is already a contingency table in array form.
DF <- as.data.frame(UCBAdmissions)
## Now 'DF' is a data frame with a grid of the factors and the counts
## in variable 'Freq'.
DF
## Nice for taking margins ...
xtabs(Freq ~ Gender + Admit, DF)
## And for testing independence ...
summary(xtabs(Freq ~ ., DF))

## Create a nice display for the warp break data.
warpbreaks$replicate <- rep(1:9, len = 54)
ftable(xtabs(breaks ~ wool + tension + replicate, data = warpbreaks))
```

## Chapter 8

# The `tools` package

---

`buildVignettes`      *List and Build Package Vignettes*

---

### Description

Run `Sweave` and `texi2dvi` on all vignettes of a package.

### Usage

```
buildVignettes(package, dir, lib.loc = NULL, quiet = TRUE)
pkgVignettes(package, dir, lib.loc = NULL)
```

### Arguments

<code>package</code>	a character string naming an installed package. If given, Sweave files are searched in subdirectory <code>doc</code> .
<code>dir</code>	a character string specifying the path to a package's root source directory. This subdirectory <code>inst/doc</code> is searched for Sweave files.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. The specified library trees are used to search for <code>package</code> .
<code>quiet</code>	logical. Run <code>Sweave</code> and <code>texi2dvi</code> in quiet mode.

### Value

`buildVignettes` is called for its side effect of creating the PDF versions of all vignettes.  
`pkgVignettes` returns an object of class `"pkgVignettes"`.

---

checkFF	<i>Check Foreign Function Calls</i>
---------	-------------------------------------

---

**Description**

Performs checks on calls to compiled code from R code. Currently only whether the interface functions such as `.C` and `.Fortran` are called with argument `PACKAGE` specified, which is highly recommended to avoid name clashes in foreign function calls.

**Usage**

```
checkFF(package, dir, file, lib.loc = NULL,  
        verbose = getOption("verbose"))
```

**Arguments**

<code>package</code>	a character string naming an installed package. If given, the installed R code of the package is checked.
<code>dir</code>	a character string specifying the path to a package's root source directory. This should contain the subdirectory <code>R</code> (for R code). Only used if <code>package</code> is not given.
<code>file</code>	the name of a file containing R code to be checked. Used if neither <code>package</code> nor <code>dir</code> are given.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. The specified library trees are used to search for <code>package</code> .
<code>verbose</code>	a logical. If <code>TRUE</code> , additional diagnostics are printed (and the result is returned invisibly).

**Value**

An object of class `"checkFF"`, which currently is a list of the (parsed) foreign function calls with no `PACKAGE` argument.

There is a `print` method for nicely displaying the information contained in such objects.

**Warning**

This function is still experimental. Both name and interface might change in future versions.

**See Also**

[.C](#), [.Fortran](#), [Foreign](#).

**Examples**

```
checkFF(package = "stats", verbose = TRUE)
```

---

checkMD5sums            *Check and Create MD5 Checksum Files*

---

**Description**

checkMD5sums checks the files against a file MD5.

**Usage**

```
checkMD5sums(pkg, dir)
```

**Arguments**

pkg	the name of an installed package
dir	the path to the top-level directory of an installed package.

**Details**

The file ‘MD5’ which is created is in a format which can be checked by `md5sum -c MD5` if a suitable command-line version of `md5sum` is available. (One is supplied in the bundle at <http://www.murdoch-sutherland.com/Rtools/tools.zip>.)

If `dir` is missing, an installed package of name `pkg` is searched for.

The private function `tools:::installMD5sums` is used to create MD5 files in the Windows build.

**Value**

checkMD5sums returns a logical, NA if there is no MD5 file to be checked.

**See Also**

[md5sum](#)

---

checkTnF                *Check R Packages or Code for T/F*

---

**Description**

Checks the specified R package or code file for occurrences of T or F, and gathers the expression containing these. This is useful as in R T and F are just variables which are set to the logicals TRUE and FALSE by default, but are not reserved words and hence can be overwritten by the user. Hence, one should always use TRUE and FALSE for the logicals.

**Usage**

```
checkTnF(package, dir, file, lib.loc = NULL)
```

**Arguments**

<code>package</code>	a character string naming an installed package. If given, the installed R code and the examples in the documentation files of the package are checked. R code installed as an image file cannot be checked.
<code>dir</code>	a character string specifying the path to a package's root source directory. This must contain the subdirectory 'R' (for R code), and should also contain 'man' (for documentation). Only used if <code>package</code> is not given. If used, the R code files and the examples in the documentation files are checked.
<code>file</code>	the name of a file containing R code to be checked. Used if neither <code>package</code> nor <code>dir</code> are given.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. The specified library trees are used to search for <code>package</code> .

**Value**

An object of class "checkTnF" which is a list containing, for each file where occurrences of T or F were found, a list with the expressions containing these occurrences. The names of the list are the corresponding file names.

There is a `print` method for nicely displaying the information contained in such objects.

**Warning**

This function is still experimental. Both name and interface might change in future versions.

---

`checkVignettes`      *Check Package Vignettes*

---

**Description**

Check all [Sweave](#) files of a package by running [Sweave](#) and/or [Stangle](#) on them. All R source code files found after the tangling step are [sourced](#) to check whether all code can be executed without errors.

**Usage**

```
checkVignettes(package, dir, lib.loc = NULL, tangle = TRUE,
               weave = TRUE, workdir = c("tmp", "src", "cur"),
               keepfiles = FALSE)
```

**Arguments**

<code>package</code>	a character string naming an installed package. If given, Sweave files are searched in subdirectory <code>doc</code> .
<code>dir</code>	a character string specifying the path to a package's root source directory. This subdirectory <code>inst/doc</code> is searched for Sweave files.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. The specified library trees are used to search for <code>package</code> .
<code>tangle</code>	Perform a tangle and <a href="#">source</a> the extracted code?
<code>weave</code>	Perform a weave?
<code>workdir</code>	Directory used as working directory while checking the vignettes. If <code>"tmp"</code> then a temporary directory is created, this is the default. If <code>"src"</code> then the directory containing the vignettes itself is used, if <code>"cur"</code> then the current working directory of R is used.
<code>keepfiles</code>	Delete file in temporary directory? This option is ignored when <code>workdir!="tmp"</code> .

**Value**

An object of class `"checkVignettes"` which is a list with the error messages found during the tangle and weave steps. There is a `print` method for nicely displaying the information contained in such objects.

---

 codoc

---

*Check Code/Documentation Consistency*


---

**Description**

Find inconsistencies between actual and documented “structure” of R objects in a package. `codoc` compares names and optionally also corresponding positions and default values of the arguments of functions. `codocClasses` and `codocData` compare slot names of S4 classes and variable names of data sets, respectively.

**Usage**

```
codoc(package, dir, lib.loc = NULL,
      use.values = NULL, verbose = getOption("verbose"))
codocClasses(package, lib.loc = NULL)
codocData(package, lib.loc = NULL)
```

## Arguments

<code>package</code>	a character string naming an installed package.
<code>dir</code>	a character string specifying the path to a package's root source directory. This must contain the subdirectories 'man' with R documentation sources (in Rd format) and 'R' with R code. Only used if <code>package</code> is not given.
<code>lib.loc</code>	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for <code>package</code> .
<code>use.values</code>	if FALSE, do not use function default values when comparing code and docs. Otherwise, compare <i>all</i> default values if TRUE, and only the ones documented in the usage otherwise (default).
<code>verbose</code>	a logical. If TRUE, additional diagnostics are printed.

## Details

The purpose of `codoc` is to check whether the documented usage of function objects agrees with their formal arguments as defined in the R code. This is not always straightforward, in particular as the usage information for methods to generic functions often employs the name of the generic rather than the method.

The following algorithm is used. If an installed package is used, it is loaded (unless it is the base package), after possibly detaching an already loaded version of the package. Otherwise, if the sources are used, the R code files of the package are collected and sourced in a new environment. Then, the usage sections of the Rd files are extracted and parsed "as much as possible" to give the formals documented. For interpreted functions in the code environment, the formals are compared between code and documentation according to the values of the argument `use.values`. Synopsis sections are used if present; their occurrence is reported if `verbose` is true.

If a package has a namespace both exported and unexported objects are checked, as well as registered S3 methods. (In the unlikely event of differences the order is exported objects in the package, registered S3 methods and finally objects in the namespace and only the first found is checked.)

Currently, the R documentation format has no high-level markup for the basic "structure" of classes and data sets (similar to the usage sections for function synopses). Variable names for data frames in documentation objects obtained by suitably editing "shells" created by `prompt` are recognized by `codocData` and used provided that the documentation object is for a single data frame (i.e., only has one alias). `codocClasses` analogously handles slot names for classes in documentation objects obtained by editing shells created by `promptClass`.

## Value

`codoc` returns an object of class "codoc". Currently, this is a list which, for each Rd object in the package where an inconsistency was found, contains an element with a list of the mismatches (which in turn are lists with elements `code` and `docs`, giving the corresponding arguments obtained from the function's code and documented usage).

`codocClasses` and `codocData` return objects of class "codocClasses" and "codocData", respectively, with a structure similar to class "codoc".

There are `print` methods for nicely displaying the information contained in such objects.

**Warning**

Both `codocClasses` and `codocData` are still experimental. Names, interfaces and values might change in future versions.

**Note**

The default for `use.values` has been changed from `FALSE` to `NULL`, for R versions 1.9.0 and later.

**See Also**

[undoc](#), [QC](#)

---

`delimMatch`*Delimited Pattern Matching*

---

**Description**

Match delimited substrings in a character vector, with proper nesting.

**Usage**

```
delimMatch(x, delim = c("{", "}"), syntax = "Rd")
```

**Arguments**

<code>x</code>	a character vector.
<code>delim</code>	a character vector of length 2 giving the start and end delimiters. Currently, both must be single characters. Future versions might allow for arbitrary regular expressions.
<code>syntax</code>	currently, always the string <code>"Rd"</code> indicating Rd syntax (i.e., <code>'%</code> starts a comment extending till the end of the line, and <code>'\` escapes). Future versions might know about other syntaxes, perhaps via "syntax tables" allowing to flexibly specify comment, escape, and quote characters.</code>

**Value**

An integer vector of the same length as `x` giving the starting position of the first match, or `-1` if there is none, with attribute `"match.length"` giving the length of the matched text (or `-1` for no match).

**See Also**

[regexpr](#) for "simple" pattern matching.

**Examples**

```
x <- c("\value{foo}", "function(bar)")
delimMatch(x)
delimMatch(x, c("(", ")"))
```

---

fileutils

*File Utilities*


---

**Description**

Utilities for testing and listing files, and manipulating file paths.

**Usage**

```
file_path_as_absolute(x)
file_path_sans_ext(x)
file_test(op, x, y)
list_files_with_exts(dir, exts, all.files = FALSE, full.names = TRUE)
list_files_with_type(dir, type, all.files = FALSE, full.names = TRUE)
```

**Arguments**

<code>x, y</code>	character vectors giving file paths.
<code>op</code>	a character string specifying the test to be performed. Unary tests (only <code>x</code> is used) are <code>"-f"</code> (existence and not being a directory) and <code>"-d"</code> (existence and directory); binary tests are <code>"-nt"</code> (newer than, using the modification dates) and <code>"-ot"</code> .
<code>dir</code>	a character string with the path name to a directory.
<code>exts</code>	a character vector of possible file extensions.
<code>all.files</code>	a logical. If <code>FALSE</code> (default), only visible files are considered; if <code>TRUE</code> , all files are used.
<code>full.names</code>	a logical indicating whether the full paths of the files found are returned (default), or just the file names.
<code>type</code>	a character string giving the “type” of the files to be listed, as characterized by their extensions. Currently, possible values are <code>"code"</code> (R code), <code>"data"</code> (data sets), <code>"demo"</code> (demos), <code>"docs"</code> (R documentation), and <code>"vignette"</code> (vignettes).

**Details**

`file_path_as_absolute` turns a possibly relative file path absolute, performing tilde expansion if necessary. Currently, only a single existing path can be given.

`file_path_sans_ext` returns the file paths without extensions. (Only purely alphanumeric extensions are recognized.)

`file_test` performs shell-style file tests. Note that `file.exists` only tests for existence (test `-e` on some systems) but not for not being a directory.

`list_files_with_exts` returns the paths or names of the files in directory `dir` with extension matching one of the elements of `exts`. Note that by default, full paths are returned, and that only visible files are used.

`list_files_with_type` returns the paths of the files in `dir` of the given “type”, as determined by the extensions recognized by R. When listing R code and documentation files, files in OS-specific subdirectories are included if present. Note that by default, full paths are returned, and that only visible files are used.

### See Also

[file.path](#), [file.info](#), [list.files](#)

### Examples

```
dir <- file.path(R.home(), "library", "stats")
file_test("-d", dir)
file_test("-nt", file.path(dir, "R"), file.path(dir, "demo"))
list_files_with_exts(file.path(dir, "demo"), "R")
list_files_with_type(file.path(dir, "demo"), "demo") # the same
file_path_sans_ext(list.files(file.path(R.home(), "modules")))
```

---

getDepList

*Functions to Retrieve Dependency Information*

---

### Description

Given a dependency matrix, will create a `DependsList` object for that package which will include the dependencies for that matrix, which ones are installed, which unresolved dependencies were found online, which unresolved dependencies were not found online, and any R dependencies.

### Usage

```
getDepList(depMtrx, instPkgs, recursive = TRUE, local = TRUE,
           reduce = TRUE, lib.loc = NULL)
```

```
pkgDepends(pkg, recursive = TRUE, local = TRUE, reduce = TRUE,
           lib.loc = NULL)
```

### Arguments

<code>depMtrx</code>	A dependency matrix as from <code>package.dependencies</code>
<code>pkg</code>	The name of the package
<code>instPkgs</code>	A matrix specifying all packages installed on the local system, as from <code>installed.packages</code>

<code>recursive</code>	Whether or not to include indirect dependencies
<code>local</code>	Whether or not to search only locally
<code>reduce</code>	Whether or not to collapse all sets of dependencies to a minimal value
<code>lib.loc</code>	What libraries to use when looking for installed packages. NULL indicates all library directories in the user's <code>.libPaths()</code> .

## Details

The function `pkgDepends` is a convenience function which wraps `getDepList` and takes as input a package name. It will then query `installed.packages` and also generate a dependency matrix, calling `getDepList` with this information and returning the result.

These functions will retrieve information about the dependencies of the matrix, resulting in a `DependsList` object. This is a list with four elements:

**Depends** A vector of the dependencies for this package.

**Installed** A vector of the dependencies which have been satisfied by the currently installed packages.

**Found** A list representing the dependencies which are not in `Installed` but were found online. This list has element names which are the URLs for the repositories in which packages were found and the elements themselves are vectors of package names which were found in the respective repositories. If `local=TRUE`, the `Found` element will always be empty.

**R** Any R version dependencies.

If `recursive` is `TRUE`, any package that is specified as a dependency will in turn have its dependencies included (and so on), these are known as indirect dependencies. If `recursive` is `FALSE`, only the dependencies directly stated by the package will be used.

If `local` is `TRUE`, the system will only look at the user's local install and not online to find unresolved dependencies.

If `reduce` is `TRUE`, the system will collapse the fields in the `DependsList` object such that a minimal set of dependencies are specified (for instance if there was `('foo', 'foo (>= 1.0.0)', 'foo (>= 1.3.0)')`, it would only return `'foo (>= 1.3.0)'`).

## Value

An object of class `DependsList`

## Author(s)

Jeff Gentry

## See Also

[installFoundDepends](#)

## Examples

```
pkgDepends("tools", local = FALSE)
```

---

`installFoundDepends`*A function to install unresolved dependencies*

---

### Description

This function will take the `Found` element of a `pkgDependsList` object and attempt to install all of the listed packages from the specified repositories.

### Usage

```
installFoundDepends(depPkgList, ...)
```

### Arguments

`depPkgList` A `Found` element from a `pkgDependsList` object  
`...` Arguments to pass on to [install.packages](#)

### Details

This function takes as input the `Found` list from a `pkgDependsList` object. This list will have element names being URLs corresponding to repositories and the elements will be vectors of package names. For each element, [install.packages](#) is called for that URL to install all packages listed in the vector.

### Author(s)

Jeff Gentry

### See Also

[pkgDepends](#), [install.packages](#)

### Examples

```
## Set up a temporary directory to install packages to
tmp <- tempfile()
dir.create(tmp)

pDL <- pkgDepends("tools", local=FALSE)
installFoundDepends(pDL$Found, destdir=tmp)
```

---

makeLazyLoading     *Lazy Loading of Packages*

---

### Description

Tools for Lazy Loading of Packages from a Database.

### Usage

```
makeLazyLoading(package, lib.loc = NULL, compress = TRUE,
                keep.source = getOption("keep.source.pkgs"))
```

### Arguments

package	package name string
lib.loc	library trees, as in library
keep.source	logical; should sources be kept when saving from source
compress	logical; whether to compress entries on the database.

### Details

A tool to set up packages for lazy loading from a database. For packages other than base you can use `makeLazyLoading(package)` to convert them to use lazy loading.

### Author(s)

Luke Tierney and Brian Ripley

### Examples

```
# set up package "splines" for lazy loading -- already done
## Not run:
  tools::makeLazyLoading("splines")

## End(Not run)
```

---

`md5sum`*Compute MD5 Checksums*

---

**Description**

Compute the 32-byte MD5 checksums of one or more files.

**Usage**

```
md5sum(files)
```

**Arguments**

`files` character. The paths of file(s) to be check-summed.

**Value**

A character vector of the same length as `files`, with names equal to `files`. The elements will be `NA` for non-existent or unreadable files, otherwise a 32-character string of hexadecimal digits.

On Windows all files are read in binary mode (as the `md5sum` utilities there do): on other OSes the files are read in the default way.

**See Also**

[checkMD5sums](#)

**Examples**

```
md5sum(dir(R.home(), pattern="^COPY", full.names=TRUE))
```

---

`package.dependencies`*Check Package Dependencies*

---

**Description**

Parses and checks the dependencies of a package against the currently installed version of R [and other packages].

**Usage**

```
package.dependencies(x, check = FALSE,  
                    depLevel = c("Depends", "Imports", "Suggests"))
```

**Arguments**

<code>x</code>	A matrix of package descriptions as returned by <a href="#">CRAN.packages</a> .
<code>check</code>	If TRUE, return logical vector of check results. If FALSE, return parsed list of dependencies.
<code>depLevel</code>	Whether to look for Depends or Suggests level dependencies.

**Details**

Currently we only check if the package conforms with the currently running version of R. In the future we might add checks for inter-package dependencies.

**See Also**

[update.packages](#)

**Description**

Functions for performing various quality checks.

**Usage**

```
checkDocFiles(package, dir, lib.loc = NULL)
checkDocStyle(package, dir, lib.loc = NULL)
checkReplaceFuns(package, dir, lib.loc = NULL)
checkS3methods(package, dir, lib.loc = NULL)
```

**Arguments**

<code>package</code>	a character string naming an installed package.
<code>dir</code>	a character string specifying the path to a package's root source directory. This should contain the subdirectories R (for R code) and 'man' with R documentation sources (in Rd format). Only used if <code>package</code> is not given.
<code>lib.loc</code>	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.

## Details

`checkDocFiles` checks, for all Rd files in a package, whether all arguments shown in the usage sections of the Rd file are documented in its arguments section. It also reports duplicated entries in the arguments section, and “over-documented” arguments which are given in the arguments section but not in the usage. Note that the match is for the usage section and not a possibly existing synopsis section, as the usage is what gets displayed.

`checkDocStyle` investigates how (S3) methods are shown in the usages of the Rd files in a package. It reports the methods shown by their full name rather than using the Rd `\method` markup for indicating S3 methods. Earlier versions of R also reported about methods shown along with their generic, which typically caused problems for the documentation of the primary argument in the generic and its methods. With `\method` now being expanded in a way that class information is preserved, “joint” documentation is no longer necessarily a problem. (The corresponding information is still contained in the object returned by `checkDocStyle`.)

`checkReplaceFuns` checks whether replacement functions or S3/S4 replacement methods in the package R code have their final argument named `value`.

`checkS3methods` checks whether all S3 methods defined in the package R code have all arguments of the corresponding generic, with positional arguments of the generics in the same positions for the method. As an exception, the first argument of a formula method *may* be called `formula` even if this is not the name used by the generic. The rules when `...` is involved are subtle: see the source code. Functions recognized as S3 generics are those with a call to `UseMethod` in their body, internal S3 generics (see [InternalMethods](#)), and S3 group generics (see [Math](#)). Possible dispatch under a different name is not taken into account. The generics are sought first in the given package, then in the **base** package and (currently) the packages **graphics**, **stats**, and **utils** added in R 1.9.0 by splitting the former **base**, and, if an installed package is tested, also in the loaded namespaces/packages listed in the package’s ‘DESCRIPTION’ `Depends` field.

If using an installed package, the checks needing access to all R objects of the package will load the package (unless it is the **base** package), after possibly detaching an already loaded version of the package.

## Value

The functions return objects of class the same as the respective function names containing the information about problems detected. There is a `print` method for nicely displaying the information contained in such objects.

## Warning

These functions are still experimental. Names, interfaces and values might change in future versions.

**Description**

Print a 2-column index table with “names” and titles from given R documentation files to a given output file or connection. The titles are nicely formatted between two column positions (typically 25 and 72, respectively).

**Usage**

```
Rdindex(RdFiles, outFile = "", type = NULL,
        width = 0.9 * getOption("width"), indent = NULL)
```

**Arguments**

RdFiles	a character vector specifying the Rd files to be used for creating the index, either by giving the paths to the files, or the path to a single directory with the sources of a package.
outFile	a connection, or a character string naming the output file to print to. "" (the default) indicates output is to the console.
type	a character string giving the documentation type of the Rd files to be included in the index, or NULL (the default). The type of an Rd file is typically specified via the <code>\docType</code> tag; if type is "data", Rd files whose <i>only</i> keyword is <code>datasets</code> are included as well.
width	a positive integer giving the target column for wrapping lines in the output.
indent	a positive integer specifying the indentation of the second column. Must not be greater than <code>width/2</code> , and defaults to <code>width/3</code> .

**Details**

If a name is not a valid alias, the first alias (or the empty string if there is none) is used instead.

---

Rdutils

*Rd Utilities*


---

**Description**

Utilities for computing on the information in Rd objects.

**Usage**

```
Rd_db(package, dir, lib.loc = NULL)
Rd_parse(file, text = NULL)
```

**Arguments**

<code>package</code>	a character string naming an installed package.
<code>dir</code>	a character string specifying the path to a package's root source directory. This should contain the subdirectory 'man' with R documentation sources (in Rd format). Only used if <code>package</code> is not given.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. The specified library trees are used to search for <code>package</code> .
<code>file</code>	a connection, or a character string giving the name of a file or a URL to read documentation in Rd format from.
<code>text</code>	character vector with documentation in Rd format. Elements are treated as if they were lines of a file.

**Details**

`Rd_db` builds a simple "data base" of all Rd sources in a package, as a list of character vectors with the lines of the Rd files in the package. This is particularly useful for working on installed packages, where the individual Rd files in the sources are no longer available.

`Rd_parse` is a simple top-level Rd parser/analyzer. It returns a list with components

**meta** a list containing the Rd meta data (aliases, concepts, keywords, and documentation type);

**data** a data frame with the names (`tags`) and corresponding text (`vals`) of the top-level sections in the R documentation object;

**rest** top-level text not accounted for (currently, silently discarded by `Rdconv`, and hence usually the indication of a problem).

Note that at least for the time being, only the top-level structure is analyzed.

**Warning**

These functions are still experimental. Names, interfaces and values might change in future versions.

**Examples**

```
## Build the Rd db for the (installed) base package.
db <- Rd_db("base")
## Run Rd_parse on all entries in the Rd db.
db <- lapply(db, function(txt) Rd_parse(text = txt))
## Extract the metadata.
meta <- lapply(db, "[", "meta")

## Keyword metadata per Rd file.
keywords <- lapply(meta, "[", "keywords")
## Tabulate the keyword entries.
kw_table <- sort(table(unlist(keywords)))
## The 5 most frequent ones:
rev(kw_table)[1 : 5]
```

```
## The "most informative" ones:
kw_table[kw_table == 1]

## Concept metadata per Rd file.
concepts <- lapply(meta, "[", "concepts")
## How many files already have \concept metadata?
sum(sapply(concepts, length) > 0)
## How many concept entries altogether?
length(unlist(concepts))
```

---

read.00Index	<i>Read 00Index-style Files</i>
--------------	---------------------------------

---

### Description

Read item/description information from 00Index-style files. Such files are description lists rendered in tabular form, and currently used for the 'INDEX' and 'demo/00Index' files of add-on packages.

### Usage

```
read.00Index(file)
```

### Arguments

file	the name of a file to read data values from. If the specified file is "", then input is taken from the keyboard (in this case input can be terminated by a blank line). Alternatively, file can be a <a href="#">connection</a> , which will be opened if necessary, and if so closed at the end of the function call.
------	--

### Value

a character matrix with 2 columns named "Item" and "Description" which hold the items and descriptions.

### See Also

[formatDL](#) for the inverse operation of creating a 00Index-style file from items and their descriptions.

---

texi2dvi	<i>Compile LaTeX Files</i>
----------	----------------------------

---

**Description**

Run latex and bibtex until all cross-references are resolved and create either a dvi or PDF file.

**Usage**

```
texi2dvi(file, pdf = FALSE, clean = FALSE, quiet = TRUE,
         texi2dvi = getOption("texi2dvi"))
```

**Arguments**

file	character. Name of TeX source file.
pdf	logical. If TRUE, a PDF file is produced instead of the default dvi file (texi2dvi command line option '--pdf').
clean	logical. If TRUE, all auxiliary files are removed (texi2dvi command line option '--clean'). Does not work on some platforms.
quiet	logical. No output unless an error occurs.
texi2dvi	character (or NULL). Script or program used to compile a TeX file to dvi or PDF, respectively. If set to NULL, the 'texi2dvi' script in R's 'bin' directory is used (if it exists), otherwise it is assumed that texi2dvi is in the search path.

**Details**

Some TeX installations on Windows do not have 'texi2dvi.exe'. If 'texify.exe' is present (e.g., part of MikTeX), then it can be used instead: `set options(texi2dvi="texify.exe")` or to the full path of the program.

**Author(s)**

Achim Zeileis

---

tools-deprecated	<i>Deprecated Functions in Package tools</i>
------------------	--

---

**Description**

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as of the next release.

**Usage**

```

filePathAsAbsolute(x)
filePathSansExt(x)
fileTest(op, x, y)
listFilesWithExts(dir, exts, all.files = FALSE, full.names = TRUE)
listFilesWithType(dir, type, all.files = FALSE, full.names = TRUE)

```

**Arguments**

<code>x, y</code>	character vectors giving file paths.
<code>op</code>	a character string specifying the test to be performed. Unary tests (only <code>x</code> is used) are <code>"-f"</code> (existence and not being a directory) and <code>"-d"</code> (existence and directory); binary tests are <code>"-nt"</code> (newer than, using the modification dates) and <code>"-ot"</code> .
<code>dir</code>	a character string with the path name to a directory.
<code>exts</code>	a character vector of possible file extensions.
<code>all.files</code>	a logical. If <code>FALSE</code> (default), only visible files are considered; if <code>TRUE</code> , all files are used.
<code>full.names</code>	a logical indicating whether the full paths of the files found are returned (default), or just the file names.
<code>type</code>	a character string giving the “type” of the files to be listed, as characterized by their extensions. Currently, possible values are <code>"code"</code> (R code), <code>"data"</code> (data sets), <code>"demo"</code> (demos), <code>"docs"</code> (R documentation), and <code>"vignette"</code> (vignettes).

**See Also**

[Deprecated](#)

---

undoc

*Find Undocumented Objects*

---

**Description**

Finds the objects in a package which are undocumented, in the sense that they are visible to the user (or data objects or S4 classes provided by the package), but no documentation entry exists.

**Usage**

```
undoc(package, dir, lib.loc = NULL)
```

**Arguments**

<code>package</code>	a character string naming an installed package.
<code>dir</code>	a character string specifying the path to a package's root source directory. This must contain the subdirectory 'man' with R documentation sources (in Rd format), and at least one of the 'R' or 'data' subdirectories with R code or data objects, respectively.
<code>lib.loc</code>	a character vector of directory names of R libraries, or NULL. The default value of NULL corresponds to all libraries currently known. The specified library trees are used to search for package.

**Details**

This function is useful for package maintainers mostly. In principle, *all* user level R objects should be documented; note however that the precise rules for documenting methods of generic functions are still under discussion.

**Value**

An object of class "undoc" which is a list of character vectors containing the names of the undocumented objects split according to documentation type. This representation is still experimental, and might change in future versions.

There is a `print` method for nicely displaying the information contained in such objects.

**See Also**

[codoc](#), [QC](#)

**Examples**

```
undoc("tools")           # Undocumented objects in 'tools'
```

---

`vignetteDepends`      *Retrieve Dependency Information for a Vignette*

---

**Description**

Given a vignette name, will create a `DependsList` object that reports information about the packages the vignette depends on.

**Usage**

```
vignetteDepends(vignette, recursive = TRUE, reduce = TRUE,
                 local = TRUE, lib.loc = NULL)
```

**Arguments**

<code>vignette</code>	The path to the vignette source
<code>recursive</code>	Whether or not to include indirect dependencies
<code>reduce</code>	Whether or not to collapse all sets of dependencies to a minimal value
<code>local</code>	Whether or not to search only locally
<code>lib.loc</code>	What libraries to search in locally

**Details**

If `recursive` is `TRUE`, any package that is specified as a dependency will in turn have its dependencies included (and so on), these are known as indirect dependencies. If `recursive` is `FALSE`, only the dependencies directly named by the vignette will be used.

If `local` is `TRUE`, the system will only look at the user's local machine and not online to find dependencies.

If `reduce` is `TRUE`, the system will collapse the fields in the `DependsList` to the minimal set of dependencies (for instance if the dependencies were `('foo', 'foo (>= 1.0.0)', 'foo (>= 1.3.0)')`, the return value would be `'foo (>= 1.3.0)'`).

**Value**

An object of class `DependsList`

**Author(s)**

Jeff Gentry

**See Also**

[pkgDepends](#)

**Examples**

```
gridEx <- system.file("doc", "grid.Snw", package = "grid")
vignetteDepends(gridEx)
```

## Chapter 9

# The `utils` package

---

`apropos`

*Find Objects by (Partial) Name*

---

### Description

`apropos` returns a character vector giving the names of all objects in the search list matching `what`.

`find` is a different user interface to the same task as `apropos`.

### Usage

```
apropos(what, where = FALSE, mode = "any")
```

```
find(what, mode = "any", numeric. = FALSE, simple.words = TRUE)
```

### Arguments

`what` name of an object, or [regular expression](#) to match against

`where, numeric.`

a logical indicating whether positions in the search list should also be returned

`mode` character; if not "any", only objects whose `mode` equals `mode` are searched.

`simple.words` logical; if TRUE, the `what` argument is only searched as whole word.

### Details

If `mode != "any"` only those objects which are of mode `mode` are considered. If `where` is TRUE, the positions in the search list are returned as the `names` attribute.

`find` is a different user interface for the same task as `apropos`. However, by default (`simple.words == TRUE`), only full words are searched with `grep(fixed = TRUE)`.

**Author(s)**

Kurt Hornik and Martin Maechler (May 1997).

**See Also**

[objects](#) for listing objects from one place, [help.search](#) for searching the help system, [search](#) for the search path.

**Examples**

```
## Not run: apropos("lm")
apropos(ls)
apropos("lq")

cor <- 1:pi
find(cor)          #> ".GlobalEnv"    "package:stats"
find(cor, num=TRUE) # numbers with these names
find(cor, num=TRUE, mode="function")# only the second one
rm(cor)

## Not run: apropos(".", mode="list") # a long list

# need a DOUBLE backslash '\\' (in case you don't see it anymore)
apropos("\\[")

## Not run: # everything
length(apropos("."))

# those starting with 'pr'
apropos("^pr")

# the 1-letter things
apropos("^[.?$]")
# the 1-2-letter things
apropos("^[.?.?]?")
# the 2-to-4 letter things
apropos("^[.?.?]{2,4}$")

# the 8-and-more letter things
apropos("^[.?.?]{8,}$")
table(nchar(apropos("^[.?.?]{8,}$")))
## End(Not run)
```

**Description**

Run R non-interactively with input from `infile` and send output (stdout/stderr) to another file.

**Usage**

```
R CMD BATCH [options] infile [outfile]
```

**Arguments**

infile	the name of a file with R code to be executed.
options	a list of R command line options, e.g., for setting the amount of memory available and controlling the load/save process. If <code>infile</code> starts with a '-', use '--' as the final option. The default options are '--restore --save --no-readline'.
outfile	the name of a file to which to write output. If not given, the name used is that of <code>infile</code> , with a possible '.R' extension stripped, and '.Rout' appended.

**Details**

Use `R CMD BATCH --help` to be reminded of the usage.

By default, the input commands are printed along with the output. To suppress this behavior, add `options(echo = FALSE)` at the beginning of `infile`.

The `infile` can have end of line marked by LF or CRLF (but not just CR), and files with an incomplete last line (missing end of line (EOL) mark) are processed correctly.

Using `R CMD BATCH` sets the GUI to "none", so none of [x11](#), [jpeg](#) and [png](#) are available.

**Note**

Unlike `Splus BATCH`, this does not run the R process in the background. In most shells, `R CMD BATCH [options] infile [outfile] &` will do so.

Report bugs to [r-bugs@r-project.org](mailto:r-bugs@r-project.org).

---

 browseEnv

*Browse Objects in Environment*


---

**Description**

The `browseEnv` function opens a browser with list of objects currently in `sys.frame()` environment.

**Usage**

```
browseEnv(envir = .GlobalEnv, pattern,
          excludepatt = "^last\\.warning",
          html = .Platform$OS.type != "mac",
          expanded = TRUE, properties = NULL,
          main = NULL, debugMe = FALSE)
```

**Arguments**

envir	an <a href="#">environment</a> the objects of which are to be browsed.
pattern	a <a href="#">regular expression</a> for object subselection is passed to the internal <code>ls()</code> call.
excludepatt	a regular expression for <i>dropping</i> objects with matching names.
html	is used on non Macintosh machines to display the workspace on a HTML page in your favorite browser.
expanded	whether to show one level of recursion. It can be useful to switch it to FALSE if your workspace is large. This option is ignored if <code>html</code> is set to FALSE.
properties	a named list of global properties (of the objects chosen) to be showed in the browser; when NULL (as per default), user, date, and machine information is used.
main	a title string to be used in the browser; when NULL (as per default) a title is constructed.
debugMe	logical switch; if true, some diagnostic output is produced.

**Details**

Very experimental code. Only allows one level of recursion into object structures. The HTML version is not dynamic.

It can be generalized. See sources (`'.../library/base/R/databrowser.R'`) for details.

`wsbrowser()` is currently just an internally used function; its argument list will certainly change.

Most probably, this should rather work through using the 'tkWidget' package (from [www.Bioconductor.org](http://www.Bioconductor.org)).

**See Also**

[str](#), [ls](#).

**Examples**

```
if(interactive()) {
  ## create some interesting objects :
  ofa <- ordered(4:1)
  ex1 <- expression(1+ 0:9)
  ex3 <- expression(u,v, 1+ 0:9)
  example(factor, echo = FALSE)
  example(table, echo = FALSE)
  example(ftable, echo = FALSE)
  example(lm, echo = FALSE)
  example(str, echo = FALSE)

  ## and browse them:
  browseEnv()

  ## a (simple) function's environment:
  af12 <- approxfun(1:2, 1:2, method = "const")
  browseEnv(envir = environment(af12))
}
```

---

browseURL                      *Load URL into a WWW Browser*

---

### Description

Load a given URL into a WWW browser.

### Usage

```
browseURL(url, browser = getOption("browser"))
```

### Arguments

url	a non-empty character string giving the URL to be loaded.
browser	a non-empty character string giving the name of the program to be used as hypertext browser. It should be in the PATH, or a full path specified.

### Details

If `browser` supports remote control and `R` knows how to perform it, the URL is opened in any already running browser or a new one if necessary. This mechanism currently is available for browsers which support the "`-remote openURL(...)`" interface (which includes Netscape 4.x, 6.2.x (but not 6.0/1), 7.1, Opera 5/6, Mozilla  $\geq$  0.9.5 and Mozilla Firefox), Galeon, KDE konqueror (via `kfmclient`) and the GNOME interface to Mozilla. Netscape 7.0 and Opera 7 behave slightly differently, and you will need to open them first. Note that the type of browser is determined from its name, so this mechanism will only be used if the browser is installed under its canonical name.

Because "`-remote`" will use any browser displaying on the X server (whatever machine it is running on), the remote control mechanism is only used if `DISPLAY` points to the local host. This may not allow displaying more than one URL at a time from a remote host.

---

bug.report                      *Send a Bug Report*

---

### Description

Invokes an editor to write a bug report and optionally mail it to the automated r-bugs repository at `<r-bugs@r-project.org>`. Some standard information on the current version and configuration of `R` are included automatically.

### Usage

```
bug.report(subject = "",
           ccaddress = Sys.getenv("USER"),
           method = getOption("mailer"),
           address = "r-bugs@r-project.org",
           file = "R.bug.report")
```

**Arguments**

subject	Subject of the email. Please do not use single quotes (') in the subject! File separate bug reports for multiple bugs
ccaddress	Optional email address for copies (default is current user). Use <code>ccaddress = FALSE</code> for no copies.
method	Submission method, one of "mailx", "gnudoit", "none", or "ess".
address	Recipient's email address.
file	File to use for setting up the email (or storing it when method is "none" or sending mail fails).

**Details**

Currently direct submission of bug reports works only on Unix systems. If the submission method is "mailx", then the default editor is used to write the bug report. Which editor is used can be controlled using [options](#), type `getOption("editor")` to see what editor is currently defined. Please use the help pages of the respective editor for details of usage. After saving the bug report (in the temporary file opened) and exiting the editor the report is mailed using a Unix command line mail utility such as mailx. A copy of the mail is sent to the current user.

If method is "gnudoit", then an emacs mail buffer is opened and used for sending the email.

If method is "none" or NULL (which is the default on Windows systems), then only an editor is opened to help writing the bug report. The report can then be copied to your favorite email program and be sent to the r-bugs list.

If method is "ess" the body of the mail is simply sent to stdout.

**Value**

Nothing useful.

**When is there a bug?**

If R executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

Taking forever to complete a command can be a bug, but you must make certain that it was really R's fault. Some commands simply take a long time. If the input was such that you KNOW it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an R error message in a case where its usual definition ought to be reasonable, it is probably a bug. If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for statistical analysis. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident

that you understand it, and know for certain that what you want is not available. The mailing list `r-devel@r-project.org` is a better place for discussions of this sort than the bug list.

If you are not sure what the command is supposed to do after a careful reading of the manual this indicates a bug in the manual. The manual's job is to make everything clear. It is just as important to report documentation bugs as program bugs.

If the online argument list of a function disagrees with the manual, one of them must be wrong, so report the bug.

### How to report a bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, from when you start R until the problem happens. Always include the version of R, machine, and operating system that you are using; type `version` in R to print this. To help us keep track of which bugs have been fixed and which are still open please send a separate report for each bug.

The most important principle in reporting a bug is to report FACTS, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how R is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that on a data set which you know to be quite large the command `data.frame(x, y, z, monday, tuesday)` never returns. Do not report that `data.frame()` fails for large data sets. Perhaps it fails when a variable name is a day of the week. If this is so then when we got your report we would try out the `data.frame()` command on a large data set, probably with no day of the week variable name, and not see any problem. There is no way in the world that we could guess that we should try a day of the week variable name.

Or perhaps the command fails because the last command you used was a `[]` method that had a bug causing R's internal data structures to be corrupted and making the `data.frame()` command fail from then on. This is why we need to know what other commands you have typed (or read from your startup file).

It is very useful to try and find simple examples that produce apparently the same bug, and somewhat useful to find simple examples that might be expected to produce the bug but actually do not. If you want to debug the problem and find exactly what caused it, that is wonderful. You should still report the facts as well as any explanations or solutions.

Invoking R with the `--vanilla` option may help in isolating a bug. This ensures that the site profile and saved data files are not read.

A bug report can be generated using the `bug.report()` function. This automatically includes the version information and sends the bug to the correct address. Alternatively the bug report can be emailed to `<r-bugs@r-project.org>` or submitted to the Web page at <http://bugs.r-project.org>.

Bug reports on **contributed packages** should be sent to the package maintainer rather than to `r-bugs`.

### Author(s)

This help page is adapted from the Emacs manual and the R FAQ

**See Also**

R FAQ

---

capture.output	<i>Send output to a character string or file</i>
----------------	--

---

**Description**

Evaluates its arguments with the output being returned as a character string or sent to a file. Related to [sink](#) in the same way that [with](#) is related to [attach](#).

**Usage**

```
capture.output(..., file = NULL, append = FALSE)
```

**Arguments**

...	Expressions to be evaluated
file	A file name or a connection, or NULL to return the output as a string. If the connection is not open it will be opened and then closed on exit.
append	Append or overwrite the file?

**Value**

A character string, or NULL if a `file` argument was supplied.

**See Also**[sink](#), [textConnection](#)**Examples**

```
require(stats)
glmout <- capture.output(example(glm))
glmout[1:5]
capture.output(1+1, 2+2)
capture.output({1+1; 2+2})
## Not run:
## on Unix with enscript available
ps <- pipe("enscript -o tempout.ps", "w")
capture.output(example(glm), file=ps)
close(ps)
## End(Not run)
```

**Description**

How to cite R and R packages in publications.

**Usage**

```
citation(package = "base", lib.loc = NULL)
## S3 method for class 'citation':
toBibtex(object, ...)
## S3 method for class 'citationList':
toBibtex(object, ...)
```

**Arguments**

<code>package</code>	a character string with the name of a single package. An error occurs if more than one package name is given.
<code>lib.loc</code>	a character vector with path names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
<code>object</code>	return object of <code>citation</code> .
<code>...</code>	currently not used.

**Details**

The R core development team and the very active community of package authors have invested a lot of time and effort in creating R as it is today. Please give credit where credit is due and cite R and R packages when you use them for data analysis.

Execute function `citation()` for information on how to cite the base R system in publications. If the name of a non-base package is given, the function either returns the information contained in the `CITATION` file of the package or auto-generates citation information. In the latter case the package `'DESCRIPTION'` file is parsed, the resulting citation object may be arbitrarily bad, but is quite useful (at least as a starting point) in most cases.

If only one reference is given, the print method shows both a text version and a BibTeX entry for it, if a package has more than one reference then only the text versions are shown. The BibTeX versions can be obtained using function `toBibtex` (see the examples below).

**Value**

An object of class `"citationList"`.

**See Also**

[citEntry](#)

**Examples**

```
## the basic R reference
citation()

## references for a package -- might not have these installed
if(nchar(system.file(package="lattice"))) citation("lattice")
if(nchar(system.file(package="foreign"))) citation("foreign")

## extract the bibtex entry from the return value
x <- citation()
toBibtex(x)
```

---

citEntry

*Writing Package CITATION Files*


---

**Description**

The ‘CITATION’ file of R packages contains an annotated list of references that should be used for citing the packages.

**Usage**

```
citEntry(entry, textVersion, header = NULL, footer = NULL, ...)
citHeader(...)
citFooter(...)
readCitationFile(file)
```

**Arguments**

entry	a character string with a BibTeX entry type
textVersion	a character string with a text representation of the reference
header	a character string with optional header text
footer	a character string with optional footer text
file	a file name
...	see details below

**Details**

The ‘CITATION’ file of an R package should be placed in the ‘inst’ subdirectory of the package source. The file is an R source file and may contain arbitrary R commands including conditionals and computations. The file is `source()`ed by the R parser in a temporary environment and all resulting objects of class "citation" (the return value of `citEntry`) are collected.

Typically the file will contain zero or more calls to `citHeader`, then one or more calls to `citEntry`, and finally zero or more calls to `citFooter`. `citHeader` and `citFooter` are simply wrappers to `paste`, and their `...` argument is passed on to `paste` as is.

**Value**

`citEntry` returns an object of class "citation", `readCitationFile` returns an object of class "citationList".

**Entry Types**

`citEntry` creates "citation" objects, which are modeled after BibTeX entries. The entry should be a valid BibTeX entry type, e.g.,

**article:** An article from a journal or magazine.

**book:** A book with an explicit publisher.

**inbook:** A part of a book, which may be a chapter (or section or whatever) and/or a range of pages.

**incollection:** A part of a book having its own title.

**inproceedings:** An article in a conference proceedings.

**manual:** Technical documentation like a software manual.

**mastersthesis:** A Master's thesis.

**misc:** Use this type when nothing else fits.

**phdthesis:** A PhD thesis.

**proceedings:** The proceedings of a conference.

**techreport:** A report published by a school or other institution, usually numbered within a series.

**unpublished:** A document having an author and title, but not formally published.

**Entry Fields**

The . . . argument of `citEntry` can be any number of BibTeX fields, including

**address:** The address of the publisher or other type of institution.

**author:** The name(s) of the author(s), either as a character string in the format described in the LaTeX book, or a `personList` object.

**booktitle:** Title of a book, part of which is being cited.

**chapter:** A chapter (or section or whatever) number.

**editor:** Name(s) of editor(s), same format as `author`.

**institution:** The publishing institution of a technical report.

**journal:** A journal name.

**note:** Any additional information that can help the reader. The first word should be capitalized.

**number:** The number of a journal, magazine, technical report, or of a work in a series.

**pages:** One or more page numbers or range of numbers.

**publisher:** The publisher's name.

**school:** The name of the school where a thesis was written.

**series:** The name of a series or set of books.

**title:** The work's title.

**volume:** The volume of a journal or multi-volume book.

**year:** The year of publication.

**Examples**

```
basecit <- system.file("CITATION", package="base")
source(basecit, echo=TRUE)
readCitationFile(basecit)
```

---

close.socket	<i>Close a Socket</i>
--------------	-----------------------

---

**Description**

Closes the socket and frees the space in the file descriptor table. The port may not be freed immediately.

**Usage**

```
close.socket(socket, ...)
```

**Arguments**

socket	A socket object
...	further arguments passed to or from other methods.

**Value**

logical indicating success or failure

**Author(s)**

Thomas Lumley

**See Also**

[make.socket](#), [read.socket](#)

---

compareVersion	<i>Compare Two Package Version Numbers</i>
----------------	--

---

**Description**

Compare two package version numbers to see which is later.

**Usage**

```
compareVersion(a, b)
```

**Arguments**

`a`, `b`            Character strings representing package version numbers.

**Details**

R package version numbers are of the form `x.y-z` for integers `x`, `y` and `z`, with components after `x` optionally missing (in which case the version number is older than those with the components present).

**Value**

0 if the numbers are equal, -1 if `b` is later and 1 if `a` is later (analogous to the C function `strcmp`).

**See Also**

[package\\_version](#), [library](#), [packageStatus](#).

**Examples**

```
compareVersion("1.0", "1.0-1")
compareVersion("7.2-0", "7.1-12")
```

---

 COMPILE

---

*Compile Files for Use with R*


---

**Description**

Compile given source files so that they can subsequently be collected into a shared library using R CMD SHLIB and be loaded into R using `dyn.load()`.

**Usage**

```
R CMD COMPILE [options] srcfiles
```

**Arguments**

`srcfiles`            A list of the names of source files to be compiled. Currently, C, C++ and FORTRAN are supported; the corresponding files should have the extensions `‘.c’`, `‘.cc’` (or `‘.cpp’` or `‘.C’`), and `‘.f’`, respectively.

`options`            A list of compile-relevant settings, such as special values for `CFLAGS` or `FFLAGS`, or for obtaining information about usage and version of the utility.

**Details**

Note that Ratfor is not supported. If you have Ratfor source code, you need to convert it to FORTRAN. On many Solaris systems mixing Ratfor and FORTRAN code will work.

**See Also**

[SHLIB](#), [dyn.load](#)

---

data

*Data Sets*

---

**Description**

Loads specified data sets, or list the available data sets.

**Usage**

```
data(..., list = character(0), package = NULL, lib.loc = NULL,
      verbose = getOption("verbose"), envir = .GlobalEnv)
```

**Arguments**

<code>...</code>	a sequence of names or literal character strings.
<code>list</code>	a character vector.
<code>package</code>	a character vector giving the package(s) to look in for data sets, or <code>NULL</code> . By default, all packages in the search path are used, then the ‘ <code>data</code> ’ subdirectory (if present) of the current working directory.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known.
<code>verbose</code>	a logical. If <code>TRUE</code> , additional diagnostics are printed.
<code>envir</code>	the <a href="#">environment</a> where the data should be loaded.

**Details**

Currently, four formats of data files are supported:

1. files ending ‘`.R`’ or ‘`.r`’ are [source\(\)](#) d in, with the R working directory changed temporarily to the directory containing the respective file.
2. files ending ‘`.RData`’ or ‘`.rda`’ are [load\(\)](#) ed.
3. files ending ‘`.tab`’, ‘`.txt`’ or ‘`.TXT`’ are read using [read.table\(..., header = TRUE\)](#), and hence result in a data frame.
4. files ending ‘`.csv`’ or ‘`.CSV`’ are read using [read.table\(..., header = TRUE, sep = ";"\)](#), and also result in a data frame.

If more than one matching file name is found, the first on this list is used.

The data sets to be loaded can be specified as a sequence of names or character strings, or as the character vector `list`, or as both.

For each given data set, the first two types (‘`.R`’ or ‘`.r`’, and ‘`.RData`’ or ‘`.rda`’ files) can create several variables in the load environment, which might all be named differently from the data set.

The second two (`.tab`, `.txt`, or `.TXT`, and `.csv` or `.CSV` files) will always result in the creation of a single variable with the same name as the data set.

If no data sets are specified, `data` lists the available data sets. It looks for a new-style data index in the `'Meta'` or, if this is not found, an old-style `'OOIndex'` file in the `'data'` directory of each specified package, and uses these files to prepare a listing. If there is a `'data'` area but no index, available data files for loading are computed and included in the listing, and a warning is given: such packages are incomplete. The information about available data sets is returned in an object of class `"packageIQR"`. The structure of this class is experimental. Where the datasets have a different name from the argument that should be used to retrieve them the index will have an entry like `beaver1 (beavers)` which tells us that dataset `beaver1` can be retrieved by the call `data(beaver)`.

If `lib.loc` and `package` are both `NULL` (the default), the data sets are searched for in all the currently loaded packages then in the `'data'` directory (if any) of the current working directory.

If `lib.loc = NULL` but `package` is specified as a character vector, the specified package(s) are searched for first amongst loaded packages and then in the default library/ies (see `.libPaths`).

If `lib.loc` is specified (and not `NULL`), packages are searched for in the specified library/ies, even if they are already loaded from another library.

To just look in the `'data'` directory of the current working directory, set `package = character(0)` (and `lib.loc = NULL`, the default).

### Value

a character vector of all data sets specified, or information about all available data sets in an object of class `"packageIQR"` if none were specified.

### Note

The data files can be many small files. On some file systems it is desirable to save space, and the files in the `'data'` directory of an installed package can be zipped up as a zip archive `'Rdata.zip'`. You will need to provide a single-column file `'filelist'` of file names in that directory.

One can take advantage of the search order and the fact that a `.R` file will change directory. If raw data are stored in `'mydata.txt'` then one can set up `'mydata.R'` to read `'mydata.txt'` and pre-process it, e.g., using `transform`. For instance one can convert numeric vectors to factors with the appropriate labels. Thus, the `.R` file can effectively contain a metadata specification for the plaintext formats.

### See Also

[help](#) for obtaining documentation on data sets, [save](#) for *creating* the second (`.rda`) kind of data, typically the most efficient one.

### Examples

```
require(utils)
data() # list all available data sets
try(data(package = "rpart") )# list the data sets in the rpart package
data(USArrests, "VADeaths") # load the data sets 'USArrests' and 'VADeaths'
help(USArrests) # give information on data set 'USArrests'
```

---

 dataentry

*Spreadsheet Interface for Entering Data*


---

### Description

A spreadsheet-like editor for entering or editing data.

### Usage

```
data.entry(..., Modes = NULL, Names = NULL)
dataentry(data, modes)
de(..., Modes = list(), Names = NULL)
```

### Arguments

<code>...</code>	A list of variables: currently these should be numeric or character vectors or list containing such vectors.
<code>Modes</code>	The modes to be used for the variables.
<code>Names</code>	The names to be used for the variables.
<code>data</code>	A list of numeric and/or character vectors.
<code>modes</code>	A list of length up to that of <code>data</code> giving the modes of (some of) the variables. <code>list()</code> is allowed.

### Details

The data entry editor is only available on some platforms and GUIs. Where available it provides a means to visually edit a matrix or a collection of variables (including a data frame) as described in the “Notes” section.

`data.entry` has side effects, any changes made in the spreadsheet are reflected in the variables. The functions `de`, `de.ncols`, `de.setup` and `de.restore` are designed to help achieve these side effects. If the user passes in a matrix, `X` say, then the matrix is broken into columns before `dataentry` is called. Then on return the columns are collected and glued back together and the result assigned to the variable `X`. If you don’t want this behaviour use `dataentry` directly.

The primitive function is `dataentry`. It takes a list of vectors of possibly different lengths and modes (the second argument) and opens a spreadsheet with these variables being the columns. The columns of the `dataentry` window are returned as vectors in a list when the spreadsheet is closed.

`de.ncols` counts the number of columns which are supplied as arguments to `data.entry`. It attempts to count columns in lists, matrices and vectors. `de.setup` sets things up so that on return the columns can be regrouped and reassigned to the correct name. This is handled by `de.restore`.

### Value

`de` and `dataentry` return the edited value of their arguments. `data.entry` invisibly returns a vector of variable names but its main value is its side effect of assigning new version of those variables in the user’s workspace.

**Note**

The details of interface to the data grid may differ by platform and GUI. The following description applies to the X11-based implementation under Unix.

You can navigate around the grid using the cursor keys or by clicking with the (left) mouse button on any cell. The active cell is highlighted by thickening the surrounding rectangle. Moving to the right or down will scroll the grid as needed: there is no constraint to the rows or columns currently in use.

There are alternative ways to navigate using the keys. Return and (keypad) Enter and LineFeed all move down. Tab moves right and Shift-Tab move left. Home moves to the top left.

PageDown or Control-F moves down a page, and PageUp or Control-B up by a page. End will show the last used column and the last few rows used (in any column).

Using any other key starts an editing process on the currently selected cell: moving away from that cell enters the edited value whereas Esc cancels the edit and restores the previous value. When the editing process starts the cell is cleared. In numerical columns (the default) only letters making up a valid number (including `- . eE`) are accepted, and entering an invalid edited value (such as blank) enters NA in that cell. The last entered value can be deleted using the BackSpace or Del(ete) key. Only a limited number of characters (currently 29) can be entered in a cell, and if necessary only the start or end of the string will be displayed, with the omissions indicated by `>` or `<`. (The start is shown except when editing.)

Entering a value in a cell further down a column than the last used cell extends the variable and fills the gap (if any) by NAs (not shown on screen).

The column names can only be selected by clicking in them. This gives a popup menu to select the column type (currently Real (numeric) or Character) or to change the name. Changing the type converts the current contents of the column (and converting from Character to Real may generate NAs.) If changing the name is selected the header cell becomes editable (and is cleared). As with all cells, the value is entered by moving away from the cell by clicking elsewhere or by any of the keys for moving down (only).

New columns are created by entering values in them (and not by just assigning a new name). The mode of the column is auto-detected from the first value entered: if this is a valid number it gives a numeric column. Unused columns are ignored, so adding data in `var5` to a three-column grid adds one extra variable, not two.

The Copy button copies the currently selected cell: paste copies the last copied value to the current cell, and right-clicking selects a cell *and* copies in the value. Initially the value is blank, and attempts to paste a blank value will have no effect.

Control-L will refresh the display, recalculating field widths to fit the current entries.

In the default mode the column widths are chosen to fit the contents of each column, with a default of 10 characters for empty columns. you can specify fixed column widths by setting option `de.cellwidth` to the required fixed width (in characters). (set it to zero to return to variable widths). The displayed width of any field is limited to 600 pixels (and by the window width).

**See Also**

[vi, edit](#): edit uses `dataentry` to edit data frames.

## Examples

```
# call data entry with variables x and y
## Not run: data.entry(x,y)
```

---

debugger	<i>Post-Mortem Debugging</i>
----------	------------------------------

---

## Description

Functions to dump the evaluation environments (frames) and to examine dumped frames.

## Usage

```
dump.frames(dumpto = "last.dump", to.file = FALSE)
debugger(dump = last.dump)
```

## Arguments

<code>dumpto</code>	a character string. The name of the object or file to dump to.
<code>to.file</code>	logical. Should the dump be to an R object or to a file?
<code>dump</code>	An R dump object created by <code>dump.frames</code> .

## Details

To use post-mortem debugging, set the option `error` to be a call to `dump.frames`. By default this dumps to an R object "last.dump" in the workspace, but it can be set to dump to a file (as dump of the object produced by a call to [save](#)). The dumped object contain the call stack, the active environments and the last error message as returned by [geterrmessage](#).

When dumping to file, `dumpto` gives the name of the dumped object and the file name has `.rda` appended.

A dump object of class "dump.frames" can be examined by calling `debugger`. This will give the error message and a list of environments from which to select repeatedly. When an environment is selected, it is copied and the `browser` called from within the copy.

If `dump.frames` is installed as the error handler, execution will continue even in non-interactive sessions. See the examples for how to dump and then quit.

## Value

None.

## Note

Functions such as [sys.parent](#) and [environment](#) applied to closures will not work correctly inside `debugger`.

Of course post-mortem debugging will not work if R is too damaged to produce and save the dump, for example if it has run out of workspace.

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

**See Also**

[options](#) for setting error options; [recover](#) is an interactive debugger working similarly to debugger but directly after the error occurs.

**Examples**

```
## Not run:
options(error=quote(dump.frames("testdump", TRUE)))

f <- function() {
  g <- function() stop("test dump.frames")
  g()
}
f() # will generate a dump on file "testdump.rda"
options(error=NULL)

## possibly in another R session
load("testdump.rda")
debugger(testdump)
Available environments had calls:
1: f()
2: g()
3: stop("test dump.frames")

Enter an environment number, or 0 to exit
Selection: 1
Browsing in the environment with call:
f()
Called from: debugger.look(ind)
Browse[1]> ls()
[1] "g"
Browse[1]> g
function() stop("test dump.frames")
<environment: 759818>
Browse[1]>
Available environments had calls:
1: f()
2: g()
3: stop("test dump.frames")

Enter an environment number, or 0 to exit
Selection: 0

## A possible setting for non-interactive sessions
options(error=quote({dump.frames(to.file=TRUE); q()}))
## End(Not run)
```

**Description**

demo is a user-friendly interface to running some demonstration R scripts. `demo()` gives the list of available topics.

**Usage**

```
demo(topic, device = getOption("device"),
      package = NULL, lib.loc = NULL,
      character.only = FALSE, verbose = getOption("verbose"))
```

**Arguments**

<code>topic</code>	the topic which should be demonstrated, given as a <a href="#">name</a> or literal character string, or a character string, depending on whether <code>character.only</code> is <code>FALSE</code> (default) or <code>TRUE</code> . If omitted, the list of available topics is displayed.
<code>device</code>	the graphics device to be used.
<code>package</code>	a character vector giving the packages to look into for demos, or <code>NULL</code> . By default, all packages in the search path are used.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
<code>character.only</code>	logical; if <code>TRUE</code> , use <code>topic</code> as character string.
<code>verbose</code>	a logical. If <code>TRUE</code> , additional diagnostics are printed.

**Details**

If no topics are given, `demo` lists the available demos. The corresponding information is returned in an object of class `"packageIQR"`. The structure of this class is experimental. In earlier versions of R, an empty character vector was returned along with listing available demos.

**See Also**

[source](#) which is called by `demo`.

**Examples**

```
demo() # for attached packages

## All available demos:
demo(package = .packages(all.available = TRUE))

demo(lm.glm, package="stats")
```

```
## Not run:
  ch <- "scoping"
  demo(ch, character = TRUE)
## End(Not run)
```

---

download.file      *Download File from the Internet*

---

## Description

This function can be used to download a file from the Internet.

## Usage

```
download.file(url, destfile, method, quiet = FALSE, mode = "w",
              cacheOK = TRUE)
```

## Arguments

url	A character string naming the URL of a resource to be downloaded.
destfile	A character string with the name where the downloaded file is saved. Tilde-expansion is performed.
method	Method to be used for downloading files. Currently download methods "internal", "wget" and "lynx" are available. The default is to choose the first of these which will be "internal". The method can also be set through the option "download.file.method": see <a href="#">options()</a> .
quiet	If TRUE, suppress status messages (if any).
mode	character. The mode with which to write the file. Useful values are "w", "wb" (binary), "a" (append) and "ab". Only used for the "internal" method.
cacheOK	logical. Is a server-side cached value acceptable? Implemented for the "internal" and "wget" methods.

## Details

The function `download.file` can be used to download a single file as described by `url` from the internet and store it in `destfile`. The `url` must start with a scheme such as "http://", "ftp://" or "file://".

`cacheOK = FALSE` is useful for "http://" URLs, and will attempt to get a copy directly from the site rather than from an intermediate cache. (Not all platforms support it.) It is used by [CRAN.packages](#).

The remaining details apply to method "internal" only.

The timeout for many parts of the transfer can be set by the option `timeout` which defaults to 60 seconds.

The level of detail provided during transfer can be set by the `quiet` argument and the `internet.info` option. The details depend on the platform and scheme, but setting

`internet.info` to 0 gives all available details, including all server responses. Using 2 (the default) gives only serious messages, and 3 or more suppresses all messages.

A progress bar tracks the transfer. If the file length is known, an equals represents 2% of the transfer completed: otherwise a dot represents 10Kb.

Method `"wget"` can be used with proxy firewalls which require user/password authentication if proper values are stored in the configuration file for `wget`.

### Setting Proxies

This applies to the internal code only.

Proxies can be specified via environment variables. Setting `"no_proxy"` stops any proxy being tried. Otherwise the setting of `"http_proxy"` or `"ftp_proxy"` (or failing that, the all upper-case version) is consulted and if non-empty used as a proxy site. For FTP transfers, the username and password on the proxy can be specified by `"ftp_proxy_user"` and `"ftp_proxy_password"`. The form of `"http_proxy"` should be `"http://proxy.dom.com/"` or `"http://proxy.dom.com:8080/"` where the port defaults to 80 and the trailing slash may be omitted. For `"ftp_proxy"` use the form `"ftp://proxy.dom.com:3128/"` where the default port is 21. These environment variables must be set before the download code is first used: they cannot be altered later by calling `Sys.putenv`.

Usernames and passwords can be set for HTTP proxy transfers via environment variable `http_proxy_user` in the form `user:passwd`. Alternatively, `"http_proxy"` can be of the form `"http://user:pass@proxy.dom.com:8080/"` for compatibility with `wget`. Only the HTTP/1.0 basic authentication scheme is supported.

### Note

Methods `"wget"` and `"lynx"` are for historical compatibility. They will block all other activity on the R process.

For methods `"wget"` and `"lynx"` a system call is made to the tool given by `method`, and the respective program must be installed on your system and be in the search path for executables.

### See Also

[options](#) to set the `timeout` and `internet.info` options.

[url](#) for a finer-grained way to read data from URLs.

[url.show](#), [CRAN.packages](#), [download.packages](#) for applications

---

edit

*Invoke a Text Editor*

---

### Description

Invoke a text editor on an R object.

## Usage

```
## Default S3 method:
edit(name = NULL, file = "", title = NULL,
      editor = getOption("editor"), ...)

vi(name = NULL, file = "")
emacs(name = NULL, file = "")
pico(name = NULL, file = "")
xemacs(name = NULL, file = "")
xedit(name = NULL, file = "")
```

## Arguments

<code>name</code>	a named object that you want to edit. If <code>name</code> is missing then the file specified by <code>file</code> is opened for editing.
<code>file</code>	a string naming the file to write the edited version to.
<code>title</code>	a display name for the object being edited.
<code>editor</code>	a string naming the text editor you want to use. On Unix the default is set from the environment variables <code>EDITOR</code> or <code>VISUAL</code> if either is set, otherwise <code>vi</code> is used. On Windows it defaults to <code>notepad</code> .
<code>...</code>	further arguments to be passed to or from methods.

## Details

`edit` invokes the text editor specified by `editor` with the object name to be edited. It is a generic function, currently with a default method and one for data frames and matrices.

`data.entry` can be used to edit data, and is used by `edit` to edit matrices and data frames on systems for which `data.entry` is available.

It is important to realize that `edit` does not change the object called `name`. Instead, a copy of `name` is made and it is that copy which is changed. Should you want the changes to apply to the object name you must assign the result of `edit` to `name`. (Try `fix` if you want to make permanent changes to an object.)

In the form `edit(name)`, `edit` deparses `name` into a temporary file and invokes the editor `editor` on this file. Quitting from the editor causes `file` to be parsed and that value returned. Should an error occur in parsing, possibly due to incorrect syntax, no value is returned. Calling `edit()`, with no arguments, will result in the temporary file being reopened for further editing.

Currently only the internal editor in Windows makes use of the `title` option; it displays the given name in the window header.

## Note

The functions `vi`, `emacs`, `pico`, `xemacs`, `xedit` rely on the corresponding editor being available and being on the path. This is system-dependent.

## See Also

[edit.data.frame](#), [data.entry](#), [fix](#).

**Examples**

```
## Not run:
# use xedit on the function mean and assign the changes
mean <- edit(mean, editor = "xedit")

# use vi on mean and write the result to file mean.out
vi(mean, file = "mean.out")
## End(Not run)
```

---

edit.data.frame      *Edit Data Frames and Matrices*


---

**Description**

Use data editor on data frame or matrix contents.

**Usage**

```
## S3 method for class 'data.frame':
edit(name, factor.mode = c("character", "numeric"),
      edit.row.names = any(row.names(name) != 1:nrow(name)), ...)

## S3 method for class 'matrix':
edit(name, edit.row.names = any(row.names(name) != 1:nrow(name)), ...)
```

**Arguments**

name	A data frame or matrix.
factor.mode	How to handle factors (as integers or using character levels) in a data frame.
edit.row.names	logical. Show the row names be displayed as a separate editable column?
...	further arguments passed to or from other methods.

**Details**

At present, this only works on simple data frames containing numeric, logical or character vectors and factors. Factors are represented in the spreadsheet as either numeric vectors (which is more suitable for data entry) or character vectors (better for browsing). After editing, vectors are padded with NA to have the same length and factor attributes are restored. The set of factor levels can not be changed by editing in numeric mode; invalid levels are changed to NA and a warning is issued. If new factor levels are introduced in character mode, they are added at the end of the list of levels in the order in which they encountered.

It is possible to use the data-editor's facilities to select the mode of columns to swap between numerical and factor columns in a data frame. Changing any column in a numerical matrix to character will cause the result to be coerced to a character matrix. Changing the mode of logical columns is not supported.

**Value**

The edited data frame.

**Note**

`fix(dataframe)` works for in-place editing by calling this function.

If the data editor is not available, a dump of the object is presented for editing using the default method of `edit`.

At present the data editor is limited to 65535 rows.

**Author(s)**

Peter Dalgaard

**See Also**

[data.entry](#), [edit](#)

**Examples**

```
## Not run:
edit(InsectSprays)
edit(InsectSprays, factor.mode="numeric")
## End(Not run)
```

---

example

*Run an Examples Section from the Online Help*

---

**Description**

Run all the R code from the **Examples** part of R's online help topic `topic` with two possible exceptions, `dontrun` and `dontshow`, see **Details** below.

**Usage**

```
example(topic, package = NULL, lib.loc = NULL,
        local = FALSE, echo = TRUE, verbose = getOption("verbose"),
        setRNG = FALSE,
        prompt.echo = paste(abbreviate(topic, 6), "> ", sep=""))
```

**Arguments**

<code>topic</code>	name or literal character string: the online <a href="#">help</a> topic the examples of which should be run.
<code>package</code>	a character vector giving the package names to look into for example code, or <code>NULL</code> . By default, all packages in the search path are used.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
<code>local</code>	logical: if <code>TRUE</code> evaluate locally, if <code>FALSE</code> evaluate in the workspace.
<code>echo</code>	logical; if <code>TRUE</code> , show the R input when sourcing.
<code>verbose</code>	logical; if <code>TRUE</code> , show even more when running example code.
<code>setRNG</code>	logical or expression; if not <code>FALSE</code> , the random number generator state is saved, then initialized to a specified state, the example is run and the (saved) state is restored. <code>setRNG = TRUE</code> sets the same state as R CMD <code>check</code> does for running a package's examples. This is currently equivalent to <code>setRNG = {RNGkind("default", "default"); set.seed(1)}</code> .
<code>prompt.echo</code>	character; gives the prompt to be used if <code>echo = TRUE</code> .

**Details**

If `lib.loc` is not specified, the packages are searched for amongst those already loaded, then in the specified libraries. If `lib.loc` is specified, they are searched for only in the specified libraries, even if they are already loaded from another library.

An attempt is made to load the package before running the examples, but this will not replace a package loaded from another location.

If `local=TRUE` objects are not created in the workspace and so not available for examination after `example` completes: on the other hand they cannot clobber objects of the same name in the workspace.

As detailed in the manual *Writing R Extensions*, the author of the help page can markup parts of the examples for two exception rules

**dontrun** encloses code that should not be run.

**dontshow** encloses code that is invisible on help pages, but will be run both by the package checking tools, and the `example()` function. This was previously `testonly`, and that form is still accepted.

**Value**

(the value of the last evaluated expression).

**Note**

The examples can be many small files. On some file systems it is desirable to save space, and the files in the 'R-ex' directory of an installed package can be zipped up as a zip archive 'Rex.zip'.

**Author(s)**

Martin Maechler and others

**See Also**

[demo](#)

**Examples**

```
example(InsectSprays)
## force use of the standard package 'stats':
example("smooth", package="stats", lib.loc=.Library)

## set RNG *before* example as when R CMD check is run:

r1 <- example(quantile, setRNG = TRUE)
x1 <- rnorm(1)
u <- runif(1)
## identical random numbers
r2 <- example(quantile, setRNG = TRUE)
x2 <- rnorm(1)
stopifnot(identical(r1, r2))
## but x1 and x2 differ since the RNG state from before example()
## differs and is restored!
x1; x2
```

---

file.edit

*Edit One or More Files*

---

**Description**

Edit one or more files in a text editor.

**Usage**

```
file.edit(..., title = file, editor = getOption("editor"))
```

**Arguments**

...	one or more character vectors containing the names of the files to be edited.
title	the title to use in the editor; defaults to the filename.
editor	the text editor to be used.

**Details**

The behaviour of this function is very system dependent. Currently files can be opened only one at a time on Unix; on Windows, the internal editor allows multiple files to be opened, but has a limit of 50 simultaneous edit windows.

The `title` argument is used for the window caption in Windows, and is ignored on other platforms.

**See Also**

[files](#), [file.show](#), [edit](#), [fix](#),

**Examples**

```
## Not run:  
# open two R scripts for editing  
file.edit("script1.R", "script2.R")  
## End(Not run)
```

---

fix

*Fix an Object*

---

**Description**

fix invokes [edit](#) on x and then assigns the new (edited) version of x in the user's workspace.

**Usage**

```
fix(x, ...)
```

**Arguments**

x                   the name of an R object, as a name or a character string.  
...                  arguments to pass to editor: see [edit](#).

**Details**

The name supplied as x need not exist as an R object, when a function with no arguments and an empty body is supplied for editing.

**See Also**

[edit](#), [edit.data.frame](#)

**Examples**

```
## Not run:  
## Assume 'my.fun' is a user defined function :  
fix(my.fun)  
## now my.fun is changed  
## Also,  
fix(my.data.frame) # calls up data editor  
fix(my.data.frame, factor.mode="char") # use of ...  
## End(Not run)
```

---

`getAnywhere`*Retrieve an R Object, Including from a Namespace*

---

**Description**

This functions locates all objects with name matching its argument, whether visible on the search path, registered as an S3 method or in a namespace but not exported.

**Usage**

```
getAnywhere(x)
```

**Arguments**

`x` a character string or name.

**Details**

The function looks at all loaded namespaces, whether or not they are associated with a package on the search list.

Where functions are found as an S3 method, an attempt is made to find which namespace registered them. This may not be correct, especially if a namespace is unloaded.

**Value**

An object of class "getAnywhere". This is a list with components

<code>name</code>	the name searched for.
<code>objs</code>	a list of objects found
<code>where</code>	a character vector explaining where the object(s) were found
<code>visible</code>	logical: is the object visible
<code>dups</code>	logical: is the object identical to one earlier in the list.

Normally the structure will be hidden by the `print` method. There is a `[` method to extract one or more of the objects found.

**See Also**

[get](#), [getFromNamespace](#)

**Examples**

```
getAnywhere("format.dist")
getAnywhere("simpleLoess") # not exported from stats
```

---

getFromNamespace     *Utility functions for Developing Namespaces*

---

### Description

Utility functions to access and replace the non-exported functions in a namespace, for use in developing packages with namespaces.

### Usage

```
getFromNamespace(x, ns, pos = -1, envir = as.environment(pos))
assignInNamespace(x, value, ns, pos = -1, envir = as.environment(pos))
fixInNamespace(x, ns, pos = -1, envir = as.environment(pos), ...)
```

### Arguments

<code>x</code>	an object name (given as a character string).
<code>value</code>	an R object.
<code>ns</code>	a namespace, or character string giving the namespace.
<code>pos</code>	where to look for the object: see <a href="#">get</a> .
<code>envir</code>	an alternative way to specify an environment to look in.
<code>...</code>	arguments to pass to the editor: see <a href="#">edit</a> .

### Details

The namespace can be specified in several ways. Using, for example, `ns = "stats"` is the most direct, but a loaded package with a namespace can be specified via any of the methods used for [get](#): `ns` can also be the environment printed as `<namespace:foo>`.

`getFromNamespace` is similar to (but predates) the `:::` operator, but is more flexible in how the namespace is specified.

`fixInNamespace` invokes [edit](#) on the object named `x` and assigns the revised object in place of the original object. For compatibility with `fix`, `x` can be unquoted.

### Value

`getFromNamespace` returns the object found (or gives an error).

`assignInNamespace` and `fixInNamespace` are invoked for their side effect of changing the object in the namespace.

### Note

`assignInNamespace` and `fixInNamespace` change the copy in the namespace, but not any copies already exported from the namespace, in particular an object of that name in the package (if already attached) and any copies already imported into other namespaces. They are really intended to be used *only* for objects which are not exported from the namespace. They do attempt to alter a copy registered as an S3 method if one is found.

**See Also**

[get](#), [fix](#), [getS3method](#)

**Examples**

```
getFromNamespace("findGeneric", "utils")
## Not run:
fixInNamespace("predict.ppr", "stats")
stats::predict.ppr
getS3method("predict", "ppr")
## alternatively
fixInNamespace("predict.ppr", pos = 3)
fixInNamespace("predict.ppr", pos = "package:stats")
## End(Not run)
```

---

getS3method

*Get An S3 Method*

---

**Description**

Get a method for an S3 generic, possibly from a namespace.

**Usage**

```
getS3method(f, class, optional = FALSE)
```

**Arguments**

<code>f</code>	character: name of the generic.
<code>class</code>	character: name of the class.
<code>optional</code>	logical: should failure to find the generic or a method be allowed?

**Details**

S3 methods may be hidden in packages with namespaces, and will not then be found by [get](#): this function can retrieve such functions, primarily for debugging purposes.

**Value**

The function found, or NULL if no function is found and `optional = TRUE`.

**See Also**

[methods](#), [get](#)

**Examples**

```
require(stats)
exists("predict.ppr") # false
getS3method("predict", "ppr")
```

---

 head

*Return the First or Last Part of an Object*


---

**Description**

Returns the first or last parts of a vector, matrix, data frame or function.

**Usage**

```

head(x, ...)
## Default S3 method:
head(x, n = 6, ...)
## S3 method for class 'data.frame':
head(x, n = 6, ...)
## S3 method for class 'matrix':
head(x, n = 6, ...)

tail(x, ...)
## Default S3 method:
tail(x, n = 6, ...)
## S3 method for class 'data.frame':
tail(x, n = 6, ...)
## S3 method for class 'matrix':
tail(x, n = 6, addrownums = TRUE, ...)

```

**Arguments**

<code>x</code>	an object
<code>n</code>	size for the resulting object: number of elements for a vector (including lists), rows for a matrix or data frame or lines for a function.
<code>addrownums</code>	if there are no row names, create them from the row numbers.
<code>...</code>	arguments to be passed to or from other methods.

**Details**

For matrices and data frames, the first/last `n` rows are returned. For functions, the first/last `n` lines of the parsed function are returned as character strings.

If a matrix has no row names, then `tail()` will add row names of the form "`[n, ]`" to the result, so that it looks similar to the last lines of `x` when printed. Setting `addrownums = FALSE` suppresses this behaviour.

**Value**

An object (usually) like `x` but generally smaller.

**Author(s)**

Patrick Burns, improved and corrected by R-Core

**Examples**

```
head(freeny.x, n = 10)
head(freeny.y)

tail(freeny.x)
tail(freeny.y)

tail(library)
```

---

help

*Documentation*

---

**Description**

These functions provide access to documentation. Documentation on a topic with name `name` (typically, an R object or a data set) can be printed with either `help(name)` or `?name`.

**Usage**

```
help(topic, offline = FALSE, package = NULL,
      lib.loc = NULL, verbose = getOption("verbose"),
      try.all.packages = getOption("help.try.all.packages"),
      chmhelp = getOption("chmhelp"),
      htmlhelp = getOption("htmlhelp"),
      pager = getOption("pager"))
?topic
type?topic
```

**Arguments**

<code>topic</code>	usually, the name on which documentation is sought. The name may be quoted or unquoted (but note that if <code>topic</code> is the name of a variable containing a character string documentation is provided for the name, not for the character string). The <code>topic</code> argument may also be a function call, to ask for documentation on a corresponding method. See the section on method documentation.
<code>offline</code>	a logical indicating whether documentation should be displayed on-line to the screen (the default) or hardcopy of it should be produced.
<code>package</code>	a name or character vector giving the packages to look into for documentation, or <code>NULL</code> . By default, all packages in the search path are used.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.

<code>verbose</code>	logical; if TRUE, the file name is reported.
<code>try.all.packages</code>	logical; see <code>Notes</code> .
<code>chmhelp</code>	logical (or NULL). Only relevant under Windows. If TRUE the Compiled HTML version of the help will be shown in a help viewer.
<code>htmlhelp</code>	logical (or NULL). If TRUE (which is the default after <code>help.start</code> has been called), the HTML version of the help will be shown in the browser specified by <code>options("browser")</code> . See <code>browseURL</code> for details of the browsers that are supported. Where possible an existing browser window is re-used.
<code>pager</code>	the pager to be used for <code>file.show</code> .
<code>type</code>	the special type of documentation to use for this topic; for example, if the type is <code>class</code> , documentation is provided for the class with name <code>topic</code> . The function <code>topicName</code> returns the actual name used in this case. See the section on method documentation for the uses of <code>type</code> to get help on formal methods.

## Details

In the case of unary and binary operators and control-flow special forms (including `if`, `for` and `function`), the topic may need to be quoted.

If `offline` is TRUE, hardcopy of the documentation is produced by running the LaTeX version of the help page through `latex` (note that LaTeX 2e is needed) and `dvips`. Depending on your `dvips` configuration, hardcopy will be sent to the printer or saved in a file. If the programs are in non-standard locations and hence were not found at compile time, you can either set the options `latexcmd` and `dvipscmd`, or the environment variables `R_LATEXCMD` and `R_DVIPSCMD` appropriately. The appearance of the output can be customized through a file ‘`Rhelp.cfg`’ somewhere in your LaTeX search path.

If LaTeX versions of help pages were not built at the installation of the package, the `print` method will ask if conversion with `R CMD Rdconv` (which requires Perl) should be attempted.

## Method Documentation

The authors of formal (‘S4’) methods can provide documentation on specific methods, as well as overall documentation on the methods of a particular function. The `"?"` operator allows access to this documentation in three ways.

The expression `methods ? f` will look for the overall documentation methods for the function `f`. Currently, this means the documentation file containing the alias `f-methods`.

There are two different ways to look for documentation on a particular method. The first is to supply the `topic` argument in the form of a function call, omitting the `type` argument. The effect is to look for documentation on the method that would be used if this function call were actually evaluated. See the examples below. If the function is not a generic (no S4 methods are defined for it), the help reverts to documentation on the function name.

The `"?"` operator can also be called with `type` supplied as `"method"`; in this case also, the `topic` argument is a function call, but the arguments are now interpreted as specifying the class of the argument, not the actual expression that will appear in a real call to the function. See the examples below.

The first approach will be tedious if the actual call involves complicated expressions, and may be slow if the arguments take a long time to evaluate. The second approach avoids these difficulties, but you do have to know what the classes of the actual arguments will be when they are evaluated.

Both approaches make use of any inherited methods; the signature of the method to be looked up is found by using `selectMethod` (see the documentation for `getMethod`).

### Note

Unless `lib.loc` is specified explicitly, the loaded packages are searched before those in the specified libraries. This ensures that if a library is loaded from a library not in the known library trees, then the help from the loaded library is used. If `lib.loc` is specified explicitly, the loaded packages are *not* searched.

If this search fails and argument `try.all.packages` is `TRUE` and neither `packages` nor `lib.loc` is specified, then all the packages in the known library trees are searched for help on `topic` and a list of (any) packages where help may be found is printed (but no help is shown). **N.B.** searching all packages can be slow.

The help files can be many small files. On some file systems it is desirable to save space, and the text files in the ‘help’ directory of an installed package can be zipped up as a zip archive ‘Rhelp.zip’. Ensure that file ‘AnIndex’ remains un-zipped. Similarly, all the files in the ‘latex’ directory can be zipped to ‘Rhelp.zip’.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### See Also

`help.search()` for finding help pages on a “vague” topic; `help.start()` which opens the HTML version of the R help pages; `library()` for listing available packages and the user-level objects they contain; `data()` for listing available data sets; `methods()`.

See `prompt()` to get a prototype for writing help pages of private packages.

### Examples

```
help()
help(help)           # the same

help(lapply)
?lapply             # the same

help("for")         # or ?"for", but the quotes are needed
?"+"

help(package="splines") # get help even when package is not loaded

data()              # list all available data sets
?women              # information about data set "women"
```

```

topi <- "women"
## Not run: help(topi) ##--> Error: No documentation for 'topi'

try(help("bs", try.all.packages=FALSE)) # reports not found (an error)
help("bs", try.all.packages=TRUE) # reports can be found in package 'splines'

## Not run:
require(methods)
## define a S4 generic function and some methods
combo <- function(x, y) c(x, y)
setGeneric("combo")
setMethod("combo", c("numeric", "numeric"), function(x, y) x+y)

## assume we have written some documentation for combo, and its methods ....

?combo ## produces the function documentation

methods?combo ## looks for the overall methods documentation

method?combo("numeric", "numeric") ## documentation for the method above

?combo(1:10, rnorm(10)) ## ... the same method, selected according to
                        ## the arguments (one integer, the other numeric)

?combo(1:10, letters) ## documentation for the default method
## End(Not run)

```

---

help.search

*Search the Help System*

---

## Description

Allows for searching the help system for documentation matching a given character string in the (file) name, alias, title, concept or keyword entries (or any combination thereof), using either [fuzzy matching](#) or [regular expression matching](#). Names and titles of the matched help entries are displayed nicely.

## Usage

```

help.search(pattern, fields = c("alias", "concept", "title"),
            apropos, keyword, whatis, ignore.case = TRUE,
            package = NULL, lib.loc = NULL,
            help.db = getOption("help.db"),
            verbose = getOption("verbose"),
            rebuild = FALSE, agrep = NULL)

```

## Arguments

`pattern` a character string to be matched in the specified fields. If this is given, the arguments `apropos`, `keyword`, and `whatis` are ignored.

<code>fields</code>	a character vector specifying the fields of the help data bases to be searched. The entries must be abbreviations of "name", "title", "alias", "concept", and "keyword", corresponding to the help page's (file) name, its title, the topics and concepts it provides documentation for, and the keywords it can be classified to.
<code>apropos</code>	a character string to be matched in the help page topics and title.
<code>keyword</code>	a character string to be matched in the help page 'keywords'. 'Keywords' are really categories: the standard categories are listed in file 'RHOME/doc/KEYWORDS' (see also the example) and some package writers have defined their own. If <code>keyword</code> is specified, <code>agrep</code> defaults to <code>FALSE</code> .
<code>what is</code>	a character string to be matched in the help page topics.
<code>ignore.case</code>	a logical. If <code>TRUE</code> , case is ignored during matching; if <code>FALSE</code> , pattern matching is case sensitive.
<code>package</code>	a character vector with the names of packages to search through, or <code>NULL</code> in which case <i>all</i> available packages in the library trees specified by <code>lib.loc</code> are searched.
<code>lib.loc</code>	a character vector describing the location of R library trees to search through, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known.
<code>help.db</code>	a character string giving the file path to a previously built and saved help data base, or <code>NULL</code> .
<code>verbose</code>	logical; if <code>TRUE</code> , the search process is traced.
<code>rebuild</code>	a logical indicating whether the help data base should be rebuilt.
<code>agrep</code>	if <code>NULL</code> (the default unless <code>keyword</code> is used) and the character string to be matched consists of alphanumeric characters, whitespace or a dash only, approximate (fuzzy) matching via <code>agrep</code> is used unless the string has fewer than 5 characters; otherwise, it is taken to contain a <a href="#">regular expression</a> to be matched via <code>grep</code> . If <code>FALSE</code> , approximate matching is not used. Otherwise, one can give a numeric or a list specifying the maximal distance for the approximate match, see argument <code>max.distance</code> in the documentation for <code>agrep</code> .

## Details

Upon installation of a package, a contents data base which contains the information on name, title, aliases and keywords and, concepts starting with R 1.8.0, is computed from the Rd files in the package and serialized as 'Rd.rds' in the 'Meta' subdirectory of the top-level package installation directory (or, prior to R 1.7.0, as 'CONTENTS' in Debian Control Format with aliases and keywords collapsed to character strings in the top-level package installation directory). This, or a pre-built help.search index serialized as 'hsearch.rds' in the 'Meta' directory, is the data base searched by `help.search()`.

The arguments `apropos` and `what is` play a role similar to the Unix commands with the same names.

If possible, the help data base is saved to the file 'help.db' in the '.R' subdirectory of the user's home directory or the current working directory.

Note that currently, the aliases in the matching help files are not displayed.

**Value**

The results are returned in an object of class "hsearch", which has a print method for nicely displaying the results of the query. This mechanism is experimental, and may change in future versions of R.

**See Also**

[help:help.start](#) for starting the hypertext (currently HTML) version of R's online documentation, which offers a similar search mechanism.

[apropos](#) uses regexps and has nice examples.

**Examples**

```
help.search("linear models")      # In case you forgot how to fit linear
                                # models
help.search("non-existent topic")
## Not run:
help.search("print")             # All help pages with topics or title
                                # matching 'print'
help.search(apropos = "print")   # The same

help.search(keyword = "hplot")   # All help pages documenting high-level
                                # plots.
file.show(file.path(R.home(), "doc", "KEYWORDS")) # show all keywords

## Help pages with documented topics starting with 'try'.
help.search("\\btry", fields = "alias")
## Do not use '^' or '$' when matching aliases or keywords
## (unless all packages were installed using R 1.7 or newer).
## End(Not run)
```

---

help.start

*Hypertext Documentation*

---

**Description**

Start the hypertext (currently HTML) version of R's online documentation.

**Usage**

```
help.start(gui = "irrelevant", browser = getOption("browser"),
           remote = NULL)
```

**Arguments**

<code>gui</code>	just for compatibility with S-PLUS.
<code>browser</code>	the name of the program to be used as hypertext browser. It should be in the PATH, or a full path specified.
<code>remote</code>	A character giving a valid URL for the '\$R_HOME' directory on a remote location.

**Details**

All the packages in the known library trees are linked to directory '.R' in the per-session temporary directory. The links are re-made each time `help.start` is run, which should be done after packages are installed, updated or removed.

If the browser given by the `browser` argument is different from the default browser as specified by `options("browser")`, the default is changed to the given browser so that it gets used for all future help requests.

**Note**

There is a Java-based search facility available from the HTML page that `help.start` brings up. Should this not work, please consult the 'R Installation and Administration' manual which is linked from that page.

**See Also**

[help\(\)](#) for on- and off-line help in ASCII/Editor or PostScript format.  
[browseURL](#) for how the help file is displayed.

**Examples**

```
## Not run:  
help.start()  
## End(Not run)
```

---

`index.search`*Search Indices for Help Files*

---

**Description**

Used to search the indices for help files, possibly under aliases.

**Usage**

```
index.search(topic, path, file="AnIndex", type = "help")
```

**Arguments**

<code>topic</code>	The keyword to be searched for in the indices.
<code>path</code>	The path(s) to the packages to be searched.
<code>file</code>	The index file to be searched. Normally "AnIndex".
<code>type</code>	The type of file required.

**Details**

For each package in `path`, examine the file `file` in directory `'type'`, and look up the matching file stem for topic `topic`, if any.

**Value**

A character vector of matching files, as if they are in directory `type` of the corresponding package. In the special cases of `type = "html"`, `"R-ex"` and `"latex"` the file extensions `".html"`, `".R"` and `".tex"` are added.

**See Also**

[help, example](#)

---

INSTALL

*Install Add-on Packages*

---

**Description**

Utility for installing add-on packages.

**Usage**

```
R CMD INSTALL [options] [-l lib] pkgs
```

**Arguments**

<code>pkgs</code>	A list with the path names of the packages to be installed.
<code>lib</code>	the path name of the R library tree to install to.
<code>options</code>	a space-separated list of options through which in particular the process for building the help files can be controlled. Options should only be given once. Use <code>R CMD INSTALL --help</code> for the current list of options.

## Details

If used as `R CMD INSTALL pkgs` without explicitly specifying `lib`, packages are installed into the library tree rooted at the first directory given in the environment variable `R_LIBS` if this is set and non-null, and to the default library tree (which is rooted at `‘$R_HOME/library’`) otherwise.

To install into the library tree `lib`, use `R CMD INSTALL -l lib pkgs`.

Both `lib` and the elements of `pkgs` may be absolute or relative path names. `pkgs` can also contain name of package archive files of the form `‘pkg_version.tar.gz’` as obtained from CRAN, these are then extracted in a temporary directory.

Some package sources contain a `‘configure’` script that can be passed arguments or variables via the option `‘--configure-args’` and `‘--configure-vars’`, respectively, if necessary. The latter is useful in particular if libraries or header files needed for the package are in non-system directories. In this case, one can use the configure variables `LIBS` and `CPPFLAGS` to specify these locations (and set these via `‘--configure-vars’`), see section “Configuration variables” in “R Installation and Administration” for more information. (If these are used more than once on the command line, only the last instance is used.) One can bypass the configure mechanism using the option `‘--no-configure’`.

If `‘--no-docs’` is given, no help files are built. Options `‘--no-text’`, `‘--no-html’`, and `‘--no-latex’` suppress creating the text, HTML, and LaTeX versions, respectively. The default is to build help files in all three versions.

If the option `‘--save’` is used, the installation procedure creates a binary image of the package code, which is then loaded when the package is attached, rather than evaluating the package source at that time. Having a file `‘install.R’` in the package directory makes this the default behavior for the package (option `‘--no-save’` overrides). You may need `‘--save’` if your package requires other packages to evaluate its own source. If the file `‘install.R’` is non-empty, it should contain R expressions to be executed when the package is attached, after loading the saved image. Options to be passed to R when creating the save image can be specified via `‘--save=ARGS’`.

Options `‘--lazy’`, `‘--no-lazy’`, `‘--lazy-data’` and `‘--no-lazy-data’` control where the R objects and the datasets are made available for lazy loading. (These options are overridden by any values set in the DESCRIPTION file.) The default is `‘--no-lazy --no-lazy-data’` except that lazy-loading is used for package with more than 50kB of R code and no saved image.

If the attempt to install the package fails, leftovers are removed. If the package was already installed, the old version is restored.

Use `R CMD INSTALL --help` for more usage information.

## Packages using the methods package

Packages that require the methods package, and that use functions such as `setMethod` or `setClass`, should be installed by creating a binary image.

The presence of a file named `‘install.R’` in the package’s main directory causes an image to be saved. Note that the file is not in the `‘R’` subdirectory: all the code in that subdirectory is used to construct the binary image.

Normally, the file `‘install.R’` will be empty; if it does contain R expressions these will be evaluated when the package is attached, e.g. by a call to the function `library`. (Specifically, the source code evaluated for a package with a saved image consists of a suitable definition of `.First.lib` to ensure loading of the saved image, followed by the R code in file `‘install.R’`, if any.)

**See Also**

[REMOVE](#) and [library](#) for information on using several library trees; [update.packages](#) for automatic update of packages using the internet; the chapter on “Creating R packages” in “Writing R Extensions” (see the ‘doc/manual’ subdirectory of the R source tree).

---

installed.packages *Find Installed Packages*

---

**Description**

Find details of all packages installed in the specified libraries.

**Usage**

```
installed.packages(lib.loc = NULL, priority = NULL)
```

**Arguments**

<code>lib.loc</code>	character vector describing the location of R library trees to search through.
<code>priority</code>	character vector or NULL (default). If non-null, used to select packages; "high" is equivalent to <code>c("base", "recommended")</code> . To select all packages without an assigned priority use <code>priority = "NA"</code> .

**Details**

`installed.packages` scans the ‘DESCRIPTION’ files of each package found along `lib.loc` and returns a matrix of package names, library paths and version numbers.

**Value**

A matrix with one row per package, row names the package names and column names "Package", "LibPath", "Version", "Priority", "Bundle", "Depends" and "Suggests".

**See Also**

[update.packages](#)

**Examples**

```
str(ip <- installed.packages(priority = "high"))
ip[, c(1,3:5)]
```

**Description**

Front-end for creating executable programs.

**Usage**

```
R CMD LINK [options] linkcmd
```

**Arguments**

<code>linkcmd</code>	a list of commands to link together suitable object files (include library objects) to create the executable program.
<code>options</code>	further options to control the linking, or for obtaining information about usage and version.

**Details**

The linker front-end is useful in particular when linking against the R shared library, in which case `linkcmd` must contain `-lR` but need not specify its library path.

Currently only works if the C compiler is used for linking, and no C++ code is used.

Use `R CMD LINK --help` for more usage information.

**Description**

`ls.str` and `lsf.str` are “variations” of `ls` applying `str()` to each matched name, see section ‘Value’.

**Usage**

```
ls.str(pos = 1, pattern, ..., envir = as.environment(pos), mode = "any")
lsf.str(pos = 1, ..., envir = as.environment(pos))
## S3 method for class 'ls_str':
print(x, max.level = 1, give.attr = FALSE, ...)
```

**Arguments**

pos	integer indicating <a href="#">search</a> path position.
pattern	a <a href="#">regular expression</a> passed to <a href="#">ls</a> . Only names matching <code>pattern</code> are considered.
max.level	maximal level of nesting which is applied for displaying nested structures, e.g., a list containing sub lists. Default 0: Display all nesting levels.
give.attr	logical; if TRUE (default), show attributes as sub structures.
envir	environment to use, see <a href="#">ls</a> .
mode	character specifying the <a href="#">mode</a> of objects to consider. Passed to <a href="#">exists</a> and <a href="#">get</a> .
x	an object of class "ls_str".
...	further arguments to pass. and <code>lsf.str</code> passes them to <code>ls.str</code> which passes them on to <code>ls</code> . The (non-exported) print method <code>print.ls_str</code> passes them to <code>str</code> .

**Value**

`ls.str` and `lsf.str` return an object of class "ls\_str", basically the character vector of matching names (functions only for `lsf.str`), similarly to `ls`, with a `print()` method that calls `str()` on each object.

**Author(s)**

Martin Maechler

**See Also**

[str](#), [summary](#), [args](#).

**Examples**

```
lsf.str()#- how do the functions look like which I am using?
ls.str(mode = "list") #- what are the structured objects I have defined?

print(ls.str(), max.level = 1)# don't show details

## which base functions have "file" in their name ?
lsf.str(pos = length(search()), pattern = "file")
```

---

make.packages.html *Update HTML documentation files*

---

### Description

Functions to re-create the HTML documentation files to reflect all installed packages.

### Usage

```
make.packages.html(lib.loc = .libPaths())
```

### Arguments

`lib.loc` character vector. List of libraries to be included.

### Details

This sets up the links from packages in libraries to the ‘.R’ subdirectory of the per-session directory (see [tempdir](#)) and then creates the ‘packages.html’ and ‘index.txt’ files to point to those links.

If a package is available in more than one library tree, all the copies are linked, after the first with suffix `.1` etc.

### Value

Logical, whether the function succeeded in recreating the files.

### See Also

[help.start](#)

---

make.socket *Create a Socket Connection*

---

### Description

With `server = FALSE` attempts to open a client socket to the specified port and host. With `server = TRUE` listens on the specified port for a connection and then returns a server socket. It is a good idea to use [on.exit](#) to ensure that a socket is closed, as you only get 64 of them.

### Usage

```
make.socket(host = "localhost", port, fail = TRUE, server = FALSE)
```

**Arguments**

host	name of remote host
port	port to connect to/listen on
fail	failure to connect is an error?
server	a server socket?

**Value**

An object of class "socket".

socket	socket number. This is for internal use
port	port number of the connection
host	name of remote computer

**Warning**

I don't know if the connecting host name returned when `server = TRUE` can be trusted. I suspect not.

**Author(s)**

Thomas Lumley

**References**

Adapted from Luke Tierney's code for `XLISP-Stat`, in turn based on code from Robbins and Robbins "Practical UNIX Programming"

**See Also**

[close.socket](#), [read.socket](#)

**Examples**

```
daytime <- function(host = "localhost"){
  a <- make.socket(host, 13)
  on.exit(close.socket(a))
  read.socket(a)
}
## Official time (UTC) from US Naval Observatory
## Not run: daytime("tick.usno.navy.mil")
```

---

menu

*Menu Interaction Function*

---

### Description

menu presents the user with a menu of choices labelled from 1 to the number of choices. To exit without choosing an item one can select '0'.

### Usage

```
menu(choices, graphics = FALSE, title = "")
```

### Arguments

choices	a character vector of choices
graphics	a logical indicating whether a graphics menu should be used. Currently unused.
title	a character string to be used as the title of the menu

### Value

The number corresponding to the selected item, or 0 if no choice was made.

### References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

### Examples

```
## Not run:
switch(menu(c("List letters", "List LETTERS"))) + 1,
       cat("Nothing done\n"), letters, LETTERS)
## End(Not run)
```

---

methods

*List Methods for S3 Generic Functions or Classes*

---

### Description

List all available methods for an S3 generic function, or all methods for a class.

### Usage

```
methods(generic.function, class)
```

**Arguments**

`generic.function`  
 a generic function, or a character string naming a generic function.

`class`  
 a symbol or character string naming a class: only used if `generic.function` is not supplied.

**Details**

Function `methods` can be used to find out about the methods for a particular generic function or class. The functions listed are those which *are named like methods* and may not actually be methods (known exceptions are discarded in the code). Note that the listed methods may not be user-visible objects, but often help will be available for them.

If `class` is used, we check that a matching generic can be found for each user-visible object named.

**Value**

An object of class "MethodsFunction", a character vector of function names with an "info" attribute. There is a `print` method which marks with an asterisk any methods which are not visible: such functions can be examined by `getS3method` or `getAnywhere`.

The "info" attribute is a data frame, currently with a logical column, `visible` and a factor column `from` (indicating where the methods were found).

**Note**

This scheme is called *S3* (S version 3). For new projects, it is recommended to use the more flexible and robust *S4* scheme provided in the **methods** package. Functions can have both S3 and S4 methods, and function `showMethods` will list the S4 methods (possibly none).

The original `methods` function was written by Martin Maechler.

**References**

Chambers, J. M. (1992) *Classes and methods: object-oriented programming in S*. Appendix A of *Statistical Models in S* eds J. M. Chambers and T. J. Hastie, Wadsworth & Brooks/Cole.

**See Also**

`S3Methods`, `class`, `getS3method`

**Examples**

```
methods(summary)
methods(class = "aov")
methods("[")      ##- does not list the C-internal ones...
methods("$")      # currently none
methods("$<-")    # replacement function
methods("+")      # binary operator
methods("Math")  # group generic
## Not run:
methods(print)
```

```
## End(Not run)
```

---

nsl	<i>Look up the IP Address by Hostname</i>
-----	---

---

**Description**

Interface to gethostbyname.

**Usage**

```
nsl(hostname)
```

**Arguments**

hostname      the name of the host.

**Value**

The IP address, as a character string, or NULL if the call fails.

**Note**

This was included as a test of internet connectivity, to fail if the node running R is not connected. It will also return NULL if BSD networking is not supported, including the header file 'arpa/inet.h'.

**Examples**

```
## Not run: nsl("www.r-project.org")
```

---

object.size	<i>Report the Space Allocated for an Object</i>
-------------	---

---

**Description**

Provides an estimate of the memory that is being used to store an R object.

**Usage**

```
object.size(x)
```

**Arguments**

x              An R object.

**Details**

Exactly which parts of the memory allocation should be attributed to which object is not clear-cut. This function merely provides a rough indication: it should be reasonably accurate for atomic vectors, but does not detect if elements of a list are shared, for example. (As from R 2.0.0 sharing amongst elements of a character vector is taken into account.)

The calculation is of the size of the object, and excludes the space needed to store its name in the symbol table.

Object sizes are larger on 64-bit platforms than 32-bit ones, but will very likely be the same on different platforms with the same word length.

**Value**

An estimate of the memory allocation attributable to the object, in bytes.

**See Also**

[Memory-limits](#) for the design limitations on object size.

**Examples**

```
object.size(letters)
object.size(ls)
## find the 10 largest objects in base
z <- sapply(ls("package:base"), function(x) object.size(get(x, envir=NULL)))
as.matrix(rev(sort(z))[1:10])
```

---

package.skeleton    *Create a skeleton for a new package*

---

**Description**

package.skeleton automates some of the setup for a new package. It creates directories, saves functions and data to appropriate places, and creates skeleton help files and 'README' files describing further steps in packaging.

**Usage**

```
package.skeleton(name = "anRpackage", list, environment = .GlobalEnv,
                 path = ".", force = FALSE)
```

**Arguments**

name	character string: the directory name for your package.
list	character vector naming the R objects to put in the package
environment	if list is omitted, the contents of this environment are packaged.
path	path to put the package directories in.
force	If FALSE will not overwrite an existing directory.

## Details

This tries to create filenames valid for all OSes known to run R. Invalid characters are replaced by `_`, invalid names are preceded by `zz`, and finally the converted names are made unique by `make.unique(sep = "_")`. This can be done for code and help files but not data files (which are looked for by name).

## Value

used for its side-effects.

## References

Read the *Writing R Extensions* manual for more details

## See Also

`prompt`, `install.packages`

## Examples

```
## Not run:
  f <- function(x,y) x+y
  g <- function(x,y) x-y
  d <- data.frame(a=1,b=2)
  e <- rnorm(1000)
  package.skeleton(list=c("f","g","d","e"), name="AnExample")
## End(Not run)
```

---

packageDescription *Package Description*

---

## Description

Parses and returns the ‘DESCRIPTION’ file of a package.

## Usage

```
packageDescription(pkg, lib.loc = NULL, fields = NULL, drop = TRUE)
```

## Arguments

<code>pkg</code>	a character string with the package name.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known. If the default is used, the loaded packages are searched before the libraries.
<code>fields</code>	a character vector giving the tags of fields to return (if other fields occur in the file they are ignored).

`drop` If TRUE and the length of `fields` is 1, then a single character string with the value of the respective field is returned instead of an object of class "packageDescription".

### Value

If a 'DESCRIPTION' for the given package is found and can successfully be read, `packageDescription` returns an object of class "packageDescription", which is a named list with the values of the (given) fields as elements and the tags as names, unless `drop = TRUE`.

If parsing the 'DESCRIPTION' file was not successful, it returns a named list of NAs with the field tags as names if `fields` is not null, and NA otherwise.

### See Also

[read.dcf](#)

### Examples

```
packageDescription("stats")
packageDescription("stats", fields = c("Package", "Version"))

packageDescription("stats", fields = "Version")
packageDescription("stats", fields = "Version", drop = FALSE)
```

---

packageStatus      *Package Management Tools*

---

### Description

Summarize information about installed packages and packages available at various repositories, and automatically upgrade outdated packages. These tools will replace [update.packages](#) and friends in the future and are currently work in progress.

### Usage

```
packageStatus(lib.loc = NULL,
              repositories = getOption("repositories"))

## S3 method for class 'packageStatus':
summary(object, ...)

## S3 method for class 'packageStatus':
update(object, lib.loc = levels(object$inst$LibPath),
       repositories = levels(object$avail$Repository), ...)

## S3 method for class 'packageStatus':
upgrade(object, ask = TRUE, ...)
```

### Arguments

<code>lib.loc</code>	a character vector describing the location of R library trees to search through, or NULL. The default value of NULL corresponds to all libraries currently known.
<code>repositories</code>	a character vector of URLs describing the location of R package repositories on the Internet or on the local machine.
<code>object</code>	an object of class "packageStatus" as returned by <code>packageStatus</code> .
<code>ask</code>	if TRUE, the user is prompted which packages should be upgraded and which not.
<code>...</code>	currently not used.

### Details

There are `print` and `summary` methods for the "packageStatus" objects: the `print` method gives a brief tabular summary and the `summary` method prints the results.

The `update` method updates the "packageStatus" object. The `upgrade` method is similar to `update.packages`: it offers to install the current versions of those packages which are not currently up-to-date.

### Value

An object of class "packageStatus". This is a list with two components

<code>inst</code>	a data frame with columns "Package", "Version", "Priority", "Bundle", "Depends", "Built", "Status" and "LibPath".
<code>avail</code>	a data frame with columns "Package", "Version", "Priority", "Bundle", "Depends", "Built", "Status" and "Repository" (the URL of the directory from which they can be fetched).

These have similar content to the *matrices* returned by `installed.packages` and `CRAN.packages`.

### Examples

```
## Not run:
x <- packageStatus()
print(x)
summary(x)
upgrade(x)
x <- update(x)
print(x)
## End(Not run)
```

---

page *Invoke a Pager on an R Object*

---

**Description**

Displays a representation of the object named by `x` in a pager.

**Usage**

```
page(x, method = c("dput", "print"), ...)
```

**Arguments**

<code>x</code>	the name of an R object.
<code>method</code>	The default method is to dump the object <i>via</i> <code>dput</code> . An alternative is to print to a file.
<code>...</code>	additional arguments for <code>file.show</code> . Intended for setting <code>pager</code> as <code>title</code> and <code>delete.file</code> are already used.

**See Also**

[file.show](#), [edit](#), [fix](#).

To go to a new page when graphing, see [frame](#).

---

person *Person Names and Contact Information*

---

**Description**

A class and utility methods for holding information about persons like name and email address.

**Usage**

```
person(first = "", last = "", middle = "", email = "")
personList(...)
as.person(x)
as.personList(x)

## S3 method for class 'person':
as.character(x, ...)
## S3 method for class 'personList':
as.character(x, ...)

## S3 method for class 'person':
toBibtex(object, ...)
## S3 method for class 'personList':
toBibtex(object, ...)
```

**Arguments**

first	character string, first name
middle	character string, middle name(s)
last	character string, last name
email	character string, email address
...	for personList an arbitrary number of person objects
x	a character string or an object of class person or personList
object	an object of class person or personList

**Examples**

```
## create a person object directly
p1 <- person("Karl", "Pearson", email = "pearson@stats.heaven")
p1

## convert a string
p2 <- as.person("Ronald Aylmer Fisher")
p2

## create one object holding both
p <- personList(p1, p2)
ps <- as.character(p)
ps
as.personList(ps)

## convert to BibTeX author field
toBibtex(p)
```

---

PkgUtils

*Utilities for Building and Checking Add-on Packages*


---

**Description**

Utilities for checking whether the sources of an R add-on package work correctly, and for building a source or binary package from them.

**Usage**

```
R CMD build [options] pkgdirs
R CMD check [options] pkgdirs
```

**Arguments**

pkgdirs	a list of names of directories with sources of R add-on packages.
options	further options to control the processing, or for obtaining information about usage and version of the utility.

**Details**

R CMD `check` checks R add-on packages from their sources, performing a wide variety of diagnostic checks.

R CMD `build` builds R source or binary packages from their sources. It will create index files in the sources if necessary, so it is often helpful to run `build` before `check`.

Use R CMD `foo --help` to obtain usage information on utility `foo`.

Several of the options to `build --binary` are passed to `INSTALL` so consult its help for the details.

**See Also**

The chapter “Processing Rd format” in “Writing R Extensions” (see the ‘doc/manual’ subdirectory of the R source tree).

`INSTALL` is called by `build --binary`.

---

prompt

*Produce Prototype of an R Documentation File*

---

**Description**

Facilitate the constructing of files documenting R objects.

**Usage**

```
prompt(object, filename = NULL, name = NULL, ...)

## Default S3 method:
prompt(object, filename = NULL, name = NULL,
       force.function = FALSE, ...)

## S3 method for class 'data.frame':
prompt(object, filename = NULL, name = NULL, ...)
```

**Arguments**

<code>object</code>	an R object, typically a function for the default method.
<code>filename</code>	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is <code>name</code> followed by <code>".Rd"</code> . Can also be <code>NA</code> (see below).
<code>name</code>	a character string specifying the name of the object.
<code>force.function</code>	a logical. If <code>TRUE</code> , treat <code>object</code> as function in any case.
<code>...</code>	further arguments passed to or from other methods.

## Details

Unless `filename` is `NA`, a documentation shell for `object` is written to the file specified by `filename`, and a message about this is given. For function objects, this shell contains the proper function and argument names. R documentation files thus created still need to be edited and moved into the ‘`man`’ subdirectory of the package containing the object to be documented.

If `filename` is `NA`, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where `x` is the list-style representation.

When `prompt` is used in `for` loops or scripts, the explicit name specification will be useful.

## Value

If `filename` is `NA`, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

## Warning

The default filename may not be a valid filename under limited file systems (e.g. those on Windows).

Currently, calling `prompt` on a non-function object assumes that the object is in fact a data set and hence documents it as such. This may change in future versions of R. Use `promptData` to create documentation skeletons for data sets.

## Note

The documentation file produced by `prompt.data.frame` does not have the same format as many of the data frame documentation files in the **base** package. We are trying to settle on a preferred format for the documentation.

## Author(s)

Douglas Bates for `prompt.data.frame`

## References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

## See Also

[promptData](#), [help](#) and the chapter on “Writing R documentation” in “Writing R Extensions” (see the ‘`doc/manual`’ subdirectory of the R source tree).

To prompt the user for input, see [readline](#).

## Examples

```
require(graphics)
prompt(plot.default)
prompt(interactive, force.function = TRUE)
unlink("plot.default.Rd")
unlink("interactive.Rd")

prompt(women) # data.frame
unlink("women.Rd")

prompt(sunspots) # non-data.frame data
unlink("sunspots.Rd")
```

---

promptData

*Generate a Shell for Documentation of Data Sets*

---

## Description

Generates a shell of documentation for a data set.

## Usage

```
promptData(object, filename = NULL, name = NULL)
```

## Arguments

object	an R object to be documented as a data set.
filename	usually, a connection or a character string giving the name of the file to which the documentation shell should be written. The default corresponds to a file whose name is name followed by ".Rd". Can also be NA (see below).
name	a character string specifying the name of the object.

## Details

Unless filename is NA, a documentation shell for object is written to the file specified by filename, and a message about this is given.

If filename is NA, a list-style representation of the documentation shell is created and returned. Writing the shell to a file amounts to `cat(unlist(x), file = filename, sep = "\n")`, where x is the list-style representation.

Currently, only data frames are handled explicitly by the code.

## Value

If filename is NA, a list-style representation of the documentation shell. Otherwise, the name of the file written to is returned invisibly.

**Warning**

This function is still experimental. Both interface and value might change in future versions. In particular, it may be preferable to use a character string naming the data set and optionally a specification of where to look for it instead of using `object/name` as we currently do. This would be different from `prompt`, but consistent with other prompt-style functions in package **methods**, and also allow prompting for data set documentation without explicitly having to load the data set.

**See Also**

[prompt](#)

**Examples**

```
promptData(sunspots)
unlink("sunspots.Rd")
```

---

read.fortran	<i>Read fixed-format data</i>
--------------	-------------------------------

---

**Description**

Read fixed-format data files using Fortran-style format specifications.

**Usage**

```
read.fortran(file, format, ..., as.is = TRUE, colClasses = NA)
```

**Arguments**

<code>file</code>	File or connection to read from
<code>format</code>	Character vector or list of vectors. See Details below.
<code>...</code>	Other arguments for <code>read.table</code>
<code>as.is</code>	Keep characters as characters?
<code>colClasses</code>	Variable classes to override defaults. See <a href="#">read.table</a> for details.

**Details**

The format for a field is of one of the following forms: `rFl.d`, `rDl.d`, `rXl`, `rAl`, `rIl`, where `l` is the number of columns, `d` is the number of decimal places, and `r` is the number of repeats. `F` and `D` are numeric formats, `A` is character, `I` is integer, and `X` indicates columns to be skipped. The repeat code `r` and decimal place code `d` are always optional. The length code `l` is required except for `X` formats when `r` is present.

For a single-line record, `format` should be a character vector. For a multiline record it should be a list with a character vector for each line.

Skipped (`X`) columns are not passed to `read.table`, so `colClasses`, `col.names`, and similar arguments passed to `read.table` should not reference these columns.

**Value**

A data frame

**See Also**

[read.fwf](#), [read.csv](#)

**Examples**

```
ff <- tempfile()
cat(file=ff, "123456", "987654", sep="\n")
read.fortran(ff, c("F2.1", "F2.0", "I2"))
read.fortran(ff, c("2F1.0", "2X", "2A1"))
unlink(ff)
cat(file=ff, "123456AB", "987654CD", sep="\n")
read.fortran(ff, list(c("2F3.1", "A2"), c("3I2", "2X")))
unlink(ff)
```

---

`read.fwf`

*Read Fixed Width Format Files*

---

**Description**

Read a “table” of fixed width formatted data into a [data.frame](#).

**Usage**

```
read.fwf(file, widths, header = FALSE, sep = "\t", as.is = FALSE,
         skip = 0, row.names, col.names, n = -1, buffersize = 2000,
         ...)
```

**Arguments**

<code>file</code>	the name of the file which the data are to be read from. Alternatively, <code>file</code> can be a <a href="#">connection</a> , which will be opened if necessary, and if so closed at the end of the function call.
<code>widths</code>	integer vector, giving the widths of the fixed-width fields (of one line), or list of integer vectors giving widths for multiline records
<code>header</code>	a logical value indicating whether the file contains the names of the variables as its first line.
<code>sep</code>	character; the separator used internally; should be a character that does not occur in the file.
<code>as.is</code>	see <a href="#">read.table</a> .
<code>skip</code>	number of initial lines to skip; see <a href="#">read.table</a> .
<code>row.names</code>	see <a href="#">read.table</a> .
<code>col.names</code>	see <a href="#">read.table</a> .

`n` the maximum number of records (lines) to be read, defaulting to no limit.

`bufferize` Maximum number of lines to read at one time

`...` further arguments to be passed to [read.table](#).

### Details

Multiline records are concatenated to a single line before processing. Fields that are of zero-width or are wholly beyond the end of the line in `file` are replaced by NA.

Negative-width fields are used to indicate columns to be skipped, eg `-5` to skip 5 columns. These fields are not seen by `read.table` and so should not be included in a `col.names` or `colClasses` argument.

Reducing the `bufferize` argument may reduce memory use when reading large files with long lines. Increasing `blocksize` may result in faster processing when enough memory is available.

### Value

A [data.frame](#) as produced by `read.table` which is called internally.

### Author(s)

Brian Ripley for R version: original Perl by Kurt Hornik.

### See Also

[scan](#) and [read.table](#).

### Examples

```
ff <- tempfile()
cat(file=ff, "123456", "987654", sep="\n")
read.fwf(ff, width=c(1,2,3)) #> 1 23 456 \ 9 87 654
read.fwf(ff, width=c(1,-2,3)) #> 1 456 \ 9 654
unlink(ff)
cat(file=ff, "123", "987654", sep="\n")
read.fwf(ff, width=c(1,0, 2,3)) #> 1 NA 23 NA \ 9 NA 87 654
unlink(ff)
cat(file=ff, "123456", "987654", sep="\n")
read.fwf(ff, width=list(c(1,0, 2,3), c(2,2,2))) #> 1 NA 23 456 98 76 54
unlink(ff)
```

read.socket                    *Read from or Write to a Socket*

---

### Description

read.socket reads a string from the specified socket, write.socket writes to the specified socket. There is very little error checking done by either.

### Usage

```
read.socket(socket, maxlen = 256, loop = FALSE)
write.socket(socket, string)
```

### Arguments

socket	a socket object
maxlen	maximum length of string to read
loop	wait for ever if there is nothing to read?
string	string to write to socket

### Value

read.socket returns the string read.

### Author(s)

Thomas Lumley

### See Also

[close.socket](#), [make.socket](#)

### Examples

```
finger <- function(user, host = "localhost", port = 79, print = TRUE)
{
  if (!is.character(user))
    stop("user name must be a string")
  user <- paste(user, "\r\n")
  socket <- make.socket(host, port)
  on.exit(close.socket(socket))
  write.socket(socket, user)
  output <- character(0)
  repeat{
    ss <- read.socket(socket)
    if (ss == "") break
    output <- paste(output, ss)
  }
}
```

```
    close.socket(socket)
    if (print) cat(output)
    invisible(output)
  }
## Not run:
finger("root") ## only works if your site provides a finger daemon
## End(Not run)
```

---

recover

*Browsing after an Error*

---

## Description

This function allows the user to browse directly on any of the currently active function calls, and is suitable as an error option. The expression `options(error=recover)` will make this the error option.

## Usage

```
recover()
```

## Details

When called, `recover` prints the list of current calls, and prompts the user to select one of them. The standard R `browser` is then invoked from the corresponding environment; the user can type ordinary S language expressions to be evaluated in that environment.

When finished browsing in this call, type `c` to return to `recover` from the browser. Type another frame number to browse some more, or type `0` to exit `recover`.

The use of `recover` largely supersedes `dump.frames` as an error option, unless you really want to wait to look at the error. If `recover` is called in non-interactive mode, it behaves like `dump.frames`. For computations involving large amounts of data, `recover` has the advantage that it does not need to copy out all the environments in order to browse in them. If you do decide to quit interactive debugging, call `dump.frames` directly while browsing in any frame (see the examples).

**WARNING:** The special `Q` command to go directly from the browser to the prompt level of the evaluator currently interacts with `recover` to effectively turn off the error option for the next error (on subsequent errors, `recover` will be called normally).

## Value

Nothing useful is returned. However, you *can* invoke `recover` directly from a function, rather than through the error option shown in the examples. In this case, execution continues after you type `0` to exit `recover`.

### Compatibility Note

The R `recover` function can be used in the same way as the S-Plus function of the same name; therefore, the error option shown is a compatible way to specify the error action. However, the actual functions are essentially unrelated and interact quite differently with the user. The navigating commands up and down do not exist in the R version; instead, exit the browser and select another frame.

### References

John M. Chambers (1998). *Programming with Data*; Springer.  
See the compatibility note above, however.

### See Also

[browser](#) for details about the interactive computations; [options](#) for setting the error option; [dump.frames](#) to save the current environments for later debugging.

### Examples

```
## Not run:

options(error = recover) # setting the error option

### Example of interaction

> myFit <- lm(y ~ x, data = xy, weights = w)
Error in lm.wfit(x, y, w, offset = offset, ...) :
  missing or negative weights not allowed

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 2
Called from: eval(expr, envir, enclos)
Browse[1]> objects() # all the objects in this frame
[1] "method" "n"      "ny"      "offset"  "tol"    "w"
[7] "x"      "y"
Browse[1]> w
[1] -0.5013844  1.3112515  0.2939348 -0.8983705 -0.1538642
[6] -0.9772989  0.7888790 -0.1919154 -0.3026882
Browse[1]> dump.frames() # save for offline debugging
Browse[1]> c # exit the browser

Enter a frame number, or 0 to exit
1:lm(y ~ x, data = xy, weights = w)
2:lm.wfit(x, y, w, offset = offset, ...)
Selection: 0 # exit recover
>

## End(Not run)
```

---

REMOVE                      *Remove Add-on Packages*

---

**Description**

Utility for removing add-on packages.

**Usage**

```
R CMD REMOVE [options] [-l lib] pkgs
```

**Arguments**

pkgs	a list with the names of the packages to be removed.
lib	the path name of the R library tree to remove from. May be absolute or relative.
options	further options.

**Details**

If used as `R CMD REMOVE pkgs` without explicitly specifying `lib`, packages are removed from the library tree rooted at the first directory given in `$R_LIBS` if this is set and non-null, and to the default library tree (which is rooted at `'$R_HOME/library'`) otherwise.

To remove from the library tree `lib`, use `R CMD REMOVE -l lib pkgs`.

Use `R CMD REMOVE --help` for more usage information.

**See Also**

[INSTALL](#)

---

`remove.packages`            *Remove Installed Packages*

---

**Description**

Removes installed packages and updates index information as necessary.

**Usage**

```
remove.packages(pkgs, lib, version)
```

**Arguments**

pkgs	a character vector with the names of the packages to be removed.
lib	a character string giving the library directory to move the packages from.
version	A character string specifying a version of the package to remove. If none is provided, the system will remove an unversioned install of the package.

**See Also**

[REMOVE](#) for a command line version; [install.packages](#) for installing packages.

---

 RHOME

*R Home Directory*


---

**Description**

Returns the location of the R home directory, which is the root of the installed R tree.

**Usage**

```
R RHOME
```

---

 Rprof

*Enable Profiling of R's Execution*


---

**Description**

Enable or disable profiling of the execution of R expressions.

**Usage**

```
Rprof(filename = "Rprof.out", append = FALSE, interval = 0.02)
```

**Arguments**

filename	The file to be used for recording the profiling results. Set to NULL or "" to disable profiling.
append	logical: should the file be over-written or appended to?
interval	real: time interval between samples.

**Details**

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling works by writing out the call stack every `interval` seconds, to the file specified. Either the [summaryRprof](#) function or the Perl script `R CMD Rprof` can be used to process the output file to produce a summary of the usage; use `R CMD Rprof --help` for usage information.

Note that the timing interval cannot be too small: once the timer goes off, the information is not recorded until the next clock tick (probably every 10msecs). Thus the interval is rounded to the nearest integer number of clock ticks, and is made to be at least one clock tick (at which resolution the total time spent is liable to be underestimated).

**Note**

Profiling is not available on all platforms. By default, it is attempted to compile support for profiling. Configure R with `'--disable-R-profiling'` to change this.

As R profiling uses the same mechanisms as C profiling, the two cannot be used together, so do not use `Rprof` in an executable built for profiling.

**See Also**

The chapter on “Tidying and profiling R code” in “Writing R Extensions” (see the `'doc/manual'` subdirectory of the R source tree).

[summaryRprof](#)

**Examples**

```
## Not run:
Rprof()
## some code to be profiled
Rprof(NULL)
## some code NOT to be profiled
Rprof(append=TRUE)
## some code to be profiled
Rprof(NULL)
...
## Now post-process the output as described in Details
## End(Not run)
```

---

Rtangle

*R Driver for Stangle*


---

**Description**

A driver for [Stangle](#) that extracts R code chunks.

**Usage**

```
Rtangle()
RtangleSetup(file, syntax, output = NULL, annotate = TRUE,
             split = FALSE, prefix = TRUE, quiet = FALSE)
```

**Arguments**

<code>file</code>	Name of Sweave source file.
<code>syntax</code>	An object of class <code>SweaveSyntax</code> .
<code>output</code>	Name of output file, default is to remove extension <code>' .nw'</code> , <code>' .Rnw'</code> or <code>' .Snw'</code> and to add extension <code>' .R'</code> . Any directory names in <code>file</code> are also removed such that the output is created in the current working directory.

<code>annotate</code>	By default, code chunks are separated by comment lines specifying the names and numbers of the code chunks. If <code>FALSE</code> , only the code chunks without any decorating comments are extracted.
<code>split</code>	Split output in single files per code chunk?
<code>prefix</code>	If <code>split = TRUE</code> , prefix the chunk labels by the basename of the input file to get output file names?
<code>quiet</code>	If <code>TRUE</code> all progress messages are suppressed.

**Author(s)**

Friedrich Leisch

**References**

Friedrich Leisch: Sweave User Manual, 2002  
<http://www.ci.tuwien.ac.at/~leisch/Sweave>

**See Also**

[Sweave](#), [RweaveLatex](#)

---

RweaveLatex

*R/LaTeX Driver for Sweave*

---

**Description**

A driver for [Sweave](#) that translates R code chunks in LaTeX files.

**Usage**

```
RweaveLatex()
```

```
RweaveLatexSetup(file, syntax, output = NULL, quiet = FALSE,
                  debug = FALSE, echo = TRUE, eval = TRUE,
                  split = FALSE, stylepath = TRUE,
                  pdf = TRUE, eps = TRUE)
```

**Arguments**

<code>file</code>	Name of Sweave source file.
<code>syntax</code>	An object of class <code>SweaveSyntax</code> .
<code>output</code>	Name of output file, default is to remove extension <code>‘.nw’</code> , <code>‘.Rnw’</code> or <code>‘.Snw’</code> and to add extension <code>‘.tex’</code> . Any directory names in <code>file</code> are also removed such that the output is created in the current working directory.
<code>quiet</code>	If <code>TRUE</code> all progress messages are suppressed.
<code>debug</code>	If <code>TRUE</code> , input and output of all code chunks is copied to the console.

<code>stylepath</code>	If <code>TRUE</code> , a hard path to the file ‘ <code>Sweave.sty</code> ’ installed with this package is set, if <code>FALSE</code> , only <code>\usepackage{Sweave}</code> is written. The hard path makes the TeX file less portable, but avoids the problem of installing the current version of ‘ <code>Sweave.sty</code> ’ to some place in your TeX input path. The argument is ignored if a <code>\usepackage{Sweave}</code> is already present in the Sweave source file.
<code>echo</code>	set default for option <code>echo</code> , see details below.
<code>eval</code>	set default for option <code>eval</code> , see details below.
<code>split</code>	set default for option <code>split</code> , see details below.
<code>pdf</code>	set default for option <code>pdf</code> , see details below.
<code>eps</code>	set default for option <code>eps</code> , see details below.

### Supported Options

RweaveLatex supports the following options for code chunks (the values in parentheses show the default values):

**echo:** logical (`TRUE`). Include S code in the output file?

**eval:** logical (`TRUE`). If `FALSE`, the code chunk is not evaluated, and hence no text or graphical output produced.

**results:** character string (`verbatim`). If `verbatim`, the output of S commands is included in the `verbatim`-like `Soutput` environment. If `tex`, the output is taken to be already proper latex markup and included as is. If `hide` then all output is completely suppressed (but the code executed during the weave).

**print:** logical (`FALSE`) If `TRUE`, each expression in the code chunk is wrapped into a `print()` statement before evaluation, such that the values of all expressions become visible.

**term:** logical (`TRUE`). If `TRUE`, visibility of values emulates an interactive R session: values of assignments are not printed, values of single objects are printed. If `FALSE`, output comes only from explicit `print` or `cat` statements.

**split:** logical (`FALSE`). If `TRUE`, text output is written to separate files for each code chunk.

**strip.white:** logical (`TRUE`). If `TRUE`, blank lines at the beginning and end of output are removed.

**prefix:** logical (`TRUE`). If `TRUE` generated filenames of figures and output have a common prefix.

**prefix.string:** a character string, default is the name of the ‘`.Snw`’ source file.

**include:** logical (`TRUE`), indicating whether input statements for text output and `includegraphics` statements for figures should be auto-generated. Use `include = FALSE` if the output should appear in a different place than the code chunk (by placing the input line manually).

**fig:** logical (`FALSE`), indicating whether the code chunk produces graphical output. Note that only one figure per code chunk can be processed this way.

**eps:** logical (`TRUE`), indicating whether EPS figures shall be generated. Ignored if `fig = FALSE`.

**pdf:** logical (`TRUE`), indicating whether PDF figures shall be generated. Ignored if `fig = FALSE`.

**width:** numeric (6), width of figures in inch.

**height:** numeric (6), height of figures in inch.

**Author(s)**

Friedrich Leisch

**References**

Friedrich Leisch: Sweave User Manual, 2002  
<http://www.ci.tuwien.ac.at/~leisch/Sweave>

**See Also**

[Sweave](#), [Rtangle](#)

---

savehistory

*Load or Save or Display the Commands History*

---

**Description**

Load or save or display the commands history.

**Usage**

```
loadhistory(file = ".Rhistory")  
savehistory(file = ".Rhistory")  
history(max.show = 25, reverse = FALSE)
```

**Arguments**

<code>file</code>	The name of the file in which to save the history, or from which to load it. The path is relative to the current working directory.
<code>max.show</code>	The maximum number of lines to show. <code>Inf</code> will give all of the currently available history.
<code>reverse</code>	logical. If true, the lines are shown in reverse order. Note: this is not useful when there are continuation lines.

**Details**

This works under the `readline` and GNOME interfaces, but not if `readline` is not available (for example, in batch use).

**Note**

If you want to save the history (almost) every session, you can put a call to `savehistory()` in `.Last`.

**Examples**

```
## Not run:
.Last <- function()
  if(interactive()) try(savehistory("~/Rhistory"))
## End(Not run)
```

---

sessionInfo

*Collect Information About the Current R Session*


---

**Description**

Print version information about R and attached packages.

**Usage**

```
sessionInfo(package=NULL)
## S3 method for class 'sessionInfo':
print(x, ...)
## S3 method for class 'sessionInfo':
toLatex(object, ...)
```

**Arguments**

package	a character vector naming installed packages. By default all attached packages are used.
x	an object of class "sessionInfo".
object	an object of class "sessionInfo".
...	currently not used.

**Examples**

```
sessionInfo()
toLatex(sessionInfo())
```

---

SHLIB

*Build Shared Library for Dynamic Loading*


---

**Description**

Compile the given source files and then link all specified object files into a shared library which can be loaded into R using `dyn.load` or `library.dynam`.

**Usage**

```
R CMD SHLIB [options] [-o libname] files
```

**Arguments**

files	a list specifying the object files to be included in the shared library. You can also include the name of source files, for which the object files are automatically made from their sources.
libname	the full name of the shared library to be built, including the extension (typically <code>.so</code> on Unix systems). If not given, the name of the library is taken from the first file.
options	Further options to control the processing. Use <code>R CMD SHLIB --help</code> for a current list.

**Details**

`R CMD SHLIB` is the mechanism used by `INSTALL` to compile source code in packages. Please consult section ‘Creating shared objects’ in the manual ‘Writing R Extensions’ for how to customize it (for example to add `cpp` flags and to add libraries to the link step) and for details of some of its quirks.

**See Also**

[COMPILE](#), [dyn.load](#), [library.dynam](#).

The ‘Writing R Extensions’ manual.

---

str

*Compactly Display the Structure of an Arbitrary R Object*


---

**Description**

Compactly display the internal **structure** of an R object, a “diagnostic” function and an alternative to [summary](#) (and to some extent, [dput](#)). Ideally, only one line for each “basic” structure is displayed. It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists. The idea is to give reasonable output for **any** R object. It calls [args](#) for (non-primitive) function objects.

**Usage**

```
str(object, ...)

## S3 method for class 'data.frame':
str(object, ...)

## Default S3 method:
str(object, max.level = NA, vec.len = 4, digits.d = 3,
      nchar.max = 128, give.attr = TRUE, give.length = TRUE,
      wid = getOption("width"), nest.lev = 0,
      indent.str = paste(rep.int(" ", max(0, nest.lev + 1)), collapse = ".."),
      comp.str="$ ", no.list = FALSE, envir = NULL,
      ...)
```

**Arguments**

<code>object</code>	any R object about which you want to have some information.
<code>max.level</code>	maximal level of nesting which is applied for displaying nested structures, e.g., a list containing sub lists. Default NA: Display all nesting levels.
<code>vec.len</code>	numeric ( $\geq 0$ ) indicating how many “first few” elements are displayed of each vector. The number is multiplied by different factors (from .5 to 3) depending on the kind of vector. Default 4.
<code>digits.d</code>	number of digits for numerical components (as for <code>print</code> ).
<code>nchar.max</code>	maximal number of characters to show for <code>character</code> strings. Longer strings are truncated, see <code>longch</code> example below.
<code>give.attr</code>	logical; if TRUE (default), show attributes as sub structures.
<code>give.length</code>	logical; if TRUE (default), indicate length (as <code>[1: . . . ]</code> ).
<code>wid</code>	the page width to be used. The default is the currently active <code>options("width")</code> .
<code>nest.lev</code>	current nesting level in the recursive calls to <code>str</code> .
<code>indent.str</code>	the indentation string to use.
<code>comp.str</code>	string to be used for separating list components.
<code>no.list</code>	logical; if true, no “list of ..” is nor the class is printed.
<code>envir</code>	the environment to be used for <i>promise</i> (see <code>delay</code> ) objects only.
<code>...</code>	potential further arguments (required for Method/Generic reasons).

**Value**

`str` does not return anything, for efficiency reasons. The obvious side effect is output to the terminal.

**Author(s)**

Martin Maechler `<maechler@stat.math.ethz.ch>` since 1990.

**See Also**

`ls.str` for *listing* objects with their structure; `summary`, `args`.

**Examples**

```
require(stats)
## The following examples show some of 'str' capabilities
str(1:12)
str(ls)
str(args) #- more useful than args(args) !
str(freeny)
str(str)
str(.Machine, digits = 20)
str( lsfit(1:9,1:9))
str( lsfit(1:9,1:9), max = 1)
```

```

op <- options(); str(op) #- save first; otherwise internal options() is used.
need.dev <- !exists(".Device") || is.null(.Device)
if(need.dev) postscript()
str(par()); if(need.dev) graphics.off()

ch <- letters[1:12]; is.na(ch) <- 3:5
str(ch) # character NA's

nchar(longch <- paste(rep(letters,100), collapse=""))
str(longch)
str(longch, nchar.max = 52)

str(quote( { A+B; list(C,D) } ))

## S4 classes :
require(stats4)
x <- 0:10; y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
ll <- function(ymax=15, xh=6) -sum(dpois(y, lambda=ymax/(1+x/xh), log=TRUE))
fit <- mle(ll)
str(fit)

```

---

summaryRprof

*Summarise Output of R Profiler*


---

## Description

Summarise the output of the [Rprof](#) function to show the amount of time used by different R functions.

## Usage

```
summaryRprof(filename = "Rprof.out", chunksize = 5000)
```

## Arguments

filename	Name of a file produced by Rprof()
chunksize	Number of lines to read at a time

## Details

This function is an alternative to R CMD Rprof. It provides the convenience of an all-R implementation but will be slower for large files.

As the profiling output file could be larger than available memory, it is read in blocks of chunksize lines. Increasing chunksize will make the function run faster if sufficient memory is available.

**Value**

A list with components

<code>by.self</code>	Timings sorted by ‘self’ time
<code>by.total</code>	Timings sorted by ‘total’ time
<code>sampling.time</code>	Total length of profiling run

**See Also**

The chapter on “Tidying and profiling R code” in “Writing R Extensions” (see the ‘doc/manual’ subdirectory of the R source tree).

[Rprof](#)

**Examples**

```
## Not run:  
## Rprof() is not available on all platforms  
Rprof(tmp <- tempfile())  
example(glm)  
Rprof()  
summaryRprof(tmp)  
unlink(tmp)  
## End(Not run)
```

---

Sweave

*Automatic Generation of Reports*

---

**Description**

Sweave provides a flexible framework for mixing text and S code for automatic report generation. The basic idea is to replace the S code with its output, such that the final document only contains the text and the output of the statistical analysis.

**Usage**

```
Sweave(file, driver = RweaveLatex(),  
       syntax = getOption("SweaveSyntax"), ...)  
  
Stangle(file, driver = Rtangle(),  
       syntax = getOption("SweaveSyntax"), ...)
```

**Arguments**

<code>file</code>	Name of Sweave source file.
<code>driver</code>	The actual workhorse, see details below.
<code>syntax</code>	An object of class <code>SweaveSyntax</code> or a character string with its name. The default installation provides <code>SweaveSyntaxNoweb</code> and <code>SweaveSyntaxLatex</code> .
<code>...</code>	Further arguments passed to the driver's setup function.

**Details**

Automatic generation of reports by mixing word processing markup (like latex) and S code. The S code gets replaced by its output (text or graphs) in the final markup file. This allows to re-generate a report if the input data change and documents the code to reproduce the analysis in the same file that also produces the report.

Sweave combines the documentation and code chunks together (or their output) into a single document. `Stangle` extracts only the code from the Sweave file creating a valid S source file (that can be run using [source](#)). Code inside `\Sexpr{ }` statements is ignored by `Stangle`.

`Stangle` is just a frontend to Sweave using a simple driver by default, which discards the documentation and concatenates all code chunks the current S engine understands.

**Hook Functions**

Before each code chunk is evaluated, a number of hook functions can be executed. If `getOption("SweaveHooks")` is set, it is taken to be a collection of hook functions. For each logical option of a code chunk (`echo`, `print`, ...) a hook can be specified, which is executed if and only if the respective option is `TRUE`. Hooks must be named elements of the list returned by `getOption("SweaveHooks")` and be functions taking no arguments. E.g., if option "SweaveHooks" is defined as `list(fig = foo)`, and `foo` is a function, then it would be executed before the code in each figure chunk. This is especially useful to set defaults for the graphical parameters in a series of figure chunks.

Note that the user is free to define new Sweave options and associate arbitrary hooks with them. E.g., one could define a hook function for option `clean` that removes all objects in the global environment. Then all code chunks with `clean=TRUE` would start operating on an empty workspace.

**Syntax Definition**

Sweave allows a very flexible syntax framework for marking documentation and text chunks. The default is a noweb-style syntax, as alternative a latex-style syntax can be used. See the user manual for details.

**Author(s)**

Friedrich Leisch

## References

Friedrich Leisch: Sweave User Manual, 2002

<http://www.ci.tuwien.ac.at/~leisch/Sweave>

Friedrich Leisch: Dynamic generation of statistical reports using literate data analysis. In W. Härdle and B. Rönz, editors, *Compstat 2002 - Proceedings in Computational Statistics*, pages 575-580. Physika Verlag, Heidelberg, Germany, 2002. ISBN 3-7908-1517-9.

## See Also

[RweaveLatex](#), [Rtangle](#)

## Examples

```
testfile <- system.file("Sweave", "Sweave-test-1.Rnw", package = "utils")

## create a LaTeX file
Sweave(testfile)

## create an S source file from the code chunks
Stangle(testfile)
## which can be simply sourced
source("Sweave-test-1.R")
```

---

SweaveSyntConv      *Convert Sweave Syntax*

---

## Description

This function converts the syntax of files in [Sweave](#) format to another Sweave syntax definition.

## Usage

```
SweaveSyntConv(file, syntax, output = NULL)
```

## Arguments

<code>file</code>	Name of Sweave source file.
<code>syntax</code>	An object of class <code>SweaveSyntax</code> or a character string with its name giving the target syntax to which the file is converted.
<code>output</code>	Name of output file, default is to remove the extension from the input file and to add the default extension of the target syntax. Any directory names in <code>file</code> are also removed such that the output is created in the current working directory.

## Author(s)

Friedrich Leisch

**References**

Friedrich Leisch: Sweave User Manual, 2002  
<http://www.ci.tuwien.ac.at/~leisch/Sweave>

**See Also**

[RweaveLatex](#), [Rtangle](#)

**Examples**

```
testfile <- system.file("Sweave", "Sweave-test-1.Rnw", package = "utils")

## convert the file to latex syntax
SweaveSyntConv(testfile, SweaveSyntaxLatex)

## and run it through Sweave
Sweave("Sweave-test-1.Stex")
```

---

toLatex

---

*Converting R Objects to BibTeX or LaTeX*


---

**Description**

These methods convert R objects to character vectors with BibTeX or LaTeX markup.

**Usage**

```
toBibtex(object, ...)
toLatex(object, ...)
## S3 method for class 'Bibtex':
print(x, prefix="", ...)
## S3 method for class 'Latex':
print(x, prefix="", ...)
```

**Arguments**

object	object of a class for which a toBibtex or toLatex method exists.
x	object of class "Bibtex" or "Latex".
prefix	a character string which is printed at the beginning of each line, mostly used to insert whitespace for indentation.
...	currently not used in the print methods.

**Details**

Objects of class "Bibtex" or "Latex" are simply character vectors where each element holds one line of the corresponding BibTeX or LaTeX file.

**See Also**

[citEntry](#) and [sessionInfo](#) for examples

---

update.packages      *Download Packages from CRAN*

---

**Description**

These functions can be used to automatically compare the version numbers of installed packages with the newest available version on CRAN and update outdated packages on the fly.

**Usage**

```
update.packages(lib.loc = NULL, CRAN = getOption("CRAN"),
               contriburl = contrib.url(CRAN),
               method, instlib = NULL,
               ask = TRUE, available = NULL, destdir = NULL,
               installWithVers = FALSE)

CRAN.packages(CRAN = getOption("CRAN"), method,
              contriburl = contrib.url(CRAN))

old.packages(lib.loc = NULL, CRAN = getOption("CRAN"),
             contriburl = contrib.url(CRAN),
             method, available = NULL)

download.packages(pkgs, destdir, available = NULL,
                 CRAN = getOption("CRAN"),
                 contriburl = contrib.url(CRAN), method)

install.packages(pkgs, lib, CRAN = getOption("CRAN"),
                contriburl = contrib.url(CRAN),
                method, available = NULL, destdir = NULL,
                installWithVers = FALSE, dependencies = FALSE)
```

**Arguments**

lib.loc	character vector describing the location of R library trees to search through (and update packages therein).
CRAN	character, the base URL of the CRAN mirror to use, i.e., the URL of a CRAN root such as "http://cran.r-project.org" (the default) or its Statlib mirror, "http://lib.stat.cmu.edu/R/CRAN".
contriburl	URL of the contrib section of CRAN. Use this argument only if your CRAN mirror is incomplete, e.g., because you burned only the contrib section on a CD. Overrides argument CRAN.
method	Download method, see <a href="#">download.file</a> .

<code>pkgs</code>	character vector of the short names of packages whose current versions should be downloaded from CRAN.
<code>destdir</code>	directory where downloaded packages are stored.
<code>available</code>	list of packages available at CRAN as returned by <code>CRAN.packages</code> .
<code>lib</code>	character vector giving the library directories where to install the packages. Recycled as needed.
<code>ask</code>	logical indicating to ask before packages are actually downloaded and installed.
<code>installWithVers</code>	If TRUE, will invoke the install the package such that it can be referenced by package version
<code>instlib</code>	character string giving the library directory where to install the packages.
<code>dependencies</code>	logical indicating to also install uninstalled packages on which these packages depend/suggest/import (and so on recursively).

### Details

`CRAN.packages` returns a matrix of package details corresponding to packages/bundles currently available in the contrib section of CRAN, the comprehensive R archive network. The current list of packages is downloaded over the internet (or copied from a local CRAN mirror).

`old.packages` compares the information from `CRAN.packages` with that from `installed.packages` and reports installed packages/bundles that have newer versions on CRAN.

`download.packages` takes a list of package/bundle names and a destination directory, downloads the newest versions of the package sources and saves them in `destdir`. If the list of available packages is not given as argument, it is also directly obtained from CRAN. If CRAN is local, i.e., the URL starts with `"file:"`, then the packages are not downloaded but used directly.

The main function of the bundle is `update.packages`. First a list of all packages found in `lib.loc` is created and compared with the packages available on CRAN. Packages with a newer version are reported and for each one the user can specify if it should be updated. If so, the package sources are downloaded from CRAN and installed in the respective library path (or `instlib` if specified) using the R `INSTALL` mechanism.

`install.packages` can be used to install new packages. It takes a vector of package names and a destination library, downloads the packages from CRAN and installs them. If the library is omitted it defaults to the first directory in `.libPaths()`, with a warning if there is more than one. An attempt is made to install the packages in an order that respects their dependencies. This does assume that all the entries in `lib` are on the default library path for installs (set by `R_LIBS`).

For `install.packages` and `update.packages`, `destdir` is the directory to which packages will be downloaded. If it is NULL (the default) a temporary directory is used, and the user will be given the option of deleting the temporary files once the packages are installed. (They will always be deleted at the end of the R session.)

### Value

For `CRAN.packages`, a matrix with one row per package/bundle, row names the package names and column names "Package", "Version", "Priority", "Bundle", "Depends", "Imports", "Suggests" and "Contains".

For `old.packages`, `NULL` or a matrix with one row per package/bundle and column names "Package", "LibPath", "Installed" and "CRAN". (The last two columns are version numbers.)

For `download.packages`, a two-column matrix of names and destination file names, for those packages/bundles successfully downloaded.

`install.packages` and `update.packages` have no return value.

### See Also

[installed.packages](#).

See [download.file](#) for how to handle proxies and other options to monitor file transfers.

[INSTALL](#), [REMOVE](#), [library](#), [.packages](#), [read.dcf](#)

---

url.show

*Display a text URL*

---

### Description

Extension of [file.show](#) to display text files from a remote server.

### Usage

```
url.show(url, title = url, file = tempfile(),
         delete.file = TRUE, method, ...)
```

### Arguments

<code>url</code>	The URL to read from.
<code>title</code>	Title for the browser.
<code>file</code>	File to copy to.
<code>delete.file</code>	Delete the file afterwards?
<code>method</code>	File transfer method: see <a href="#">download.file</a>
<code>...</code>	Arguments to pass to <a href="#">file.show</a> .

### See Also

[url](#), [file.show](#), [download.file](#)

### Examples

```
## Not run: url.show("http://lib.stat.cmu.edu/datasets/csb/ch3a.txt")
```

---

`utils-deprecated`     *Deprecated Functions in Package `utils`*

---

**Description**

These functions are provided for compatibility with older versions of R only, and may be defunct as soon as of the next release.

**Usage****See Also**

[Deprecated](#), [Defunct](#)

---

`vignette`     *View or List Vignettes*

---

**Description**

View a specified vignette, or list the available ones.

**Usage**

```
vignette(topic, package = NULL, lib.loc = NULL)
```

**Arguments**

<code>topic</code>	a character string giving the (base) name of the vignette to view.
<code>package</code>	a character vector with the names of packages to search through, or <code>NULL</code> in which case <i>all</i> available packages in the library trees specified by <code>lib.loc</code> are searched.
<code>lib.loc</code>	a character vector of directory names of R libraries, or <code>NULL</code> . The default value of <code>NULL</code> corresponds to all libraries currently known.

**Details**

Currently, only PDF versions of vignettes can be viewed. The program specified by the `pdfviewer` option is used for this. If several vignettes have PDF versions with base name identical to `topic`, the first one found is used for viewing.

If no topics are given, the available vignettes are listed. The corresponding information is returned in an object of class "`packageIQR`". The structure of this class is experimental.

**Examples**

```
## List vignettes in all attached packages  
vignette()  
## List vignettes in all available packages  
vignette(package = .packages(all.available = TRUE))
```



# Index

- ! (*Logic*), 209
- != (*Comparison*), 60
- \*Topic **NA**
  - complete.cases, 875
  - factor, 129
  - NA, 234
  - na.action, 1031
  - na.fail, 1032
  - naprint, 1033
  - naresid, 1034
- \*Topic **algebra**
  - backsolve, 33
  - chol, 52
  - chol2inv, 54
  - colSums, 57
  - crossprod, 76
  - eigen, 113
  - matrix, 224
  - qr, 276
  - QR.Auxiliaries, 278
  - solve, 343
  - svd, 374
- \*Topic **aplot**
  - abline, 551
  - arrows, 552
  - axis, 555
  - box, 562
  - bxp, 568
  - contour, 572
  - coplot, 575
  - filled.contour, 580
  - frame, 585
  - grid, 585
  - Hershey, 519
  - image, 592
  - Japanese, 523
  - legend, 596
  - lines, 600
  - matplot, 603
  - mtext, 608
  - persp, 622
  - plot.window, 637
  - plot.xy, 638
  - plotmath, 530
  - points, 639
  - polygon, 641
  - rect, 643
  - rect.hclust, 1127
  - rug, 645
  - screen, 646
  - segments, 648
  - symbols, 658
  - text, 660
  - title, 662
- \*Topic **arith**
  - all.equal, 9
  - approxfun, 16
  - Arithmetic, 19
  - cumsum, 77
  - diff, 98
  - Extremes, 128
  - findInterval, 138
  - gl, 166
  - matmult, 223
  - ppoints, 1087
  - prod, 273
  - range, 289
  - Round, 314
  - sign, 338
  - sort, 344
  - sum, 372
  - tabulate, 392
- \*Topic **array**
  - addmargins, 817
  - aggregate, 819
  - aperm, 13
  - apply, 15
  - array, 20

- backsolve, 33
- cbind, 46
- chol, 52
- chol2inv, 54
- col, 56
- colSums, 57
- contrast, 879
- cor, 884
- crossprod, 76
- data.matrix, 83
- det, 95
- diag, 97
- dim, 101
- dimnames, 102
- drop, 106
- eigen, 113
- expand.grid, 120
- Extract, 121
- Extract.data.frame, 124
- kronecker, 189
- lm.fit, 991
- lower.tri, 212
- margin.table, 217
- mat.or.vec, 218
- matmult, 223
- matplot, 603
- matrix, 224
- maxCol, 225
- merge, 230
- nrow, 242
- outer, 257
- prop.table, 274
- qr, 276
- QR.Auxiliaries, 278
- row, 317
- row/colnames, 319
- scale, 324
- slice.index, 341
- svd, 374
- sweep, 376
- t, 389
- \*Topic **attribute**
  - attr, 30
  - attributes, 31
  - call, 43
  - comment, 59
  - length, 193
  - mode, 233
  - name, 236
  - names, 237
  - NULL, 247
  - numeric, 248
  - structure, 366
  - typeof, 412
  - which, 424
- \*Topic **category**
  - aggregate, 819
  - by, 41
  - cut, 78
  - Extract.factor, 127
  - factor, 129
  - ftable, 940
  - ftable.formula, 942
  - gl, 166
  - interaction, 177
  - levels, 194
  - loglin, 1002
  - nlevels, 240
  - plot.table, 636
  - read.ftable, 1125
  - split, 350
  - table, 390
  - tapply, 393
  - xtabs, 1231
- \*Topic **character**
  - abbreviate, 5
  - agrep, 7
  - char.expand, 48
  - character, 49
  - charmatch, 50
  - chartr, 51
  - delimMatch, 1239
  - format, 144
  - format.info, 148
  - formatC, 149
  - grep, 167
  - make.names, 214
  - make.unique, 215
  - nchar, 239
  - paste, 261
  - pmatch, 262
  - regex, 305
  - sprintf, 351
  - sQuote, 353
  - strsplit, 364
  - strwidth, 654

- strwrap, 366
- substr, 370
- symnum, 1192
- \*Topic **chron**
  - as.POSIX\*, 24
  - axis.POSIXct, 557
  - cut.POSIXt, 80
  - Dates, 85
  - DateTimeClasses, 85
  - difftime, 99
  - format.Date, 146
  - hist.POSIXt, 589
  - rep, 310
  - round.POSIXt, 316
  - seq.Date, 332
  - seq.POSIXt, 333
  - strptime, 361
  - Sys.time, 385
  - weekdays, 423
- \*Topic **classes**
  - as, 738
  - as.data.frame, 21
  - BasicClasses, 742
  - callNextMethod, 743
  - character, 49
  - class, 55
  - Classes, 745
  - classRepresentation-class, 746
  - data.class, 81
  - data.frame, 82
  - Documentation, 747
  - double, 103
  - environment-class, 749
  - fixPre1.8, 750
  - genericFunction-class, 751
  - GenericFunctions, 752
  - getClass, 756
  - getMethod, 757
  - integer, 176
  - is, 764
  - is.object, 183
  - is.recursive, 185
  - is.single, 186
  - isSealedMethod, 767
  - language-class, 768
  - LinearMethodsList-class, 769
  - logical, 211
  - makeClassRepresentation, 770
  - MethodDefinition-class, 771
  - Methods, 772
  - MethodsList-class, 775
  - MethodWithNext-class, 776
  - new, 777
  - numeric, 248
  - ObjectsWithPackage-class, 779
  - promptClass, 779
  - raw, 291
  - rawConversion, 292
  - real, 303
  - representation, 782
  - row.names, 318
  - SClassExtension-class, 784
  - setClass, 785
  - setClassUnion, 789
  - setMethod, 795
  - signature-class, 803
  - slot, 804
  - StructureClasses, 806
  - TraceClasses, 806
  - validObject, 808
  - vector, 420
- \*Topic **cluster**
  - as.hclust, 848
  - cophenetic, 882
  - cutree, 891
  - dist, 907
  - hclust, 953
  - identify.hclust, 963
  - kmeans, 979
  - rect.hclust, 1127
- \*Topic **color**
  - col2rgb, 510
  - colors, 512
  - gray, 518
  - hsv, 522
  - palette, 524
  - Palettes, 525
  - rgb, 543
  - rgb2hsv, 544
- \*Topic **complex**
  - complex, 61
- \*Topic **connection**
  - cat, 45
  - connections, 67
  - dput, 105

- dump, 107
- gzcon, 172
- parse, 260
- pushBack, 275
- read.00Index, 1250
- read.fortran, 1313
- read.fwf, 1314
- read.table, 294
- readBin, 298
- readLines, 302
- scan, 325
- seek, 329
- showConnections, 336
- sink, 339
- socketSelect, 342
- source, 346
- textConnection, 400
- write, 428
- writeLines, 431
- \*Topic **datasets**
  - ability.cov, 435
  - airmiles, 436
  - AirPassengers, 437
  - airquality, 438
  - anscombe, 439
  - attenu, 440
  - attitude, 441
  - austres, 442
  - beavers, 442
  - BJsales, 444
  - BOD, 444
  - cars, 445
  - ChickWeight, 446
  - chickwts, 447
  - CO2, 448
  - co2, 449
  - data, 1268
  - discoveries, 450
  - DNase, 451
  - esoph, 452
  - euro, 453
  - eurodist, 454
  - EuStockMarkets, 454
  - faithful, 455
  - Formaldehyde, 456
  - freeny, 457
  - HairEyeColor, 458
  - Harman23.cor, 459
  - Harman74.cor, 459
  - Indometh, 460
  - infert, 461
  - InsectSprays, 462
  - iris, 463
  - islands, 464
  - JohnsonJohnson, 465
  - LakeHuron, 465
  - lh, 466
  - LifeCycleSavings, 466
  - Loblolly, 467
  - longley, 468
  - lynx, 469
  - morley, 470
  - mtcars, 471
  - nhtemp, 471
  - Nile, 472
  - nottem, 473
  - Orange, 474
  - OrchardSprays, 475
  - PlantGrowth, 476
  - precip, 477
  - presidents, 477
  - pressure, 478
  - Puromycin, 479
  - quakes, 480
  - randu, 481
  - rivers, 482
  - rock, 483
  - sleep, 483
  - stackloss, 484
  - state, 485
  - sunspot.month, 487
  - sunspot.year, 488
  - sunspots, 488
  - swiss, 489
  - Theoph, 490
  - Titanic, 491
  - ToothGrowth, 493
  - treering, 493
  - trees, 494
  - UCBAdmissions, 495
  - UKDriverDeaths, 496
  - UKgas, 497
  - UKLungDeaths, 498
  - USAccDeaths, 499
  - USArrests, 499
  - USJudgeRatings, 500

- USPersonalExpenditure, 501
- uspop, 501
- VADeaths, 502
- volcano, 503
- warpbreaks, 504
- women, 505
- WorldPhones, 505
- WWWusage, 506
- \*Topic **data**
  - apropos, 1255
  - as.environment, 23
  - assign, 26
  - assignOps, 28
  - attach, 29
  - autoload, 32
  - bquote, 39
  - delay, 90
  - deparse, 91
  - detach, 96
  - environment, 114
  - eval, 116
  - exists, 118
  - force, 139
  - get, 156
  - getAnywhere, 1283
  - getFromNamespace, 1284
  - getS3method, 1285
  - libPaths, 195
  - library, 196
  - library.dynam, 199
  - ns-load, 245
  - search, 328
  - substitute, 369
  - sys.parent, 381
  - with, 427
  - zpackages, 432
- \*Topic **debugging**
  - recover, 1317
  - trace, 404
- \*Topic **design**
  - contrast, 879
  - contrasts, 880
  - TukeyHSD, 1213
- \*Topic **device**
  - .Device, 1
  - dev.interactive, 512
  - dev.xxx, 513
  - dev2, 514
  - Devices, 517
  - pdf, 526
  - pictex, 528
  - png, 533
  - postscript, 535
  - postscriptFonts, 539
  - quartz, 541
  - quartzFonts, 542
  - screen, 646
  - x11, 547
  - X11Fonts, 548
  - xfig, 549
- \*Topic **distribution**
  - bandwidth, 850
  - Beta, 853
  - Binomial, 856
  - birthday, 860
  - Cauchy, 866
  - chisq.test, 868
  - Chisquare, 870
  - density, 899
  - Exponential, 917
  - FDist, 926
  - fivenum, 933
  - GammaDist, 943
  - Geometric, 945
  - hist, 586
  - Hypergeometric, 962
  - IQR, 972
  - Logistic, 999
  - Lognormal, 1004
  - Multinomial, 1029
  - NegBinomial, 1035
  - Normal, 1048
  - Poisson, 1078
  - ppoints, 1087
  - qqnorm, 1120
  - r2dtable, 282
  - Random, 283
  - Random.user, 287
  - sample, 321
  - SignRank, 1145
  - stem, 652
  - TDist, 1197
  - Tukey, 1212
  - Uniform, 1214
  - Weibull, 1222
  - Wilcoxon, 1228

**\*Topic documentation**

apropos, 1255  
 args, 18  
 buildVignettes, 1233  
 checkTnF, 1235  
 checkVignettes, 1236  
 codoc, 1237  
 data, 1268  
 Defunct, 90  
 demo, 1274  
 Deprecated, 94  
 Documentation, 747  
 example, 1279  
 help, 1287  
 help.search, 1290  
 help.start, 1292  
 NotYet, 241  
 prompt, 1310  
 promptData, 1312  
 QC, 1246  
 Rdindex, 1247  
 Rdutils, 1248  
 str, 1326  
 Syntax, 378  
 undoc, 1252  
 vignette, 1336

**\*Topic dplot**

absolute.size, 667  
 approxfun, 16  
 axTicks, 558  
 boxplot.stats, 566  
 col2rgb, 510  
 colors, 512  
 convertNative, 668  
 convolve, 881  
 dataViewport, 669  
 drawDetails, 670  
 ecdf, 911  
 editDetails, 671  
 expression, 120  
 fft, 927  
 gEdit, 672  
 gpar, 673  
 gPath, 675  
 Grid, 676  
 Grid Viewports, 676  
 grid.add, 680  
 grid.arrows, 681

grid.circle, 684  
 grid.collection, 685  
 grid.convert, 686  
 grid.copy, 688  
 grid.display.list, 689  
 grid.draw, 690  
 grid.edit, 691  
 grid.frame, 692  
 grid.get, 693  
 grid.grab, 694  
 grid.grill, 695  
 grid.grob, 696  
 grid.layout, 697  
 grid.lines, 699  
 grid.locator, 700  
 grid.move.to, 702  
 grid.newpage, 703  
 grid.pack, 704  
 grid.place, 705  
 grid.plot.and.legend, 706  
 grid.points, 707  
 grid.polygon, 708  
 grid.pretty, 709  
 grid.prompt, 710  
 grid.rect, 711  
 grid.remove, 712  
 grid.segments, 713  
 grid.set, 714  
 grid.show.layout, 715  
 grid.show.viewport, 716  
 grid.text, 717  
 grid.xaxis, 719  
 grid.yaxis, 720  
 grobWidth, 722  
 hist, 586  
 hist.POSIXt, 589  
 hsv, 522  
 jitter, 186  
 layout, 594  
 n2mfrow, 610  
 Palettes, 525  
 panel.smooth, 614  
 par, 615  
 plot.density, 1067  
 plotViewport, 722  
 pop.viewport, 723  
 ppoints, 1087  
 pretty, 265

- push.viewport, 724
- Querying the Viewport Tree, 724
- rgb2hsv, 544
- screen, 646
- splinefun, 1160
- stepfun, 1176
- stringWidth, 726
- strwidth, 654
- unit, 726
- unit.c, 728
- unit.length, 729
- unit.pmin, 730
- unit.rep, 730
- units, 546
- validDetails, 731
- vpPath, 732
- widthDetails, 733
- Working with Viewports, 734
- xy.coords, 663
- xyz.coords, 665
- \*Topic environment**
  - apropos, 1255
  - as.environment, 23
  - browser, 40
  - commandArgs, 58
  - debug, 89
  - eapply, 112
  - gc, 154
  - gctorture, 156
  - interactive, 178
  - is.R, 184
  - layout, 594
  - ls, 212
  - Memory, 227
  - Memory-limits, 229
  - options, 250
  - par, 615
  - quit, 279
  - R.Version, 281
  - reg.finalizer, 304
  - remove, 309
  - Startup, 356
  - stop, 358
  - stopifnot, 360
  - Sys.getenv, 379
  - Sys.putenv, 383
  - taskCallback, 395
  - taskCallbackManager, 397
  - taskCallbackNames, 398
- \*Topic error**
  - bug.report, 1259
  - conditions, 63
  - debugger, 1272
  - options, 250
  - stop, 358
  - stopifnot, 360
  - warning, 421
  - warnings, 422
- \*Topic file**
  - .Platform, 3
  - basename, 35
  - browseURL, 1259
  - cat, 45
  - connections, 67
  - count.fields, 75
  - dataentry, 1270
  - dcf, 88
  - dput, 105
  - dump, 107
  - file.access, 131
  - file.choose, 133
  - file.info, 133
  - file.path, 135
  - file.show, 135
  - files, 136
  - fileutils, 1240
  - gzcon, 172
  - list.files, 203
  - load, 204
  - package.skeleton, 1304
  - parse, 260
  - path.expand, 262
  - read.00Index, 1250
  - read.fortran, 1313
  - read.fwf, 1314
  - read.table, 294
  - readBin, 298
  - readLines, 302
  - save, 322
  - scan, 325
  - seek, 329
  - sink, 339
  - source, 346
  - sys.source, 385
  - system, 386

- system.file, 387
- tempfile, 399
- textConnection, 400
- unlink, 414
- url.show, 1335
- write, 428
- write.table, 429
- writeLines, 431
- zip.file.extract, 431
- \*Topic **graphs**
  - chull, 571
- \*Topic **hplot**
  - assocplot, 554
  - barplot, 559
  - biplot, 858
  - biplot.princomp, 859
  - boxplot, 563
  - contour, 572
  - coplot, 575
  - cpgram, 890
  - curve, 578
  - dendrogram, 895
  - dotchart, 579
  - ecdf, 911
  - filled.contour, 580
  - fourfoldplot, 583
  - heatmap, 956
  - hist, 586
  - hist.POSIXt, 589
  - image, 592
  - interaction.plot, 970
  - lag.plot, 986
  - matplot, 603
  - monthplot, 1026
  - mosaicplot, 605
  - pairs, 612
  - panel.smooth, 614
  - persp, 622
  - pie, 625
  - plot, 626
  - plot.acf, 1066
  - plot.data.frame, 628
  - plot.default, 629
  - plot.design, 631
  - plot.factor, 633
  - plot.formula, 634
  - plot.histogram, 635
  - plot.isoreg, 1069
  - plot.lm, 1070
  - plot.ppr, 1072
  - plot.spec, 1074
  - plot.stepfun, 1075
  - plot.table, 636
  - plot.ts, 1077
  - qqnorm, 1120
  - stars, 649
  - stripchart, 653
  - sunflowerplot, 656
  - symbols, 658
  - termpplot, 1198
- \*Topic **htest**
  - ansari.test, 828
  - bartlett.test, 852
  - binom.test, 855
  - chisq.test, 868
  - cor.test, 886
  - fisher.test, 930
  - fligner.test, 934
  - friedman.test, 938
  - kruskal.test, 980
  - ks.test, 982
  - mantelhaen.test, 1014
  - mcnemar.test, 1017
  - mood.test, 1028
  - oneway.test, 1052
  - p.adjust, 1061
  - pairwise.prop.test, 1063
  - pairwise.t.test, 1063
  - pairwise.table, 1064
  - pairwise.wilcox.test, 1065
  - power.anova.test, 1082
  - power.prop.test, 1083
  - power.t.test, 1085
  - print.power.htest, 1107
  - prop.test, 1117
  - prop.trend.test, 1119
  - quade.test, 1121
  - shapiro.test, 1144
  - t.test, 1195
  - var.test, 1219
  - wilcox.test, 1225
- \*Topic **interface**
  - .Script, 4
  - browseEnv, 1257
  - dyn.load, 109
  - getDLLRegisteredRoutines, 159

- getLoadedDLLs, 160
- getNativeSymbolInfo, 161
- getNumCConverters, 163
- Internal, 179
- Primitive, 267
- system, 386
- \*Topic **iplot**
  - dev.xxx, 513
  - frame, 585
  - identify, 591
  - identify.hclust, 963
  - layout, 594
  - locator, 602
  - par, 615
  - plot.histogram, 635
  - recordPlot, 543
- \*Topic **iteration**
  - apply, 15
  - by, 41
  - Control, 74
  - dendrapply, 894
  - eapply, 112
  - identical, 174
  - lapply, 191
  - sweep, 376
  - tapply, 393
- \*Topic **list**
  - clearNames, 872
  - eapply, 112
  - Extract, 121
  - lapply, 191
  - list, 202
  - NULL, 247
  - setNames, 1143
  - unlist, 415
- \*Topic **loess**
  - loess, 996
  - loess.control, 998
- \*Topic **logic**
  - all, 8
  - all.equal, 9
  - any, 12
  - Comparison, 60
  - complete.cases, 875
  - Control, 74
  - duplicated, 108
  - identical, 174
  - ifelse, 175
  - Logic, 209
  - logical, 211
  - match, 219
  - NA, 234
  - unique, 413
  - which, 424
- \*Topic **manip**
  - addmargins, 817
  - append, 14
  - c, 42
  - cbind, 46
  - cut.POSIXt, 80
  - deparse, 91
  - dimnames, 102
  - duplicated, 108
  - expand.model.frame, 916
  - getInitial, 946
  - head, 1286
  - list, 202
  - mapply, 216
  - match, 219
  - merge, 230
  - model.extract, 1020
  - NA, 234
  - NLSstAsymptotic, 1045
  - NLSstClosestX, 1046
  - NLSstLfAsymptote, 1047
  - NLSstRtAsymptote, 1048
  - NULL, 247
  - order, 255
  - order.dendrogram, 1060
  - reorder, 1129
  - rep, 310
  - replace, 312
  - reshape, 1132
  - rev, 313
  - rle, 313
  - row/colnames, 319
  - rowsum, 320
  - seq, 330
  - seq.Date, 332
  - seq.POSIXt, 333
  - sequence, 334
  - slotOp, 342
  - sort, 344
  - sortedXyData, 1153
  - stack, 354
  - structure, 366

- subset, 367
- transform, 408
- type.convert, 411
- unique, 413
- unlist, 415
- \*Topic **math**
  - .Machine, 1
  - abs, 6
  - Bessel, 36
  - convolve, 881
  - deriv, 902
  - fft, 927
  - Hyperbolic, 173
  - integrate, 968
  - is.finite, 181
  - kappa, 187
  - log, 208
  - nextn, 1037
  - poly, 1080
  - polyroot, 264
  - Special, 348
  - splinefun, 1160
  - Trig, 409
- \*Topic **methods**
  - .BasicFunsList, 737
  - addmargins, 817
  - as, 738
  - as.data.frame, 21
  - callNextMethod, 743
  - class, 55
  - Classes, 745
  - data.class, 81
  - data.frame, 82
  - Documentation, 747
  - GenericFunctions, 752
  - getMethod, 757
  - groupGeneric, 169
  - initialize-methods, 762
  - InternalMethods, 179
  - is, 764
  - is.object, 183
  - isSealedMethod, 767
  - Methods, 772
  - methods, 1301
  - MethodsList-class, 775
  - na.action, 1031
  - noquote, 240
  - plot.data.frame, 628
  - predict, 1093
  - promptMethods, 781
  - row.names, 318
  - setClass, 785
  - setGeneric, 791
  - setMethod, 795
  - setOldClass, 798
  - showMethods, 802
  - summary, 372
  - UseMethod, 417
- \*Topic **misc**
  - base-deprecated, 34
  - citation, 1263
  - citEntry, 1264
  - close.socket, 1266
  - contributors, 73
  - copyright, 75
  - license, 201
  - make.socket, 1299
  - person, 1308
  - read.socket, 1316
  - sessionInfo, 1325
  - sets, 335
  - toLatex, 1332
  - tools-deprecated, 1251
  - url.show, 1335
  - utils-deprecated, 1336
- \*Topic **models**
  - add1, 815
  - AIC, 821
  - alias, 822
  - anova, 824
  - anova.glm, 825
  - anova.lm, 826
  - aov, 830
  - AsIs, 25
  - asOneSidedFormula, 849
  - C, 863
  - case/variable.names, 865
  - coef, 874
  - confint, 876
  - deviance, 904
  - df.residual, 905
  - dummy.coef, 909
  - eff.aovlist, 913
  - effects, 914
  - expand.grid, 120
  - extractAIC, 918

- factor.scope, 923
- family, 924
- fitted, 932
- formula, 935
- formula.nls, 937
- getInitial, 946
- glm, 947
- glm.control, 951
- glm.summaries, 952
- is.empty.model, 973
- labels, 190
- lm.summaries, 994
- logLik, 1001
- loglin, 1002
- make.link, 1011
- makepredictcall, 1012
- manova, 1013
- model.extract, 1020
- model.frame, 1021
- model.matrix, 1023
- model.tables, 1025
- naprint, 1033
- naresid, 1034
- nls, 1040
- nls.control, 1042
- nlsModel, 1044
- numericDeriv, 1050
- offset, 1051
- plot.profile.nls, 1073
- power, 1081
- predict.glm, 1096
- predict.nls, 1102
- preplot, 1104
- profile, 1110
- profile.nls, 1111
- profiler, 1112
- profiler.nls, 1113
- proj, 1115
- relevel, 1128
- replications, 1131
- residuals, 1134
- se.contrast, 1140
- selfStart, 1142
- SSasymp, 1162
- SSasympOff, 1163
- SSasympOrig, 1164
- SSbiexp, 1165
- SSfol, 1166
- SSfpl, 1167
- SSgompertz, 1168
- SSlogis, 1169
- SSmicmen, 1170
- SSweibull, 1171
- stat.anova, 1173
- step, 1174
- summary.aov, 1184
- summary.glm, 1185
- summary.lm, 1187
- summary.manova, 1189
- terms, 1200
- terms.formula, 1201
- terms.object, 1202
- tilde, 402
- TukeyHSD, 1213
- update, 1217
- update.formula, 1218
- vcov, 1222
- \*Topic **multivariate**
  - as.hclust, 848
  - biplot, 858
  - biplot.princomp, 859
  - cancor, 864
  - cmdscale, 872
  - cophenetic, 882
  - cor, 884
  - cov.wt, 889
  - cutree, 891
  - dendrogram, 895
  - dist, 907
  - factanal, 919
  - hclust, 953
  - kmeans, 979
  - loadings, 995
  - mahalanobis, 1010
  - prcomp, 1091
  - princomp, 1105
  - screeplot, 1139
  - stars, 649
  - summary.princomp, 1190
  - symbols, 658
  - varimax, 1220
- \*Topic **nonlinear**
  - CO2, 448
  - deriv, 902
  - getInitial, 946
  - nlm, 1037

- nls, 1040
- nls.control, 1042
- nlsModel, 1044
- optim, 1053
- plot.profile.nls, 1073
- predict.nls, 1102
- profile.nls, 1111
- profiler, 1112
- profiler.nls, 1113
- vcov, 1222
- \*Topic **nonparametric**
  - sunflowerplot, 656
- \*Topic **optimize**
  - constrOptim, 877
  - glm.control, 951
  - nlm, 1037
  - optim, 1053
  - optimize, 1058
  - uniroot, 1216
- \*Topic **print**
  - cat, 45
  - dcf, 88
  - format, 144
  - format.info, 148
  - formatC, 149
  - formatDL, 151
  - labels, 190
  - loadings, 995
  - ls.str, 1297
  - noquote, 240
  - octmode, 249
  - options, 250
  - plot.isoreg, 1069
  - print, 268
  - print.data.frame, 269
  - print.default, 270
  - printCoefmat, 1109
  - prmatrix, 271
  - sprintf, 351
  - str, 1326
  - write.table, 429
- \*Topic **programming**
  - .BasicFunsList, 737
  - .Machine, 1
  - all.names, 11
  - as, 738
  - as.function, 23
  - autoload, 32
  - body, 38
  - bquote, 39
  - browser, 40
  - call, 43
  - callNextMethod, 743
  - check.options, 509
  - checkFF, 1234
  - Classes, 745
  - commandArgs, 58
  - conditions, 63
  - Control, 74
  - debug, 89
  - delay, 90
  - delete.response, 893
  - deparse, 91
  - deparseOpts, 93
  - do.call, 103
  - Documentation, 747
  - dput, 105
  - environment, 114
  - eval, 116
  - expression, 120
  - fixPre1.8, 750
  - force, 139
  - Foreign, 140
  - formals, 143
  - format.info, 148
  - function, 153
  - GenericFunctions, 752
  - getCallingDLL, 158
  - getClass, 756
  - getMethod, 757
  - getNumCConverters, 163
  - getPackageName, 760
  - hasArg, 761
  - identical, 174
  - ifelse, 175
  - initialize-methods, 762
  - interactive, 178
  - invisible, 180
  - is, 764
  - is.finite, 181
  - is.function, 182
  - is.language, 183
  - is.recursive, 185
  - isSealedMethod, 767
  - Last.value, 192
  - makeClassRepresentation, 770

- match.arg, 220
- match.call, 221
- match.fun, 222
- menu, 1301
- Methods, 772
- missing, 232
- model.extract, 1020
- name, 236
- nargs, 238
- new, 777
- ns-dblcolon, 243
- ns-topenv, 246
- on.exit, 250
- Paren, 259
- parse, 260
- promptClass, 779
- promptMethods, 781
- R.Version, 281
- Recall, 304
- recover, 1317
- reg.finalizer, 304
- representation, 782
- setClass, 785
- setClassUnion, 789
- setGeneric, 791
- setMethod, 795
- setOldClass, 798
- show, 800
- slot, 804
- source, 346
- stop, 358
- stopifnot, 360
- substitute, 369
- switch, 377
- sys.parent, 381
- trace, 404
- traceback, 407
- try, 410
- validObject, 808
- warning, 421
- warnings, 422
- with, 427
- \*Topic **regression**
  - anova, 824
  - anova.glm, 825
  - anova.lm, 826
  - aov, 830
  - case/variable.names, 865
  - coef, 874
  - contrast, 879
  - contrasts, 880
  - df.residual, 905
  - effects, 914
  - expand.model.frame, 916
  - fitted, 932
  - glm, 947
  - glm.summaries, 952
  - influence.measures, 965
  - isoreg, 973
  - line, 987
  - lm, 988
  - lm.fit, 991
  - lm.influence, 992
  - lm.summaries, 994
  - ls.diag, 1006
  - ls.print, 1007
  - lsfit, 1008
  - nls, 1040
  - nls.control, 1042
  - plot.lm, 1070
  - plot.profile.nls, 1073
  - ppr, 1088
  - predict.glm, 1096
  - predict.lm, 1098
  - predict.nls, 1102
  - profile.nls, 1111
  - profiler.nls, 1113
  - residuals, 1134
  - stat.anova, 1173
  - summary.aov, 1184
  - summary.glm, 1185
  - summary.lm, 1187
  - termplot, 1198
  - weighted.residuals, 1224
- \*Topic **robust**
  - fivenum, 933
  - IQR, 972
  - line, 987
  - mad, 1009
  - median, 1018
  - medpolish, 1019
  - runmed, 1135
  - smooth, 1146
  - smoothEnds, 1152
- \*Topic **smooth**
  - bandwidth, 850

- density, 899
- isoreg, 973
- ksmooth, 984
- loess, 996
- loess.control, 998
- lowess, 1005
- predict.loess, 1100
- predict.smooth.spline, 1103
- runmed, 1135
- scatter.smooth, 1137
- smooth, 1146
- smooth.spline, 1148
- smoothEnds, 1152
- sunflowerplot, 656
- supsmu, 1191
- \*Topic **sysdata**
  - .Machine, 1
  - colors, 512
  - commandArgs, 58
  - Constants, 72
  - NULL, 247
  - palette, 524
  - R.Version, 281
  - Random, 283
  - Random.user, 287
- \*Topic **tree**
  - dendrogram, 895
- \*Topic **ts**
  - acf, 812
  - acf2AR, 814
  - ar, 832
  - ar.ols, 835
  - arima, 837
  - arima.sim, 841
  - arima0, 842
  - ARMAacf, 846
  - ARMAtoMA, 847
  - Box.test, 862
  - cpgram, 890
  - decompose, 892
  - diffinv, 906
  - embed, 915
  - filter, 928
  - HoltWinters, 959
  - KalmanLike, 975
  - kernapply, 977
  - kernel, 978
  - lag, 985
  - lag.plot, 986
  - monthplot, 1026
  - na.contiguous, 1031
  - plot.acf, 1066
  - plot.HoltWinters, 1068
  - plot.spec, 1074
  - plot.ts, 1077
  - PP.test, 1086
  - predict.Arima, 1094
  - predict.HoltWinters, 1097
  - print.ts, 1108
  - spec.ar, 1154
  - spec.pgram, 1155
  - spec.taper, 1157
  - spectrum, 1158
  - start, 1172
  - stl, 1178
  - stlmethods, 1180
  - StructTS, 1181
  - time, 1203
  - toeplitz, 1204
  - ts, 1204
  - ts-methods, 1206
  - ts.plot, 1207
  - ts.union, 1208
  - tsdiag, 1209
  - tsp, 1210
  - tsSmooth, 1211
  - window, 1230
- \*Topic **univar**
  - ave, 849
  - cor, 884
  - Extremes, 128
  - fivenum, 933
  - IQR, 972
  - mad, 1009
  - mean, 226
  - median, 1018
  - nclass, 611
  - order, 255
  - quantile, 1123
  - range, 289
  - rank, 290
  - sd, 1140
  - sort, 344
  - stem, 652
  - weighted.mean, 1224
- \*Topic **utilities**

.Platform, 3  
.checkMFClasses, 811  
all.equal, 9  
as.POSIX\*, 24  
axis.POSIXct, 557  
BATCH, 1256  
bug.report, 1259  
buildVignettes, 1233  
builtins, 40  
capabilities, 44  
capture.output, 1262  
check.options, 509  
checkFF, 1234  
checkMD5sums, 1235  
checkTnF, 1235  
checkVignettes, 1236  
compareVersion, 1266  
COMPILE, 1267  
conflicts, 66  
dataentry, 1270  
date, 84  
Dates, 85  
DateTimeClasses, 85  
debugger, 1272  
Defunct, 90  
demo, 1274  
Deprecated, 94  
dev2bitmap, 516  
difftime, 99  
download.file, 1275  
edit, 1276  
edit.data.frame, 1278  
example, 1279  
file.edit, 1281  
findInterval, 138  
fix, 1282  
format.Date, 146  
gc.time, 155  
getDepList, 1241  
getpid, 164  
getwd, 165  
grep, 167  
index.search, 1293  
INSTALL, 1294  
installed.packages, 1296  
installFoundDepends, 1243  
integrate, 968  
is.R, 184  
jitter, 186  
LINK, 1297  
localeconv, 206  
locales, 207  
ls.str, 1297  
make.packages.html, 1299  
makeLazyLoading, 1244  
manglePackageName, 216  
mapply, 216  
maxCol, 225  
md5sum, 1245  
memory.profile, 230  
menu, 1301  
n2mfrow, 610  
noquote, 240  
NotYet, 241  
ns-hooks, 244  
ns-load, 245  
nsl, 1303  
object.size, 1303  
package-version, 258  
package.dependencies, 1245  
package.skeleton, 1304  
packageDescription, 1305  
packageStatus, 1306  
page, 1308  
PkgUtils, 1309  
pos.to.env, 265  
proc.time, 272  
QC, 1246  
R.home, 281  
Rdindex, 1247  
RdUtils, 293  
Rdutils, 1248  
readline, 301  
relevel, 1128  
REMOVE, 1319  
remove.packages, 1319  
reorder.factor, 1130  
RHOME, 1320  
Rprof, 1320  
Rtangle, 1321  
RweaveLatex, 1322  
savehistory, 1324  
SHLIB, 1325  
shQuote, 337  
Signals, 339  
str, 1326

- strptime, 361
- summaryRprof, 1328
- Sweave, 1329
- SweaveSyntConv, 1331
- symnum, 1192
- Sys.getenv, 379
- Sys.info, 380
- Sys.putenv, 383
- Sys.sleep, 384
- sys.source, 385
- Sys.time, 385
- system, 386
- system.file, 387
- system.time, 388
- texi2dvi, 1251
- toString, 403
- uname, 416
- update.packages, 1333
- UserHooks, 418
- vignetteDepends, 1253
- which.min, 426
- \* (Arithmetic), 19
- \*.difftime (difftime), 99
- + (Arithmetic), 19
- +.Date (Dates), 85
- +.POSIXt (DateTimeClasses), 85
- (Arithmetic), 19
- .Date (Dates), 85
- .POSIXt (DateTimeClasses), 85
- > (assignOps), 28
- >> (assignOps), 28
- .AutoloadEnv (autoload), 32
- .Autoloaded (autoload), 32
- .BaseNamespaceEnv (environment), 114
- .BasicFunsList, 737
- .C, 104, 110, 112, 158, 159, 161–164, 179, 1234
- .C (Foreign), 140
- .Call, 110, 112, 158, 159, 161, 162
- .Call (Foreign), 140
- .Class (groupGeneric), 169
- .Defunct (Defunct), 90
- .Deprecated (Deprecated), 94
- .Device, 1, 541
- .Devices (.Device), 1
- .External, 110, 112, 158, 159, 161, 162
- .External (Foreign), 140
- .First, 178, 280
- .First (Startup), 356
- .First.lib, 111, 200, 201, 244
- .First.lib (library), 196
- .Fortran, 104, 110, 112, 158, 159, 161, 162, 179, 1234
- .Fortran (Foreign), 140
- .Generic (groupGeneric), 169
- .GlobalEnv, 328, 369, 381
- .GlobalEnv (environment), 114
- .Group (groupGeneric), 169
- .InitTraceFunctions (TraceClasses), 806
- .Internal, 41, 267
- .Internal (Internal), 179
- .Last, 339, 356, 357, 1324
- .Last (quit), 279
- .Last.lib, 200, 244
- .Last.lib (library), 196
- .Last.value (Last.value), 192
- .Library (libPaths), 195
- .MFclass, 1202
- .MFclass (.checkMFClasses), 811
- .Machine, 1, 4, 299, 1059
- .Method (groupGeneric), 169
- .NotYetImplemented (NotYet), 241
- .NotYetUsed (NotYet), 241
- .OldClassesList (setOldClass), 798
- .Options (options), 250
- .Pars (par), 615
- .Platform, 3, 3, 45, 282, 380, 387
- .Primitive, 179, 259
- .Primitive (Primitive), 267
- .Random.seed, 1049, 1215
- .Random.seed (Random), 283
- .Renviron (Startup), 356
- .Rprofile (Startup), 356
- .Script, 4
- .Traceback (traceback), 407
- .checkMFClasses, 811
- .decode\_package\_version (package-version), 258
- .deparseOpts, 92, 105, 107
- .deparseOpts (deparseOpts), 93
- .doTracePrint (TraceClasses), 806
- .dynLibs (library.dynam), 199
- .encode\_package\_version (package-version), 258

- .getXlevels (.checkMFClasses), 811
- .handleSimpleError (conditions), 63
- .helpForCall (help), 1287
- .isMethodsDispatchOn (UseMethod), 417
- .leap.seconds (DateTimeClasses), 85
- .libPaths, 199, 201, 433, 1269
- .libPaths (libPaths), 195
- .makeTracedFunction (TraceClasses), 806
- .noGenerics (library), 196
- .onAttach, 246
- .onAttach (ns-hooks), 244
- .onLoad, 197, 200, 245
- .onLoad (ns-hooks), 244
- .onUnload, 200
- .onUnload (ns-hooks), 244
- .packages, 199, 201, 329, 1335
- .packages (zpackages), 432
- .primTrace (trace), 404
- .primUntrace (trace), 404
- .ps.prolog (postscript), 535
- .setOldIs (setOldClass), 798
- .signalSimpleWarning (conditions), 63
- .tryHelp (help), 1287
- .untracedFunction (TraceClasses), 806
- .userHooksEnv (UserHooks), 418
- / (Arithmetic), 19
- / .difftime (difftime), 99
- :, 178
- : (seq), 330
- :: (ns-dblcolon), 243
- :::, 1284
- ::: (ns-dblcolon), 243
- < (Comparison), 60
- <-, 27
- <- (assignOps), 28
- <-class (language-class), 768
- <= (Comparison), 60
- <<- (assignOps), 28
- = (assignOps), 28
- ==, 10
- == (Comparison), 60
- > (Comparison), 60
- >= (Comparison), 60
- ? (help), 1287
- [, 106, 179, 368
- [ (Extract), 121
- [ .AsIs (AsIs), 25
- [ .Date (Dates), 85
- [ .POSIXct (DateTimeClasses), 85
- [ .POSIXlt (DateTimeClasses), 85
- [ .acf (acf), 812
- [ .data.frame, 83, 122, 123, 1022
- [ .data.frame (Extract.data.frame), 124
- [ .difftime (difftime), 99
- [ .factor, 122, 123, 130, 131
- [ .factor (Extract.factor), 127
- [ .formula (formula), 935
- [ .getAnywhere (getAnywhere), 1283
- [ .noquote (noquote), 240
- [ .octmode (octmode), 249
- [ .package\_version (package-version), 258
- [ .terms (delete.response), 893
- [ .ts (ts), 1204
- [ <-, 179
- [ <- (Extract), 121
- [ <- .Date (Dates), 85
- [ <- .POSIXct (DateTimeClasses), 85
- [ <- .POSIXlt (DateTimeClasses), 85
- [ <- .data.frame (Extract.data.frame), 124
- [ <- .factor (Extract.factor), 127
- [ [, 179, 898
- [ [ (Extract), 121
- [ [ .Date (Dates), 85
- [ [ .POSIXct (DateTimeClasses), 85
- [ [ .data.frame (Extract.data.frame), 124
- [ [ .dendrogram (dendrogram), 895
- [ [ .package\_version (package-version), 258
- [ [ <-, 179
- [ [ <- (Extract), 121
- [ [ <- .data.frame (Extract.data.frame), 124
- \$, 179
- \$ (Extract), 121
- \$.package\_version (package-version), 258

- \$<- , 179
- \$<- (Extract), 121
- \$<- .data.frame  
(Extract.data.frame), 124
- %\*%, 77, 189, 257
- %\*% (matmult), 223
- %/% (Arithmetic), 19
- %% (Arithmetic), 19
- %in%, 335
- %in% (match), 219
- %o%, 77
- %o% (outer), 257
- %x% (kronecker), 189
- & (Logic), 209
- && (Logic), 209
- \_\_ClassMetaData (Classes), 745
- ^ (Arithmetic), 19
- ~ (tilde), 402
- | (Logic), 209
  
- abbreviate, 5, 1193
- ability.cov, 435, 922
- abline, 551, 586, 642
- abs, 6, 338
- absolute.size, 667, 733
- acf, 812, 1067
- acf2AR, 814
- acos, 173
- acos (Trig), 409
- acosh (Hyperbolic), 173
- adapt, 969
- add.scope (factor.scope), 923
- add1, 815, 824, 919, 923, 1174, 1176
- addGrob, 675, 681, 692, 694
- addGrob (grid.add), 680
- addmargins, 817
- addTaskCallback, 395, 397–399
- addTaskCallback (taskCallback),  
395
- aggregate, 15, 320, 394, 819
- agnes, 848, 883
- agrep, 7, 168, 1291
- AIC, 821, 919, 1041
- airmiles, 436
- AirPassengers, 437, 1183, 1211
- airquality, 438
- alias, 822, 831, 874
- alist, 24, 38, 143
- alist (list), 202
  
- all, 8, 10, 12, 210, 360
- all.equal, 9, 61, 175
- all.equal.POSIXct  
(DateTimeClasses), 85
- all.names, 11
- all.vars, 937
- all.vars (all.names), 11
- allGenerics (GenericFunctions),  
752
- anova, 373, 817, 824, 826, 827, 948, 950,  
981, 989, 1007, 1083, 1173
- anova-class (setOldClass), 798
- anova.glm, 825, 948, 950, 953, 1173
- anova.glm-class (setOldClass), 798
- anova.glm.null-class  
(setOldClass), 798
- anova.glmmlist (anova.glm), 825
- anova.lm, 826, 990, 995, 1173
- anova.lmlist (anova.lm), 826
- anova.mlm (anova.lm), 826
- ansari.test, 828, 853, 935, 1029, 1220
- anscombe, 439
- any, 9, 12, 210
- ANY-class (BasicClasses), 742
- aov, 252, 631, 815, 817, 830, 880, 881, 910,  
913, 914, 919, 923, 988, 990, 995,  
1013, 1014, 1020, 1026, 1115, 1116,  
1174, 1184, 1190, 1202, 1213, 1214
- aov-class (setOldClass), 798
- aperm, 13, 21, 390, 1133
- append, 14
- apply, 15, 57, 58, 109, 191, 222, 376, 394,  
820
- applyEdit (gEdit), 672
- applyEdits (gEdit), 672
- approx, 139, 1161
- approx (approxfun), 16
- approxfun, 16, 912, 1076, 1161, 1177
- apropos, 168, 213, 309, 1255, 1292
- ar, 832, 837, 840, 845, 1154, 1155
- ar.burg (ar), 832
- ar.mle (ar), 832
- ar.ols, 833, 835, 835
- ar.yw, 814
- ar.yw (ar), 832
- Arg (complex), 61
- args, 18, 38, 143, 153, 238, 1298, 1326, 1327

- arima, [837](#), [841](#), [842](#), [845–847](#), [976](#), [1094](#),  
[1095](#), [1182](#), [1209](#)
- arima.sim, [840](#), [841](#), [929](#)
- arima0, [835](#), [839](#), [840](#), [842](#)
- Arith(*groupGeneric*), [169](#)
- Arithmetic, [7](#), [19](#), [181](#), [209](#), [224](#), [349](#), [379](#),  
[1082](#)
- ARMAacf, [814](#), [846](#), [847](#)
- ARMAtoMA, [846](#), [847](#)
- array, [20](#), [102](#), [106](#), [123](#), [243](#), [394](#), [425](#)
- array-class (*StructureClasses*),  
[806](#)
- arrows, [552](#), [649](#)
- arrowsGrob (*grid.arrows*), [681](#)
- as, [56](#), [738](#), [746](#), [762](#), [784](#)
- as.array (*array*), [20](#)
- as.call (*call*), [43](#)
- as.character, [146](#), [179](#), [239](#), [261](#), [1194](#)
- as.character (*character*), [49](#)
- as.character.condition  
(*conditions*), [63](#)
- as.character.Date (*format.Date*),  
[146](#)
- as.character.error (*conditions*),  
[63](#)
- as.character.octmode (*octmode*),  
[249](#)
- as.character.package\_version  
(*package-version*), [258](#)
- as.character.person (*person*), [1308](#)
- as.character.personList (*person*),  
[1308](#)
- as.character.POSIXt (*strptime*),  
[361](#)
- as.complex (*complex*), [61](#)
- as.data.frame, [21](#), [26](#), [82](#), [126](#)
- as.data.frame.Date (*Dates*), [85](#)
- as.data.frame.package\_version  
(*package-version*), [258](#)
- as.data.frame.POSIXct  
(*DateTimeClasses*), [85](#)
- as.data.frame.POSIXlt  
(*DateTimeClasses*), [85](#)
- as.data.frame.table, [1232](#)
- as.data.frame.table (*table*), [390](#)
- as.Date (*format.Date*), [146](#)
- as.dendrogram, [895](#), [1060](#)
- as.dendrogram (*dendrogram*), [895](#)
- as.difftime (*difftime*), [99](#)
- as.dist (*dist*), [907](#)
- as.double, [248](#)
- as.double (*double*), [103](#)
- as.environment, [23](#), [750](#)
- as.expression (*expression*), [120](#)
- as.factor (*factor*), [129](#)
- as.formula (*formula*), [935](#)
- as.function, [23](#)
- as.hclust, [848](#), [883](#)
- as.integer, [74](#), [122](#), [124](#), [315](#)
- as.integer (*integer*), [176](#)
- as.list, [415](#)
- as.list (*list*), [202](#)
- as.logical (*logical*), [211](#)
- as.matrix, [84](#), [390](#)
- as.matrix (*matrix*), [224](#)
- as.matrix.dist (*dist*), [907](#)
- as.matrix.noquote (*noquote*), [240](#)
- as.matrix.POSIXlt  
(*DateTimeClasses*), [85](#)
- as.name (*name*), [236](#)
- as.null (*NULL*), [247](#)
- as.numeric (*numeric*), [248](#)
- as.ordered (*factor*), [129](#)
- as.package\_version  
(*package-version*), [258](#)
- as.pairlist (*list*), [202](#)
- as.person (*person*), [1308](#)
- as.personList (*person*), [1308](#)
- as.POSIX\*, [24](#)
- as.POSIXct, [87](#), [361](#)
- as.POSIXct (*as.POSIX\**), [24](#)
- as.POSIXlt, [87](#), [207](#), [424](#)
- as.POSIXlt (*as.POSIX\**), [24](#)
- as.qr (*qr*), [276](#)
- as.raw (*raw*), [291](#)
- as.real (*real*), [303](#)
- as.single, [141](#)
- as.single (*double*), [103](#)
- as.stepfun, [974](#)
- as.stepfun (*stepfun*), [1176](#)
- as.symbol (*name*), [236](#)
- as.table (*table*), [390](#)
- as.table.ftable (*read.ftable*),  
[1125](#)
- as.ts (*ts*), [1204](#)

- as.vector, 42, 49, 62, 74, 104, 177, 179, 211
- as.vector (vector), 420
- as<- (as), 738
- asin, 173
- asin (Trig), 409
- asinh (Hyperbolic), 173
- AsIs, 25
- asOneSidedFormula, 849
- assign, 26, 29, 157, 509
- assignInNamespace (getFromNamespace), 1284
- assignOps, 28
- assocplot, 554, 607
- atan, 173
- atan (Trig), 409
- atan2 (Trig), 409
- atanh (Hyperbolic), 173
- attach, 27, 29, 96, 199, 328, 329, 427, 1262
- attachNamespace (ns-load), 245
- attenu, 440
- attitude, 441
- attr, 30, 31, 60, 252, 1136
- attr.all.equal (all.equal), 9
- attr<- (attr), 30
- attributes, 10, 31, 31, 59, 60, 102, 128, 234, 509, 894
- attributes<- (attributes), 31
- austres, 442
- autoload, 32, 199
- autoloader (autoload), 32
- ave, 849
- axis, 145, 519, 530, 532, 555, 557–559, 561, 569, 615, 645
- axis.Date (axis.POSIXct), 557
- axis.POSIXct, 557, 590
- axTicks, 266, 556, 558, 586, 619
- backsolve, 33, 53, 344
- bandwidth, 850
- bandwidth.kernel (kernel), 978
- bandwidth.nrd, 851
- barplot, 559, 598, 633, 644
- bartlett.test, 830, 852, 935, 1029, 1220
- base-deprecated, 94
- base-deprecated, 34
- basename, 35, 137, 262
- BasicClasses, 742
- BATCH, 59, 518, 1256
- bcv, 851
- beaver1 (beavers), 442
- beaver2 (beavers), 442
- beavers, 442
- Bessel, 36, 349
- bessel (Bessel), 36
- besselI (Bessel), 36
- besselJ (Bessel), 36
- besselK (Bessel), 36
- besselY (Bessel), 36
- Beta, 853
- beta, 37, 854
- beta (Special), 348
- binom.test, 855, 1118
- Binomial, 856
- binomial, 950
- binomial (family), 924
- biplot, 858, 860, 1106
- biplot.default, 860
- biplot.prcomp, 1093
- biplot.prcomp (biplot.princomp), 859
- biplot.princomp, 859, 859, 1107
- birthday, 860
- bitmap, 517, 534
- bitmap (dev2bitmap), 516
- BJsales, 444
- BOD, 444
- body, 38, 143, 153
- body<- (body), 38
- body<- ,MethodDefinition-method (MethodsList-class), 775
- box, 562, 569, 581, 637, 642, 644, 651
- Box.test, 862, 1209
- boxplot, 563, 566, 568, 569, 633, 634, 653
- boxplot.stats, 564, 565, 566, 933, 1124
- bquote, 39, 370, 532
- break (Control), 74
- browseEnv, 309, 1257
- browser, 40, 90, 404–406, 1317, 1318
- browseURL, 1259, 1288, 1293
- bs, 1013
- bug.report, 251, 1259
- build (PkgUtils), 1309
- buildVignettes, 1233
- builtins, 40
- bw.bcv (bandwidth), 850
- bw.nrd, 899, 901

- bw.nrd (*bandwidth*), 850
- bw.nrd0 (*bandwidth*), 850
- bw.SJ (*bandwidth*), 850
- bw.ucv (*bandwidth*), 850
- bxp, 565, 568, 568
- by, 41, 231, 394
- bzfile, 44
- bzfile (*connections*), 67
  
- C, 131, 863, 880, 881
- c, 42, 48, 87, 179, 203, 240, 415, 421
- c.Date (*Dates*), 85
- c.noquote (*noquote*), 240
- c.package\_version  
  (*package-version*), 258
- c.POSIXct (*DateTimeClasses*), 85
- c.POSIXlt (*DateTimeClasses*), 85
- call, 23, 43, 103, 121, 183, 221, 222, 234,  
  236, 304, 597, 902, 903, 908, 974
- call-class (*language-class*), 768
- callGeneric (*GenericFunctions*),  
  752
- callNextMethod, 743, 772, 776
- cancor, 864
- capabilities, 44, 70, 114, 375, 518, 534
- capture.output, 340, 401, 1262
- cars, 445, 1013
- case.names, 319
- case.names (*case/variable.names*),  
  865
- case/variable.names, 865
- casefold (*chartr*), 51
- cat, 45, 261, 269, 423, 431, 951, 1323
- Cauchy, 866
- cbind, 46, 231, 1208
- cbind.ts (*ts*), 1204
- ccf (*acf*), 812
- ceiling (*Round*), 314
- char.expand, 48
- character, 49, 130, 214, 240, 270, 281,  
  387, 640, 661, 662, 1327
- character-class (*BasicClasses*),  
  742
- charmatch, 49, 50, 168, 220, 263
- charToRaw (*rawConversion*), 292
- chartr, 50, 51, 168
- check, 1280
- check (*PkgUtils*), 1309
- check.options, 509, 536, 538
- checkDocFiles (*QC*), 1246
- checkDocStyle (*QC*), 1246
- checkFF, 1234
- checkMD5sums, 1235, 1245
- checkReplaceFuns (*QC*), 1246
- checkS3methods (*QC*), 1246
- checkTnF, 1235
- checkVignettes, 1236
- ChickWeight, 446
- chickwts, 447
- childNames (*grid.grob*), 696
- chisq.test, 391, 459, 555, 868, 931, 1232
- Chisquare, 870, 926
- chol, 34, 52, 54, 114
- chol2inv, 53, 54, 344
- choose (*Special*), 348
- chron, 25, 147
- chull, 571
- circleGrob (*grid.circle*), 684
- CITATION, 1263
- CITATION (*citEntry*), 1264
- citation, 1263
- citEntry, 1263, 1264, 1333
- citFooter (*citEntry*), 1264
- citHeader (*citEntry*), 1264
- class, 55, 81, 170, 184, 213, 241, 268, 372,  
  391, 418, 586, 634, 823, 989, 1093,  
  1302
- class<- (*class*), 55
- Classes, 745, 747, 757, 778, 805
- classRepresentation-class, 770,  
  784
- classRepresentation-class, 746,  
  790
- ClassUnionRepresentation-class  
  (*setClassUnion*), 789
- clearNames, 872, 1144
- close (*connections*), 67
- close.screen (*screen*), 646
- close.socket, 1266, 1300, 1316
- closeAllConnections  
  (*showConnections*), 336
- cm (*units*), 546
- cm.colors (*Palettes*), 525
- cmdscale, 872
- co.intervals (*coplot*), 575
- CO2, 448
- co2, 449

- codoc, [1237](#), [1253](#)
- codocClasses (*codoc*), [1237](#)
- codocData (*codoc*), [1237](#)
- coef, [839](#), [874](#), [915](#), [953](#), [990](#), [995](#), [1041](#), [1189](#)
- coefficients, [824](#), [933](#), [949](#), [1135](#)
- coefficients (*coef*), [874](#)
- coerce (*as*), [738](#)
- coerce, ANY, array-method (*as*), [738](#)
- coerce, ANY, call-method (*as*), [738](#)
- coerce, ANY, character-method (*as*), [738](#)
- coerce, ANY, complex-method (*as*), [738](#)
- coerce, ANY, environment-method (*as*), [738](#)
- coerce, ANY, expression-method (*as*), [738](#)
- coerce, ANY, function-method (*as*), [738](#)
- coerce, ANY, integer-method (*as*), [738](#)
- coerce, ANY, list-method (*as*), [738](#)
- coerce, ANY, logical-method (*as*), [738](#)
- coerce, ANY, matrix-method (*as*), [738](#)
- coerce, ANY, name-method (*as*), [738](#)
- coerce, ANY, NULL-method (*as*), [738](#)
- coerce, ANY, numeric-method (*as*), [738](#)
- coerce, ANY, single-method (*as*), [738](#)
- coerce, ANY, ts-method (*as*), [738](#)
- coerce, ANY, vector-method (*as*), [738](#)
- coerce-methods (*as*), [738](#)
- coerce<- (*as*), [738](#)
- col, [56](#), [317](#), [331](#), [341](#)
- col2rgb, [510](#), [512](#), [524](#), [525](#), [544](#), [545](#)
- colMeans (*colSums*), [57](#)
- colnames, [102](#)
- colnames (*row/colnames*), [319](#)
- colnames<- (*row/colnames*), [319](#)
- colors, [510](#), [511](#), [512](#), [524](#), [525](#), [620](#), [621](#), [639](#), [911](#)
- colours (*colors*), [512](#)
- colSums, [57](#)
- commandArgs, [58](#)
- comment, [59](#)
- comment<- (*comment*), [59](#)
- Compare (*groupGeneric*), [169](#)
- compareVersion, [258](#), [1266](#)
- Comparison, [60](#), [175](#), [291](#), [345](#), [379](#)
- COMPILE, [1267](#), [1326](#)
- complete.cases, [235](#), [875](#)
- Complex, [62](#)
- Complex (*groupGeneric*), [169](#)
- complex, [6](#), [61](#), [264](#)
- complex-class (*BasicClasses*), [742](#)
- computeRestarts (*conditions*), [63](#)
- conditionCall (*conditions*), [63](#)
- conditionMessage (*conditions*), [63](#)
- conditions, [63](#)
- confint, [876](#)
- confint.nls, [876](#)
- conflicts, [66](#), [197](#)
- Conj (*complex*), [61](#)
- connection, [76](#), [295](#), [326](#), [1250](#), [1314](#)
- connection (*connections*), [67](#)
- connections, [67](#), [253](#), [275](#), [299](#), [302](#), [330](#), [337](#), [401](#), [431](#)
- Constants, [72](#)
- constrOptim, [877](#), [1056](#)
- contour, [519](#), [521](#), [523](#), [572](#), [582](#), [593](#), [624](#)
- contourLines (*contour*), [572](#)
- contr.helmert, [881](#)
- contr.helmert (*contrast*), [879](#)
- contr.poly, [881](#), [1081](#)
- contr.poly (*contrast*), [879](#)
- contr.sum, [863](#), [881](#)
- contr.sum (*contrast*), [879](#)
- contr.treatment, [880](#), [881](#), [1128](#)
- contr.treatment (*contrast*), [879](#)
- contrast, [879](#)
- contrasts, [127](#), [252](#), [863](#), [880](#), [880](#), [1024](#), [1141](#)
- contrasts<- (*contrasts*), [880](#)
- contrib.url (*update.packages*), [1333](#)
- contributors, [73](#), [75](#)
- Control, [74](#), [379](#)
- convertHeight (*grid.convert*), [686](#)
- convertNative, [668](#)
- convertUnit (*grid.convert*), [686](#)
- convertWidth (*grid.convert*), [686](#)
- convertX (*grid.convert*), [686](#)
- convertY (*grid.convert*), [686](#)
- convolve, [881](#), [928](#), [929](#), [977](#), [1037](#)

- cooks.distance, 994, 1071
- cooks.distance
  - (influence.measures), 965
- cophenetic, 882, 1129
- coplot, 575, 614, 615, 647, 937
- copyright, 75
- copyrights (copyright), 75
- cor, 884, 1093, 1106, 1107
- cor.test, 885, 886
- cos, 173
- cos (Trig), 409
- cosh (Hyperbolic), 173
- count.fields, 75, 297
- cov, 889, 1011, 1093, 1107
- cov (cor), 884
- cov.mcd, 1106
- cov.mve, 1106
- cov.wt, 885, 889, 920, 1106
- cov2cor (cor), 884
- covratio, 994
- covratio (influence.measures), 965
- coxph, 1199, 1202
- cpgram, 890, 1158
- CRAN.packages, 1246, 1275, 1276, 1307
- CRAN.packages (update.packages), 1333
- crossprod, 76
- cummax (cumsum), 77
- cummin (cumsum), 77
- cumprod, 274
- cumprod (cumsum), 77
- cumsum, 77, 274
- current.transform (Querying the Viewport Tree), 724
- current.viewport (Querying the Viewport Tree), 724
- current.vpTree (Querying the Viewport Tree), 724
- curve, 578
- cut, 78, 80, 351, 593
- cut.Date, 85
- cut.Date (cut.POSIXt), 80
- cut.dendrogram (dendrogram), 895
- cut.POSIXt, 80, 87
- cutree, 891, 956
- cycle (time), 1203
- D (deriv), 902
- daisy, 908
- data, 73, 199, 324, 1268, 1289
- data.class, 81
- data.entry, 1277, 1279
- data.entry (dataentry), 1270
- data.frame, 22, 26, 47, 48, 59, 82, 84, 96, 101, 102, 126, 170, 214, 225, 231, 270, 297, 318, 328, 389, 409, 416, 627, 628, 1021, 1023, 1314, 1315
- data.frame-class (setOldClass), 798
- data.matrix, 83, 225
- dataentry, 253, 1270
- dataViewport, 669, 723
- Date, 98, 147, 148, 316, 333, 386, 424
- Date (Dates), 85
- date, 25, 84, 147, 207, 386, 1203
- Dates, 85, 87, 558
- DateTimeClasses, 25, 85, 85, 100, 134, 316, 334, 363, 386, 424, 558
- dbeta, 944
- dbeta (Beta), 853
- dbinom, 1036, 1079
- dbinom (Binomial), 856
- dcauchy (Cauchy), 866
- dcf, 88
- dchisq, 927, 944
- dchisq (Chisquare), 870
- de (dataentry), 1270
- debug, 40, 89, 153
- debugger, 1272
- decompose, 892
- Defunct, 90, 94, 242, 1336
- delay, 32, 90, 93, 108, 369, 1327
- delete.response, 893
- delimMatch, 1239
- deltat, 985
- deltat (time), 1203
- demo, 348, 1274, 1281
- dendrapply, 894
- dendrogram, 313, 891, 894, 895, 956, 957, 1060
- density, 589, 627, 636, 657, 850, 851, 899, 1067
- density-class (setOldClass), 798
- deparse, 50, 91, 93, 106, 107, 239, 260, 369, 659
- deparseOpts, 93
- Deprecated, 34, 90, 94, 242, 1252, 1336

- deriv, 902, 1038, 1039
- deriv3 (*deriv*), 902
- det, 95, 114, 277, 278
- detach, 30, 96, 197–199, 246, 329, 419
- determinant (*det*), 95
- dev.control (*dev2*), 514
- dev.copy (*dev2*), 514
- dev.copy2eps (*dev2*), 514
- dev.cur, 516, 518
- dev.cur (*dev.xxx*), 513
- dev.interactive, 512, 518
- dev.list (*dev.xxx*), 513
- dev.next (*dev.xxx*), 513
- dev.off (*dev.xxx*), 513
- dev.prev (*dev.xxx*), 513
- dev.print, 518, 534
- dev.print (*dev2*), 514
- dev.set (*dev.xxx*), 513
- dev.xxx, 513
- dev2, 514
- dev2bitmap, 516, 518
- deviance, 904, 905, 919, 953, 995
- device (*Devices*), 517
- Devices, 513, 514, 517, 528, 529, 534, 538, 541, 548, 550, 647
- dexp, 1223
- dexp (*Exponential*), 917
- df, 1198
- df (*FDist*), 926
- df.kernel (*kernel*), 978
- df.residual, 905, 905, 953, 995
- dfbeta (*influence.measures*), 965
- dfbetas, 994
- dfbetas (*influence.measures*), 965
- dffits, 994
- dffits (*influence.measures*), 965
- dgamma, 854, 871, 918
- dgamma (*GammaDist*), 943
- dgeom, 1036
- dgeom (*Geometric*), 945
- dget, 108
- dget (*dput*), 105
- dhyper (*Hypergeometric*), 962
- diag, 97, 212, 224
- diag<- (*diag*), 97
- diana, 848
- diff, 98, 906, 985, 1207
- diff.ts, 99
- diff.ts (*ts-methods*), 1206
- diffinv, 99, 906
- difftime, 85, 87, 99, 170, 332, 333
- digamma (*Special*), 348
- dim, 21, 31, 101, 128, 179, 225, 243, 394, 425
- dim<-, 179
- dim<- (*dim*), 101
- dimnames, 21, 31, 101, 102, 179, 225, 271, 319, 416, 637
- dimnames<-, 179
- dimnames<- (*dimnames*), 102
- dir (*list.files*), 203
- dir.create (*files*), 136
- dirname (*basename*), 35
- discoveries, 450
- dist, 874, 883, 907, 908, 957
- dlnorm, 1049
- dlnorm (*Lognormal*), 1004
- dlogis (*Logistic*), 999
- dmultinom (*Multinomial*), 1029
- DNase, 451
- dnbinom, 857, 946, 1079
- dnbinom (*NegBinomial*), 1035
- dnorm, 1005
- dnorm (*Normal*), 1048
- do.call, 43, 103, 304
- Documentation, 747
- Documentation-class (*Documentation*), 747
- Documentation-methods (*Documentation*), 747
- dotchart, 562, 579, 626
- double, 103, 128, 177, 665
- double-class (*BasicClasses*), 742
- download.file, 44, 69, 205, 253, 1275, 1333, 1335
- download.packages, 1276
- download.packages (*update.packages*), 1333
- downViewport, 679, 733
- downViewport (*Working with Viewports*), 734
- dpois, 857, 1036
- dpois (*Poisson*), 1078
- dput, 93, 105, 108, 324, 1308, 1326
- dQuote, 370
- dQuote (*sQuote*), 353
- draw.details (*drawDetails*), 670

- drawDetails, 670
- drop, 106, 224
- drop.scope (*Factor.scope*), 923
- drop.terms (*delete.response*), 893
- drop1, 106, 824, 826, 827, 919, 923, 1174, 1176
- drop1 (*add1*), 815
- dsignrank, 1229
- dsignrank (*SignRank*), 1145
- dt, 867, 927
- dt (*TDist*), 1197
- dummy.coef, 909
- dump, 93, 106, 107, 324
- dump.frames, 252, 1317, 1318
- dump.frames (*debugger*), 1272
- dump.frames-class (*setOldClass*), 798
- dumpMethod (*GenericFunctions*), 752
- dumpMethods (*GenericFunctions*), 752
- dunif (*Uniform*), 1214
- duplicated, 108, 413
- dweibull, 918
- dweibull (*Weibull*), 1222
- dwilcox, 1146
- dwilcox (*Wilcoxon*), 1228
- dyn.load, 109, 141, 143, 162, 200, 201, 1268, 1326
- dyn.unload (*dyn.load*), 109
  
- eapply, 112
- ecdf, 139, 911, 1076, 1124, 1177
- edit, 251, 404, 405, 1271, 1276, 1279, 1282, 1284, 1308
- edit.data.frame, 1277, 1278, 1282
- edit.matrix (*edit.data.frame*), 1278
- editDetails, 671
- editGrob, 672, 675
- editGrob (*grid.edit*), 691
- eff.aovlist, 913
- effects, 824, 914, 950, 953, 990, 995
- eigen, 113, 277, 278, 375, 1093, 1106, 1107
- else (*Control*), 74
- emacs (*edit*), 1276
- embed, 915
- end, 1206
- end (start), 1172
  
- engine.display.list (*grid.display.list*), 689
- environment, 23, 27, 29, 30, 91, 93, 107, 114, 116, 117, 119, 157, 213, 309, 381, 509, 1177, 1258, 1268, 1272
- environment-class, 749
- environment<- (*environment*), 114
- erase.screen (*screen*), 646
- Error (*aov*), 830
- esoph, 452, 950
- euro, 453
- eurodist, 454
- EuStockMarkets, 454
- eval, 116, 116, 121, 192, 260, 348, 369, 382
- evalq, 427
- evalq (*eval*), 116
- example, 1279, 1294
- exists, 27, 118, 157, 1298
- existsMethod (*getMethod*), 757
- exp, 918
- exp (*log*), 208
- expand.grid, 120, 1101
- expand.model.frame, 916, 1023
- expml (*log*), 208
- Exponential, 917
- expression, 43, 92, 116, 117, 120, 183, 369, 597, 655, 661, 662, 902, 903
- expression-class (*BasicClasses*), 742
- extends, 790, 796
- extends (*is*), 764
- externalptr-class (*BasicClasses*), 742
- Extract, 121, 126, 127, 342, 379
- Extract.data.frame, 124
- Extract.factor, 127
- extractAIC, 817, 822, 905, 918, 1174, 1175
- Extremes, 128
  
- F (*logical*), 211
- factanal, 919, 996, 1221
- factanal.fit.mle (*factanal*), 919
- factor, 78, 79, 122, 127, 129, 166, 170, 178, 194, 211, 240, 331, 373, 393, 576, 631, 633, 948, 1024, 1128
- factor-class (*setOldClass*), 798
- factor.scope, 923
- factorial (*Special*), 348
- faithful, 455

- FALSE (*logical*), 211  
 family, 924, 947, 949, 1001, 1011, 1012, 1082  
 family.glm (*glm.summaries*), 952  
 family.lm (*lm.summaries*), 994  
 fdeaths (*UKLungDeaths*), 498  
 FDist, 926  
 fft, 881, 882, 900, 927, 1037, 1156  
 fifo (*connections*), 67  
 file, 298, 302, 401, 431  
 file (*connections*), 67  
 file.access, 131, 134, 137, 204  
 file.append (*files*), 136  
 file.choose, 133, 204  
 file.copy (*files*), 136  
 file.create (*files*), 136  
 file.edit, 1281  
 file.exists, 1241  
 file.exists (*files*), 136  
 file.info, 132, 133, 137, 204, 249, 1241  
 file.path, 35, 135, 137, 1241  
 file.remove, 414  
 file.remove (*files*), 136  
 file.rename (*files*), 136  
 file.show, 135, 137, 251, 1282, 1288, 1308, 1335  
 file.symlink (*files*), 136  
 file\_path\_as\_absolute (*fileutils*), 1240  
 file\_path\_sans\_ext (*fileutils*), 1240  
 file\_test (*fileutils*), 1240  
 filePathAsAbsolute (*tools-deprecated*), 1251  
 filePathSansExt (*tools-deprecated*), 1251  
 files, 134, 136, 136, 204, 1282  
 fileTest (*tools-deprecated*), 1251  
 fileutils, 1240  
 filled.contour, 503, 574, 580, 593  
 filter, 846, 882, 928, 977  
 find, 213, 754  
 find (*apropos*), 1255  
 findClass (*setClass*), 785  
 findFunction (*GenericFunctions*), 752  
 findInterval, 79, 138  
 findMethod (*getMethod*), 757  
 findRestart (*conditions*), 63  
 fisher.test, 930  
 fitted, 932, 953, 990, 995, 1041  
 fitted.values, 824, 875, 950, 1135  
 fivenum, 568, 933, 972, 1124  
 fix, 1277, 1282, 1282, 1285, 1308  
 fixInNamespace (*getFromNamespace*), 1284  
 fixPre1.8, 750  
 fligner.test, 830, 853, 934, 1029  
 floor (*Round*), 314  
 flush (*connections*), 67  
 for, 1311  
 for (*Control*), 74  
 for-class (*language-class*), 768  
 force, 117, 139  
 Foreign, 140, 1234  
 Formaldehyde, 456  
 formals, 18, 19, 143, 202, 203, 238  
 formals<- (*formals*), 143  
 format, 46, 144, 148–151, 225, 268–270, 373, 403, 907, 1110  
 format.char (*formatC*), 149  
 format.Date, 85, 146  
 format.dist (*dist*), 907  
 format.info, 146, 148  
 format.octmode (*octmode*), 249  
 format.POSIXct, 25  
 format.POSIXct (*strptime*), 361  
 format.POSIXlt, 25  
 format.POSIXlt (*strptime*), 361  
 format.pval, 1110  
 formatC, 145, 146, 149, 149, 353  
 formatDL, 151, 1250  
 formula, 26, 115, 402, 631, 634, 849, 903, 935, 938, 989, 1022, 1023, 1041, 1201–1203  
 formula-class (*setOldClass*), 798  
 formula.lm (*lm.summaries*), 994  
 formula.nls, 937  
 forwardsolve (*backsolve*), 33  
 fourfoldplot, 583  
 frame, 585, 1308  
 frameGrob (*grid.frame*), 692  
 freeny, 457  
 frequency, 1206  
 frequency (*time*), 1203  
 friedman.test, 938, 1122

- ftable, 391, 818, 940, 942, 943, 1126
- ftable.default, 942, 943
- ftable.formula, 941, 942
- function, 24, 38, 43, 115, 121, 143, 153, 180, 576, 627
- function-class (*BasicClasses*), 742
- functionWithTrace-class (*TraceClasses*), 806
- fuzzy matching, 1290
- fuzzy matching (*agrep*), 7
- Gamma, 1001
- Gamma (*family*), 924
- gamma, 36, 37, 944
- gamma (*Special*), 348
- gammaCody (*Bessel*), 36
- GammaDist, 943
- gaussian, 1001
- gaussian (*family*), 924
- gc, 154, 155, 228, 230, 305, 389
- gc.time, 155, 273
- gcinfo, 228
- gcinfo (*gc*), 154
- gctorture, 154, 156
- gEdit, 672
- gEditList (*gEdit*), 672
- genericFunction-class, 751
- GenericFunctions, 752, 770, 803
- genericFunctionWithTrace-class (*TraceClasses*), 806
- Geometric, 945
- get, 27, 119, 156, 223, 265, 509, 787, 1283–1285, 1298
- get.gpar (*gpar*), 673
- getAllConnections (*showConnections*), 336
- getAnywhere, 1283, 1302
- getCallingDLL, 158
- getCConverterDescriptions (*getNumCConverters*), 163
- getCConverterStatus (*getNumCConverters*), 163
- getClass, 746, 756, 805
- getClassDef, 746
- getClassDef (*getClass*), 756
- getClasses (*setClass*), 785
- getConnection (*showConnections*), 336
- getDepList, 1241
- getDLLRegisteredRoutines, 159
- geterrmessage, 410, 1272
- geterrmessage (*stop*), 358
- getFromNamespace, 1283, 1284
- getGeneric, 753
- getGenerics, 779, 780
- getGenerics (*GenericFunctions*), 752
- getGrob, 675, 681, 692, 694, 713
- getGrob (*grid.get*), 693
- getHook (*UserHooks*), 418
- getInitial, 946
- getLoadedDLLs, 111, 159, 160, 160
- getMethod, 757, 1289
- getMethods (*getMethod*), 757
- getMethodsMetaData, 759
- getNativeSymbolInfo, 161, 161
- getNumCConverters, 163
- getOption (*options*), 250
- getPackageName, 760, 770
- getpid, 164
- getRversion, 282
- getRversion (*package-version*), 258
- getS3method, 418, 1285, 1285, 1302
- getTaskCallbackNames, 395, 396, 398
- getTaskCallbackNames (*taskCallbackNames*), 398
- getValidity, 808
- getwd, 165, 204, 295, 326, 379
- gl, 131, 166, 334
- gList (*grid.grob*), 696
- glm, 373, 815, 816, 825, 826, 875, 880, 881, 905, 924, 925, 933, 936, 937, 947, 951–953, 966, 973, 990, 995, 1011, 1012, 1021, 1033, 1051, 1070, 1097, 1135, 1174, 1175, 1186, 1187, 1199, 1201, 1222, 1225
- glm-class (*setOldClass*), 798
- glm.control, 948, 951
- glm.fit, 951, 952
- glm.null-class (*setOldClass*), 798
- glm.summaries, 952
- globalenv (*environment*), 114
- gpar, 673
- gPath, 675, 705, 706
- graphics.off, 518
- graphics.off (*dev.xxx*), 513
- gray, 512, 518, 522, 524, 525, 544, 621

- grep, 8, 44, 50, 51, 167, 213, 263, 305, 309, 365, 1291
- grey (*gray*), 518
- Grid, 676, 679, 682, 685, 696, 699, 700, 703, 708, 709, 712, 714, 716–718, 720, 721
- grid, 585, 1069
- Grid Viewports, 676
- grid.add, 680
- grid.arrows, 681
- grid.circle, 684
- grid.collection, 685
- grid.convert, 668, 686
- grid.convertHeight (*grid.convert*), 686
- grid.convertWidth (*grid.convert*), 686
- grid.convertX (*grid.convert*), 686
- grid.convertY (*grid.convert*), 686
- grid.copy, 688
- grid.display.list, 689
- grid.draw, 671, 690, 697
- grid.edit, 671, 691, 697, 705, 706, 732
- grid.frame, 692, 705, 706
- grid.get, 693, 697
- grid.grab, 694
- grid.grill, 695
- grid.grob, 685, 688, 696, 715
- grid.layout, 676, 679, 697, 716
- grid.line.to, 682
- grid.line.to (*grid.move.to*), 702
- grid.lines, 682, 699
- grid.locator, 700
- grid.move.to, 702
- grid.newpage, 703, 710
- grid.pack, 693, 704, 706
- grid.place, 705, 705
- grid.plot.and.legend, 706
- grid.points, 707
- grid.polygon, 708
- grid.pretty, 709
- grid.prompt, 710
- grid.rect, 711
- grid.remove, 712
- grid.segments, 682, 713
- grid.set, 714
- grid.show.layout, 679, 699, 715
- grid.show.viewport, 716
- grid.text, 717
- grid.xaxis, 719, 721
- grid.yaxis, 720, 720
- grob, 672, 675, 681, 690, 692, 694, 713
- grob (*grid.grob*), 696
- grobHeight (*grobWidth*), 722
- grobWidth, 722, 726
- group generic, 56
- group generic (*groupGeneric*), 169
- groupGeneric, 169
- groupGenericFunction-class (*genericFunction-class*), 751
- groupGenericFunctionWithTrace-class (*TraceClasses*), 806
- gsub, 51
- gsub (*grep*), 167
- gTree, 695
- gTree (*grid.grob*), 696
- gzcon, 172
- gzfile, 44, 172
- gzfile (*connections*), 67
- HairEyeColor, 458
- Harman23.cor, 459, 922
- Harman74.cor, 459, 922, 1221
- hasArg, 761
- hasMethod (*getMethod*), 757
- hasTsp (*tsp*), 1210
- hat, 994, 1007, 1071
- hat (*influence.measures*), 965
- hatvalues (*influence.measures*), 965
- hclust, 848, 883, 891, 908, 953, 957, 959, 964, 1127
- head, 1286
- heat.colors, 512, 592, 593
- heat.colors (*Palettes*), 525
- heatmap, 593, 956, 1129
- heightDetails, 667
- heightDetails (*widthDetails*), 733
- help, 19, 136, 251, 748, 1269, 1280, 1287, 1292–1294, 1311
- help.search, 309, 1256, 1289, 1290
- help.start, 251, 1288, 1289, 1292, 1292, 1299
- Hershey, 519, 523, 574, 661, 674
- hist, 562, 586, 589, 590, 611, 635, 636, 638, 644, 901

- hist.Date, 85
- hist.Date (*hist.POSIXt*), 589
- hist.default, 590
- hist.POSIXt, 589
- history (*savehistory*), 1324
- HoltWinters, 959, 1069, 1098
- hsearch-class (*setOldClass*), 798
- hsv, 512, 519, 522, 524, 525, 544, 545, 593, 617
- Hyperbolic, 173
- Hypergeometric, 962
  
- I, 22, 82, 83, 936, 937
- I (*AsIs*), 25
- identical, 9, 10, 61, 174, 181
- identify, 591, 603, 964
- identify.hclust, 955, 956, 963, 1127
- if, 176, 210, 259
- if (*Control*), 74
- if-class (*language-class*), 768
- ifelse, 75, 175
- Im (*complex*), 61
- image, 517, 518, 574, 582, 592, 624, 638, 956, 958, 959
- index.search, 1293
- Indometh, 460
- Inf, 19, 139, 140, 933
- Inf (*is.finite*), 181
- infert, 461, 950
- influence, 966, 967, 995, 1071, 1225
- influence (*lm.influence*), 992
- influence.measures, 965, 993, 994
- inherits (*class*), 55
- initialize, 748, 763, 807
- initialize (*new*), 777
- initialize, ANY-method (*initialize-methods*), 762
- initialize, environment-method (*initialize-methods*), 762
- initialize, signature-method (*initialize-methods*), 762
- initialize, traceable-method (*initialize-methods*), 762
- initialize-methods, 777
- initialize-methods, 762
- InsectSprays, 462
- INSTALL, 198, 199, 761, 1294, 1310, 1319, 1326, 1334, 1335
- install.packages, 199, 1243, 1305, 1320
- install.packages (*update.packages*), 1333
- installed.packages, 197, 199, 1242, 1296, 1307, 1334, 1335
- installFoundDepends, 1242, 1243
- Insurance, 1051
- integer, 81, 101, 104, 128, 149, 176, 193, 242, 285, 426
- integer-class (*BasicClasses*), 742
- integrate, 968
- integrate-class (*setOldClass*), 798
- interaction, 177, 331
- interaction.plot, 632, 970
- interactive, 178, 252
- Internal, 179
- InternalMethods, 21, 43, 49, 62, 104, 115, 122, 130, 177, 179, 181–186, 193, 202, 211, 224, 236, 247, 248, 415, 1205, 1247
- interpSpline, 1161
- intersect (*sets*), 335
- intToBits (*rawConversion*), 292
- inverse.gaussian, 1001
- inverse.gaussian (*family*), 924
- inverse.rle (*rle*), 313
- invisible, 153, 180, 200, 268, 524, 964
- invokeRestart (*conditions*), 63
- invokeRestartInteractively (*conditions*), 63
- IQR, 933, 972, 1010
- iris, 463
- iris3 (*iris*), 463
- is, 746, 764, 784
- is.array, 179
- is.array (*array*), 20
- is.atomic, 179
- is.atomic (*is.recursive*), 185
- is.call, 179
- is.call (*call*), 43
- is.character, 179
- is.character (*character*), 49
- is.complex, 179
- is.complex (*complex*), 61
- is.data.frame (*as.data.frame*), 21
- is.double, 179
- is.double (*double*), 103

- is.element, 220
- is.element (sets), 335
- is.empty.model, 973
- is.environment, 179
- is.environment (environment), 114
- is.expression (expression), 120
- is.factor (factor), 129
- is.finite, 181
- is.function, 179, 182
- is.infinite (is.finite), 181
- is.integer, 179
- is.integer (integer), 176
- is.language, 43, 179, 183, 185, 236
- is.leaf (dendrogram), 895
- is.list, 179, 185, 421
- is.list (list), 202
- is.loaded, 161, 162
- is.loaded (dyn.load), 109
- is.logical, 179
- is.logical (logical), 211
- is.matrix, 179
- is.matrix (matrix), 224
- is.mts (ts), 1204
- is.na, 130, 179, 875
- is.na (NA), 234
- is.na.POSIXlt (DateTimeClasses), 85
- is.na<- (NA), 234
- is.na<- .factor (factor), 129
- is.name (name), 236
- is.nan, 179, 235
- is.nan (is.finite), 181
- is.null, 179
- is.null (NULL), 247
- is.numeric, 179, 421
- is.numeric (numeric), 248
- is.object, 179, 183
- is.ordered (factor), 129
- is.package\_version (package-version), 258
- is.pairlist, 179
- is.pairlist (list), 202
- is.qr (qr), 276
- is.R, 184
- is.real (real), 303
- is.recursive, 122, 179, 185
- is.single, 179, 186
- is.stepfun (stepfun), 1176
- is.symbol, 179
- is.symbol (name), 236
- is.table (table), 390
- is.ts (ts), 1204
- is.tskernel (kernel), 978
- is.unsorted (sort), 344
- is.vector (vector), 420
- isClass, 756, 757
- isClass (setClass), 785
- isClassUnion (setClassUnion), 789
- isGeneric (GenericFunctions), 752
- isGroup (GenericFunctions), 752
- isIncomplete, 401
- isIncomplete (connections), 67
- islands, 464
- ISOdate (strptime), 361
- ISOdatetime (strptime), 361
- ISOLatin1 (connections), 67
- isoMDS, 874, 974
- isOpen (connections), 67
- isoreg, 973, 1069, 1070
- isRestart (conditions), 63
- isSealedClass (isSealedMethod), 767
- isSealedMethod, 767
- isSeekable (seek), 329
- Japanese, 521, 523
- jitter, 186, 646, 657
- JohnsonJohnson, 465, 1211
- jpeg, 44, 517, 518, 1257
- jpeg (png), 533
- julian (weekdays), 423
- KalmanForecast, 1095
- KalmanForecast (KalmanLike), 975
- KalmanLike, 839, 975, 1183
- KalmanRun (KalmanLike), 975
- KalmanSmooth, 1211
- KalmanSmooth (KalmanLike), 975
- kappa, 187
- kernapply, 977, 979
- kernel, 977, 978
- kmeans, 956, 979
- knots, 1075, 1177
- knots (stepfun), 1176
- kronecker, 189, 257
- kruskal.test, 980, 1052, 1227
- ks.test, 982

- ksmooth, 984
- La.chol (*chol*), 52
- La.chol2inv (*chol2inv*), 54
- La.svd (*svd*), 374
- labels, 190, 898
- labels.dendrogram  
(*order.dendrogram*), 1060
- lag, 985
- lag.plot, 986
- LakeHuron, 465
- language-class, 768
- lapply, 15, 112, 191, 222, 394, 820, 895
- Last.value, 192
- layout, 514, 594, 610, 618, 621, 647, 699, 958
- lbeta (*Special*), 348
- lchoose (*Special*), 348
- lcm (*layout*), 594
- ldeaths (*UKLungDeaths*), 498
- legend, 121, 596, 644, 970
- length, 179, 193
- length<-, 179
- length<- (*length*), 193
- LETTERS (*Constants*), 72
- letters (*Constants*), 72
- levelplot, 582
- levels, 131, 194, 211, 240
- levels<- (*levels*), 194
- lfactorial (*Special*), 348
- lgamma (*Special*), 348
- lh, 466
- libPaths, 195
- library, 30, 32, 96, 196, 196, 201, 244, 252, 328, 385, 419, 433, 754, 761, 1267, 1289, 1295, 1296, 1335
- library.dynam, 111, 198, 199, 199, 1326
- libraryIQR-class (*setOldClass*), 798
- licence (*license*), 201
- license, 75, 201
- LifeCycleSavings, 466
- limitedLabels (*recover*), 1317
- line, 987
- linearizeMlist, 769, 770, 773
- LinearMethodsList-class, 769
- lines, 552, 578, 586, 600, 604, 614, 615, 624, 627, 636, 639, 640, 642, 649, 664, 1069, 1077, 1078, 1199
- lines.formula (*plot.formula*), 634
- lines.histogram (*plot.histogram*), 635
- lines.stepfun (*plot.stepfun*), 1075
- lines.ts (*plot.ts*), 1077
- linesGrob (*grid.lines*), 699
- lineToGrob (*grid.move.to*), 702
- LINK, 1297
- list, 91, 96, 123, 202, 235, 253, 277, 281, 394, 409, 1069
- list-class (*BasicClasses*), 742
- list.files, 133, 134, 136, 137, 165, 203, 309, 388, 1241
- list\_files\_with\_exts (*fileutils*), 1240
- list\_files\_with\_type (*fileutils*), 1240
- listFilesWithExts  
(*tools-deprecated*), 1251
- listFilesWithType  
(*tools-deprecated*), 1251
- listFromMlist, 773
- lm, 252, 355, 373, 415, 815–817, 826, 827, 831, 866, 875, 880, 881, 905, 914, 918, 919, 923, 933, 936, 937, 950, 966, 973, 981, 987, 988, 991–995, 1008, 1009, 1033, 1070, 1083, 1100, 1116, 1135, 1174, 1187, 1189, 1199, 1201, 1225
- lm-class (*setOldClass*), 798
- lm.fit, 989, 990, 991
- lm.influence, 966, 967, 990, 992, 1007, 1008, 1071, 1225
- lm.summaries, 994
- lm.wfit, 990
- lm.wfit (*lm.fit*), 991
- lme, 831
- load, 204, 324, 1268
- loadedNamespaces, 329
- loadedNamespaces (*ns-load*), 245
- loadhistory (*savehistory*), 1324
- loadings, 995, 1106
- loadNamespace, 244, 419
- loadNamespace (*ns-load*), 245
- loadURL (*load*), 204
- Loblolly, 467
- local, 356
- local (*eval*), 116

- localeconv, 206
- locales, 60, 148, 207, 306, 363
- locator, 592, 596, 602, 701
- loess, 996, 999, 1006, 1101, 1138, 1147, 1180
- loess.control, 997, 998, 998
- loess.smooth(*scatter.smooth*), 1137
- log, 7, 208
- log10(*log*), 208
- loglp(*log*), 208
- log2(*log*), 208
- logb(*log*), 208
- Logic, 209, 291, 379, 425
- logical, 210, 211, 360, 425
- logical-class(*BasicClasses*), 742
- Logistic, 999
- logLik, 821, 822, 1001
- logLik-class(*setOldClass*), 798
- logLik.gls, 1002
- logLik.lme, 1002
- loglin, 459, 606, 607, 1002
- Lognormal, 1004
- longley, 468
- lower.tri, 98, 212
- lowess, 614, 615, 664, 998, 1005, 1147
- lqs, 1022
- ls, 212, 265, 309, 310, 1258, 1297, 1298
- ls.diag, 1006, 1008, 1009
- ls.print, 1007, 1007, 1009
- ls.str, 213, 1297, 1327
- lsf.str(*ls.str*), 1297
- lsfit, 278, 1006, 1008, 1008
- lynx, 469
- MacRoman(*connections*), 67
- mad, 972, 1009, 1140
- mahalanobis, 1010
- make.link, 1011, 1081, 1082
- make.names, 82, 83, 214, 215, 296
- make.packages.html, 1299
- make.socket, 44, 1266, 1299, 1316
- make.unique, 125, 214, 215, 1305
- makeARIMA(*KalmanLike*), 975
- makeClassRepresentation, 770, 788
- makeLazyLoading, 1244
- makepredictcall, 1012
- makepredictcall.poly(*poly*), 1080
- manglePackageName, 216
- manova, 1013, 1190
- mantelhaen.test, 1014
- maov-class(*setOldClass*), 798
- mapply, 216, 394
- margin.table, 217, 274, 818
- mat.or.vec, 218
- match, 51, 168, 219, 263, 425
- match.arg, 220, 220, 222, 223, 263, 908
- match.call, 221, 221, 263, 759
- match.fun, 221, 222, 222, 263
- Math, 6, 173, 208, 315, 338, 349, 409, 1247
- Math(*groupGeneric*), 169
- Math.data.frame, 83
- Math.Date(*Dates*), 85
- Math.difftime(*difftime*), 99
- Math.POSIXlt(*DateTimeClasses*), 85
- Math.POSIXt(*DateTimeClasses*), 85
- Math2(*groupGeneric*), 169
- matlines(*matplot*), 603
- matmult, 223
- matplot, 463, 603
- matpoints(*matplot*), 603
- matrix, 21, 84, 98, 102, 123, 212, 224, 224, 243, 577, 604
- matrix-class(*StructureClasses*), 806
- max, 17, 289, 426
- max(*Extremes*), 128
- max.col, 426
- max.col(*maxCol*), 225
- maxCol, 225
- mcnemar.test, 1017
- md5sum, 1235, 1245
- mdeaths(*UKLungDeaths*), 498
- mean, 17, 58, 226, 850, 1197, 1224
- mean.Date(*Dates*), 85
- mean.difftime(*difftime*), 99
- mean.POSIXct, 227
- mean.POSIXct(*DateTimeClasses*), 85
- mean.POSIXlt(*DateTimeClasses*), 85
- median, 850, 933, 1010, 1018, 1020, 1147
- medpolish, 1019
- mem.limits(*Memory*), 227
- Memory, 154, 227, 229, 305
- Memory-limits, 228, 1304
- Memory-limits, 229
- memory.profile, 228, 230
- menu, 1301

- merge, [230](#)
- MethodDefinition-class, [759](#), [776](#), [803](#), [804](#)
- MethodDefinition-class, [771](#)
- MethodDefinitionWithTrace-class (*TraceClasses*), [806](#)
- Methods, [289](#), [744](#), [746](#), [759](#), [763](#), [772](#), [788](#), [794](#), [796](#), [797](#), [799](#), [805](#)
- methods, [10](#), [171](#), [179](#), [184](#), [213](#), [241](#), [268](#), [372](#), [417](#), [418](#), [952](#), [994](#), [1185](#), [1285](#), [1289](#), [1301](#)
- MethodsList, [772](#), [796](#), [797](#)
- MethodsList-class, [770](#), [772](#)
- MethodsList-class, [775](#)
- MethodsListSelect, [774](#)
- MethodsListSelect (*getMethod*), [757](#)
- MethodWithNext-class, [772](#)
- MethodWithNext-class, [776](#)
- MethodWithNextWithTrace-class (*TraceClasses*), [806](#)
- mget (*get*), [156](#)
- min, [17](#), [289](#), [426](#)
- min (*Extremes*), [128](#)
- missing, [232](#), [370](#), [762](#)
- missing-class (*BasicClasses*), [742](#)
- mlm-class (*setOldClass*), [798](#)
- Mod, [6](#), [10](#)
- Mod (*complex*), [61](#)
- mode, [10](#), [55](#), [233](#), [335](#), [369](#), [412](#), [417](#), [1255](#), [1298](#)
- mode<- (*mode*), [233](#)
- model.extract, [1020](#), [1024](#)
- model.frame, [916](#), [936](#), [1013](#), [1020](#), [1021](#), [1021](#), [1023](#), [1024](#), [1051](#)
- model.frame.default, [1012](#)
- model.matrix, [22](#), [989](#), [1023](#), [1023](#), [1201](#), [1219](#)
- model.offset, [1051](#)
- model.offset (*model.extract*), [1020](#)
- model.response (*model.extract*), [1020](#)
- model.tables, [831](#), [910](#), [1025](#), [1116](#), [1132](#), [1141](#), [1184](#), [1214](#)
- model.tables.aovlist, [913](#)
- model.weights (*model.extract*), [1020](#)
- month.abb (*Constants*), [72](#)
- month.name (*Constants*), [72](#)
- monthplot, [1026](#)
- months (*weekdays*), [423](#)
- mood.test, [830](#), [853](#), [935](#), [1028](#), [1220](#)
- morley, [470](#)
- mosaicplot, [459](#), [555](#), [584](#), [605](#), [636](#)
- mostattributes<- (*attributes*), [31](#)
- moveToGrob (*grid.move.to*), [702](#)
- mtable-class (*setOldClass*), [798](#)
- mtcars, [471](#)
- mtext, [519](#), [530](#), [532](#), [577](#), [608](#), [615](#), [661](#), [662](#)
- mts-class (*setOldClass*), [798](#)
- Multinomial, [1029](#)
- mvfft (*fft*), [927](#)
- n2mfrow, [610](#), [986](#)
- NA, [20](#), [98](#), [130](#), [139](#), [140](#), [182](#), [193](#), [210](#), [233](#), [234](#), [252](#), [268](#), [270](#), [289](#), [290](#), [296](#), [326](#), [411](#), [425](#), [510](#), [564](#), [567](#), [586](#), [644](#), [663](#), [665](#), [885](#), [933](#), [957](#), [1031](#), [1032](#), [1089](#), [1110](#), [1120](#), [1123](#), [1193](#)
- na.action, [235](#), [993](#), [1031](#), [1033](#), [1034](#)
- na.contiguous, [1031](#), [1207](#)
- na.exclude, [993](#), [1034](#)
- na.exclude (*na.fail*), [1032](#)
- na.fail, [235](#), [875](#), [916](#), [947](#), [988](#), [1022](#), [1031](#), [1032](#), [1032](#), [1092](#), [1106](#), [1207](#)
- na.omit, [235](#), [875](#), [916](#), [947](#), [988](#), [1022](#), [1031](#), [1032](#), [1034](#), [1092](#), [1106](#), [1207](#)
- na.omit (*na.fail*), [1032](#)
- na.omit.ts, [1032](#)
- na.omit.ts (*ts-methods*), [1206](#)
- na.pass (*na.fail*), [1032](#)
- name, [183](#), [196](#), [236](#), [1274](#)
- name-class (*language-class*), [768](#)
- names, [21](#), [31](#), [97](#), [102](#), [128](#), [191](#), [214](#), [237](#), [319](#), [416](#), [1123](#)
- names.dist (*dist*), [907](#)
- names<- (*names*), [237](#)
- names<- .dist (*dist*), [907](#)
- NaN, [20](#), [140](#), [235](#), [567](#), [933](#), [1123](#)
- NaN (*is.finite*), [181](#)
- napredict, [921](#), [932](#), [1032](#), [1106](#)
- napredict (*naresid*), [1034](#)
- naprint, [1033](#)
- naresid, [1032](#), [1034](#), [1135](#)
- nargs, [238](#)
- native.enc (*connections*), [67](#)
- nchar, [239](#), [261](#), [365](#), [371](#), [655](#)
- nclass, [611](#)

- nclass.FD, 588
- nclass.scott, 588
- nclass.Sturges, 588, 589
- NCOL, 319
- NCOL(*nrow*), 242
- ncol, 101
- ncol(*nrow*), 242
- NegBinomial, 1035
- new, 746, 747, 762, 777, 786, 799
- new.env, 750
- new.env(*environment*), 114
- next(*Control*), 74
- NextMethod, 56
- NextMethod(*UseMethod*), 417
- nextn, 882, 928, 1037
- nhtemp, 471
- Nile, 472, 1211
- nlevels, 131, 194, 240
- nlm, 903, 1037, 1056, 1059, 1216
- nls, 162, 938, 946, 1039, 1040, 1043–1045, 1074, 1103, 1112–1114, 1143, 1162–1165, 1167–1170, 1172
- nls.control, 1042
- nlsModel, 1041, 1044, 1114
- NLSstAsymptotic, 1045
- NLSstClosestX, 1046, 1047, 1048, 1153
- NLSstLfAsymptote, 1047, 1047, 1153
- NLSstRtAsymptote, 1047, 1048, 1048, 1153
- noquote, 240, 268, 269, 271, 1193
- Normal, 1048
- nottem, 473
- NotYet, 241
- NotYetImplemented(*NotYet*), 241
- NotYetUsed(*NotYet*), 241
- NROW, 319
- NROW(*nrow*), 242
- nrow, 101, 242
- ns, 1013
- ns-dblcolon, 243
- ns-hooks, 244
- ns-load, 245
- ns-topenv, 246
- nsl, 1303
- NULL, 242, 247, 360, 544, 957, 958
- NULL-class(*BasicClasses*), 742
- numeric, 145, 248, 289
- numeric-class(*BasicClasses*), 742
- numericDeriv, 1050
- object.size, 228, 229, 1303
- objects, 30, 96, 199, 310, 329, 1256
- objects(*ls*), 212
- ObjectsWithPackage-class, 779
- octmode, 249
- offset, 989, 1021, 1051, 1202
- old.packages(*update.packages*), 1333
- oldClass, 170
- oldClass(*class*), 55
- oldClass-class(*setOldClass*), 798
- oldClass<-(*class*), 55
- on.exit, 250, 339, 382, 1299
- oneway.test, 1052
- open(*connections*), 67
- Ops, 19, 60, 100, 210
- Ops(*groupGeneric*), 169
- Ops.Date(*Dates*), 85
- Ops.difftime(*difftime*), 99
- Ops.package\_version(*package-version*), 258
- Ops.POSIXt(*DateTimeClasses*), 85
- Ops.ts(*ts*), 1204
- optim, 838, 839, 843, 844, 877, 878, 903, 920, 961, 1039, 1053, 1181, 1183
- optimise(*optimize*), 1058
- optimize, 1039, 1056, 1058, 1216
- options, 111, 145, 149, 197, 250, 269, 271, 347, 348, 356, 359, 385, 410, 422, 515, 518, 591, 602, 621, 880, 947, 951, 988, 1022, 1031, 1033, 1092, 1106, 1109, 1110, 1260, 1273, 1275, 1276, 1318, 1327
- Orange, 474
- OrchardSprays, 475
- order, 255, 290, 346, 974
- order.dendrogram, 898, 958, 1060
- ordered, 170, 268
- ordered(*factor*), 129
- ordered-class(*setOldClass*), 798
- outer, 189, 217, 222, 257
- p.adjust, 1061, 1063–1065
- pacf(*acf*), 812
- package-version, 258
- package.dependencies, 1245
- package.skeleton, 1304

- package\_version, 1267
- package\_version
  - (package-version), 258
- packageDescription, 1305
- packageEvent (UserHooks), 418
- packageInfo-class (setOldClass), 798
- packageIQR-class (setOldClass), 798
- packageSlot (getPackageName), 760
- packageSlot<- (getPackageName), 760
- packageStatus, 1267, 1306
- packBits (rawConversion), 292
- packGrob (grid.pack), 704
- page, 1308
- pairlist (list), 202
- pairs, 577, 612, 614, 615, 628
- pairwise.prop.test, 1063
- pairwise.t.test, 1062, 1063, 1065
- pairwise.table, 1064
- pairwise.wilcox.test, 1065
- palette, 511, 512, 524, 525, 582, 620, 639, 858
- Palettes, 525
- panel.smooth, 577, 614, 1070, 1199
- par, 325, 519, 543, 546, 552, 553, 556, 558, 559, 561, 563, 573, 577, 579, 593, 595, 601, 602, 604, 606, 609, 610, 614, 615, 623, 626, 627, 630, 631, 633, 634, 638, 640, 642, 644, 647, 648, 650, 651, 655, 656, 660–662, 957, 958, 970, 986, 1066, 1069, 1071, 1072, 1077, 1181, 1199
- Paren, 75, 259, 379
- parent.env (environment), 114
- parent.env<- (environment), 114
- parent.frame, 117
- parent.frame (sys.parent), 381
- parse, 92, 260, 347, 348
- paste, 46, 50, 146, 239, 261, 353, 365, 371, 1264
- path.expand, 35, 137, 262
- pbeta (Beta), 853
- pbinom (Binomial), 856
- pbirthday (birthday), 860
- pcauchy (Cauchy), 866
- pchisq, 1197, 1212
- pchisq (Chisquare), 870
- pdf, 517, 526
- periodicSpline, 1161
- person, 1308
- personList, 1265
- personList (person), 1308
- persp, 419, 622
- pexp (Exponential), 917
- pf (FDist), 926
- pgamma, 349, 1035
- pgamma (GammaDist), 943
- pgeom (Geometric), 945
- phyper (Hypergeometric), 962
- pi (Constants), 72
- pico (edit), 1276
- pictex, 517, 528
- pie, 625
- pipe (connections), 67
- pkgDepends, 253, 1243, 1254
- pkgDepends (getDepList), 1241
- PkgUtils, 1309
- pkgVignettes (buildVignettes), 1233
- placeGrob (grid.place), 705
- PlantGrowth, 476
- plclust (hclust), 953
- plnorm (Lognormal), 1004
- plogis, 173
- plogis (Logistic), 999
- plot, 562, 586, 593, 598, 601, 603, 604, 609, 615, 626, 628–631, 633, 637–640, 656, 663, 912, 955, 1019, 1069, 1075–1077, 1106
- plot.acf, 813, 1066
- plot.data.frame, 83, 628
- plot.Date, 85
- plot.Date (axis.POSIXct), 557
- plot.decomposed.ts (decompose), 892
- plot.default, 557, 569, 573, 581, 585, 604, 619–621, 627, 628, 629, 631, 633, 635, 637–639, 651, 656, 664, 897, 954, 986, 1077
- plot.dendrogram (dendrogram), 895
- plot.density, 901, 1067
- plot.design, 631
- plot.ecdf (ecdf), 911
- plot.factor, 633, 635, 637

- plot.formula, 627, 633, 634
- plot.function (*curve*), 578
- plot.hclust, 896
- plot.hclust (*hclust*), 953
- plot.histogram, 586, 588, 635
- plot.HoltWinters, 1068
- plot.isoreg, 974, 1069
- plot.lm, 1070, 1200
- plot.mlm (*plot.lm*), 1070
- plot.new, 419, 618, 637
- plot.new (*frame*), 585
- plot.POSIXct (*axis.POSIXct*), 557
- plot.POSIXlt (*axis.POSIXct*), 557
- plot.ppr, 1072, 1090
- plot.prcomp (*prcomp*), 1091
- plot.princomp (*princomp*), 1105
- plot.profile.nls, 1073, 1112
- plot.spec, 1074, 1154, 1156, 1160
- plot.stepfun, 911, 912, 1075, 1177
- plot.stl, 1179, 1180
- plot.stl (*stlmethods*), 1180
- plot.table, 636
- plot.ts, 986, 987, 1077, 1181, 1206, 1207
- plot.TukeyHSD (*TukeyHSD*), 1213
- plot.window, 561, 573, 580, 581, 585, 593, 630, 637, 659
- plot.xy, 601, 638, 638, 640
- plotmath, 519, 529, 530, 597, 609, 661, 662
- plotViewport, 670, 722
- pmatch, 49, 51, 168, 220–222, 262
- pmax (*Extremes*), 128
- pmin (*Extremes*), 128
- pnbinom (*NegBinomial*), 1035
- png, 44, 517, 518, 533, 1257
- pnorm, 1213
- pnorm (*Normal*), 1048
- points, 577, 586, 597, 601, 603, 604, 614, 615, 624, 627, 630, 639, 639, 664, 896, 897, 1069, 1070, 1199
- points.default, 639
- points.formula (*plot.formula*), 634
- pointsGrob (*grid.points*), 707
- Poisson, 1078
- poisson (*family*), 924
- poly, 1013, 1080
- polygon, 572, 625, 641, 644, 649, 897
- polygonGrob (*grid.polygon*), 708
- polym (*poly*), 1080
- polyroot, 264, 1216
- pop.viewport, 723, 724
- popViewport, 679, 733
- popViewport (*Working with Viewports*), 734
- pos.to.env, 265
- POSIXct, 22, 147
- POSIXct (*DateTimeClasses*), 85
- POSIXct-class (*setOldClass*), 798
- POSIXlt, 22, 147
- POSIXlt (*DateTimeClasses*), 85
- POSIXlt-class (*setOldClass*), 798
- POSIXt, 98, 170
- POSIXt (*DateTimeClasses*), 85
- POSIXt-class (*setOldClass*), 798
- possibleExtends, 738
- postDrawDetails (*drawDetails*), 670
- postscript, 252, 514–518, 527–529, 535, 540, 550, 621, 640
- postscriptFont (*postscriptFonts*), 539
- postscriptFonts, 528, 538, 539
- power, 924, 925, 1081
- power.anova.test, 1082
- power.prop.test, 1083, 1108
- power.t.test, 1082, 1085, 1108
- PP.test, 1086
- ppoints, 1087, 1121
- ppois (*Poisson*), 1078
- ppr, 1022, 1072, 1088, 1192
- prcomp, 859, 1091, 1106, 1107
- precip, 477
- predict, 120, 916, 990, 1034, 1093, 1100, 1103
- predict.ar, 1094
- predict.ar (*ar*), 832
- predict.Arima, 840, 1094, 1094
- predict.arima0, 1094
- predict.arima0 (*arima0*), 842
- predict.glm, 950, 1094, 1096, 1200
- predict.HoltWinters, 961, 1069, 1094, 1097
- predict.lm, 990, 1094, 1098
- predict.loess, 998, 1094, 1100
- predict.mlm (*predict.lm*), 1098
- predict.nls, 1094, 1102
- predict.poly, 1094
- predict.poly (*poly*), 1080

- predict.prcomp (*prcomp*), 1091
- predict.princomp, 1094
- predict.princomp (*princomp*), 1105
- predict.smooth.spline, 1094, 1103, 1150, 1151
- predict.StructTS, 1094
- predict.StructTS (*StructTS*), 1181
- preDrawDetails (*drawDetails*), 670
- preplot, 1104
- presidents, 477
- pressure, 478
- pretty, 265, 556, 559
- prettyNum, 150
- prettyNum (*format*), 144
- Primitive, 267
- princomp, 859, 860, 922, 996, 1093, 1105, 1139, 1191
- print, 46, 59, 145, 240, 241, 268, 270–272, 823, 831, 898, 911, 955, 1019, 1041, 1106, 1108–1110, 1177, 1323, 1327
- print.anova, 1109
- print.anova (*anova*), 824
- print.aov (*aov*), 830
- print.aovlist (*aov*), 830
- print.ar (*ar*), 832
- print.arima0 (*arima0*), 842
- print.AsIs (*AsIs*), 25
- print.Bibtex (*toLatex*), 1332
- print.by (*by*), 41
- print.checkDocFiles (*QC*), 1246
- print.checkDocStyle (*QC*), 1246
- print.checkFF (*checkFF*), 1234
- print.checkReplaceFuns (*QC*), 1246
- print.checkS3methods (*QC*), 1246
- print.checkTnF (*checkTnF*), 1235
- print.checkVignettes (*checkVignettes*), 1236
- print.codoc (*codoc*), 1237
- print.codocClasses (*codoc*), 1237
- print.codocData (*codoc*), 1237
- print.condition (*conditions*), 63
- print.connection (*connections*), 67
- print.data.frame, 83, 269
- print.Date (*Dates*), 85
- print.default, 59, 251, 268, 269, 270, 272, 897
- print.dendrogram (*dendrogram*), 895
- print.density (*density*), 899
- print.difftime (*difftime*), 99
- print.dist (*dist*), 907
- print.DLLInfo (*getLoadedDLLs*), 160
- print.DLLInfoList (*getLoadedDLLs*), 160
- print.DLLRegisteredRoutines (*getDLLRegisteredRoutines*), 159
- print.dummy.coef (*dummy.coef*), 909
- print.ecdf (*ecdf*), 911
- print.factanal (*loadings*), 995
- print.family (*family*), 924
- print.formula (*formula*), 935
- print.ftable (*ftable*), 940
- print.getAnywhere (*getAnywhere*), 1283
- print.glm (*glm*), 947
- print.hclust (*hclust*), 953
- print.hsearch (*help.search*), 1290
- print.infl (*influence.measures*), 965
- print.integrate (*integrate*), 968
- print.Latex (*toLatex*), 1332
- print.libraryIQR (*library*), 196
- print.lm (*lm*), 988
- print.loadings, 922
- print.loadings (*loadings*), 995
- print.logLik (*logLik*), 1001
- print.ls\_str (*ls.str*), 1297
- print.matrix (*print.default*), 270
- print.MethodsFunction (*methods*), 1301
- print.NativeRoutineList (*getDLLRegisteredRoutines*), 159
- print.noquote (*noquote*), 240
- print.octmode (*octmode*), 249
- print.package\_version (*package-version*), 258
- print.packageDescription (*packageDescription*), 1305
- print.packageInfo (*library*), 196
- print.packageIQR (*data*), 1268
- print.packageStatus (*packageStatus*), 1306
- print.POSIXct (*DateTimeClasses*), 85

- `print.POSIXlt` (*DateTimeClasses*), 85
- `print.power.htest`, 1107
- `print.prcomp` (*prcomp*), 1091
- `print.princomp` (*princomp*), 1105
- `print.recordedplot` (*recordPlot*), 543
- `print.restart` (*conditions*), 63
- `print.rle` (*rle*), 313
- `print.sessionInfo` (*sessionInfo*), 1325
- `print.simple.list` (*print*), 268
- `print.socket` (*make.socket*), 1299
- `print.stepfun` (*stepfun*), 1176
- `print.StructTS` (*StructTS*), 1181
- `print.summary.aov` (*summary.aov*), 1184
- `print.summary.aovlist` (*summary.aov*), 1184
- `print.summary.glm`, 1109
- `print.summary.glm` (*summary.glm*), 1185
- `print.summary.lm`, 145, 1109, 1110
- `print.summary.lm` (*summary.lm*), 1187
- `print.summary.manova` (*summary.manova*), 1189
- `print.summary.prcomp` (*prcomp*), 1091
- `print.summary.princomp` (*summary.princomp*), 1190
- `print.summary.table` (*table*), 390
- `print.table` (*print*), 268
- `print.terms` (*terms*), 1200
- `print.ts`, 1108, 1206
- `print.TukeyHSD` (*TukeyHSD*), 1213
- `print.undoc` (*undoc*), 1252
- `print.xtabs` (*xtabs*), 1231
- `printCoefmat`, 252, 1109
- `prmatrix`, 271
- `proc.time`, 155, 272, 388, 389
- `prod`, 273
- `profile`, 1074, 1110, 1112, 1113
- `profile.glm`, 1111
- `profile.nls`, 1074, 1111, 1111, 1114
- `profiler`, 1112, 1114
- `profiler.nls`, 1112, 1113, 1113
- `proj`, 831, 1026, 1115
- `promax` (*varimax*), 1220
- `promises`, 140
- `prompt`, 781, 782, 1238, 1289, 1305, 1310, 1313
- `promptClass`, 779, 782, 1238
- `promptData`, 1311, 1312
- `promptMethods`, 781, 781
- `prop.table`, 274
- `prop.test`, 856, 1063, 1084, 1117, 1119, 1196
- `prop.trend.test`, 1119
- `prototype`, 770
- `prototype` (*representation*), 782
- `ps.options`, 510, 550
- `ps.options` (*postscript*), 535
- `psigamma` (*Special*), 348
- `psignrank` (*SignRank*), 1145
- `pt` (*TDist*), 1197
- `ptukey` (*Tukey*), 1212
- `punif` (*Uniform*), 1214
- `Puromycin`, 479
- `push.viewport`, 723, 724
- `pushBack`, 69, 70, 275, 401
- `pushBackLength` (*pushBack*), 275
- `pushViewport`, 679, 733
- `pushViewport` (*Working with Viewports*), 734
- `pweibull` (*Weibull*), 1222
- `pwilcox` (*Wilcoxon*), 1228
- `q`, 322, 359
- `q` (*quit*), 279
- `qbeta` (*Beta*), 853
- `qbinom` (*Binomial*), 856
- `qbirthday` (*birthday*), 860
- `QC`, 1239, 1246, 1253
- `qcauchy` (*Cauchy*), 866
- `qchisq` (*Chisquare*), 870
- `qexp` (*Exponential*), 917
- `qf` (*FDist*), 926
- `qgamma` (*GammaDist*), 943
- `qgeom` (*Geometric*), 945
- `qhyper` (*Hypergeometric*), 962
- `qlnorm` (*Lognormal*), 1004
- `qlogis` (*Logistic*), 999
- `qnbinom` (*NegBinomial*), 1035
- `qnorm`, 285, 1213
- `qnorm` (*Normal*), 1048
- `qpois` (*Poisson*), 1078

- qqline (*qqnorm*), 1120
- qqnorm, 1088, 1120, 1145
- qqplot, 1088
- qqplot (*qqnorm*), 1120
- qr, 34, 53, 114, 188, 276, 278, 279, 375, 865, 991, 992
- QR.Auxiliaries, 278
- qr.Q, 278
- qr.Q (*QR.Auxiliaries*), 278
- qr.qy, 279
- qr.R, 278
- qr.R (*QR.Auxiliaries*), 278
- qr.solve, 344
- qr.X, 278
- qr.X (*QR.Auxiliaries*), 278
- qsignrank (*SignRank*), 1145
- qt (*TDist*), 1197
- qtukey, 1214
- qtukey (*Tukey*), 1212
- quade.test, 939, 1121
- quakes, 480
- quantile, 567, 933, 972, 1018, 1123
- quarters (*weekdays*), 423
- quartz, 541, 542, 591, 602
- quartzFont (*quartzFonts*), 542
- quartzFonts, 541, 542
- quasi (*family*), 924
- quasibinomial (*family*), 924
- quasipoisson (*family*), 924
- Querying the Viewport Tree, 724
- quit, 279
- qunif (*Uniform*), 1214
- quote, 39, 405, 406, 532, 768
- quote (*substitute*), 369
- qweibull (*Weibull*), 1222
- qwilcox (*Wilcoxon*), 1228
  
- R.home, 281
- R.Version, 281
- R.version, 4, 184, 380
- R.version (*R.Version*), 281
- r2dtable, 282
- R\_HOME (*RHOME*), 1320
- R\_LIBS (*library*), 196
- rainbow, 512, 519, 522, 524, 544, 593, 621
- rainbow (*Palettes*), 525
- Random, 283
- Random.user, 285, 287
- randu, 481
  
- range, 129, 289, 577, 933, 972
- rank, 256, 290, 346
- raw, 291
- rawConversion, 292
- rawShift (*rawConversion*), 292
- rawToBits (*rawConversion*), 292
- rawToChar, 291
- rawToChar (*rawConversion*), 292
- rbeta (*Beta*), 853
- rbind (*cbind*), 46
- rbinom, 1030
- rbinom (*Binomial*), 856
- rcauchy (*Cauchy*), 866
- rchisq (*Chisquare*), 870
- Rd2dvi (*RdUtils*), 293
- Rd2txt (*RdUtils*), 293
- Rd\_db (*Rdutils*), 1248
- Rd\_parse (*Rdutils*), 1248
- Rdconv, 781
- Rdconv (*RdUtils*), 293
- Rdindex, 1247
- RdUtils, 293
- Rdutils, 1248
- Re (*complex*), 61
- read.00Index, 1250
- read.csv, 1314
- read.csv (*read.table*), 294
- read.csv2 (*read.table*), 294
- read.dcf, 1306, 1335
- read.dcf (*dcf*), 88
- read.delim (*read.table*), 294
- read.delim2 (*read.table*), 294
- read.fortran, 1313
- read.ftable, 941, 1125
- read.fwf, 297, 1314, 1314
- read.socket, 1266, 1300, 1316
- read.table, 76, 83, 228, 294, 328, 412, 430, 1268, 1313–1315
- readBin, 70, 298, 302
- readChar (*readBin*), 298
- readCitationFile (*citEntry*), 1264
- readline, 301, 1311
- readLines, 70, 275, 299, 302, 327, 328, 431
- real, 303
- Recall, 43, 304
- recordedplot-class (*setOldClass*), 798
- recordPlot, 543

- recover, 90, 404–406, 1273, 1317
- rect, 563, 635, 642, 643
- rect.hclust, 955, 956, 964, 1127
- rectGrob(*grid.rect*), 711
- reformulate(*delete.response*), 893
- reg.finalizer, 154, 304
- regex, 305
- regexp, 168
- regexp(*regex*), 305
- regexpr, 51, 1239
- regexpr(*grep*), 167
- regular expression, 167, 168, 204, 213, 364, 365, 1255, 1258, 1290, 1291, 1298
- regular expression(*regex*), 305
- relevel, 1128
- REMOVE, 199, 1296, 1319, 1320, 1335
- remove, 309
- remove.packages, 1319
- removeCConverter(*getNumCConverters*), 163
- removeClass(*setClass*), 785
- removeGeneric(*GenericFunctions*), 752
- removeGrob, 675, 681, 692, 694, 713
- removeGrob(*grid.remove*), 712
- removeMethod(*setMethod*), 795
- removeMethods(*GenericFunctions*), 752
- removeTaskCallback, 398, 399
- removeTaskCallback(*taskCallback*), 395
- Renviron.site(*Startup*), 356
- reorder, 958, 1060, 1129
- reorder.dendrogram, 898, 957
- reorder.factor, 1130
- rep, 125, 310, 331, 334, 663, 665, 731
- repeat(*Control*), 74
- repeat-class(*language-class*), 768
- replace, 312
- replayPlot(*recordPlot*), 543
- replicate(*lapply*), 191
- replications, 831, 1026, 1131
- representation, 782, 785
- require, 252, 356
- require(*library*), 196
- resetClass(*setClass*), 785
- reshape, 355, 1132
- resid, 1034, 1041
- resid(*residuals*), 1134
- residuals, 824, 875, 915, 933, 950, 953, 990, 995, 1134, 1199, 1225
- residuals.glm, 995, 1186
- residuals.glm(*glm.summaries*), 952
- residuals.lm(*lm.summaries*), 994
- residuals.tukeyline(*line*), 987
- restartDescription(*conditions*), 63
- restartFormals(*conditions*), 63
- return, 180, 259
- return(*function*), 153
- rev, 313, 957
- rev.dendrogram, 1129
- rev.dendrogram(*dendrogram*), 895
- rexp(*Exponential*), 917
- rf(*FDist*), 926
- rgamma(*GammaDist*), 943
- rgb, 511, 512, 519, 522, 525, 543, 545, 621
- rgb2hsv, 544
- rgeom(*Geometric*), 945
- RHOME, 1320
- rhyper(*Hypergeometric*), 962
- rivers, 482
- rle, 313
- rle-class(*setOldClass*), 798
- rlnorm(*Lognormal*), 1004
- rlogis(*Logistic*), 999
- rm(*remove*), 309
- rmultinom(*Multinomial*), 1029
- rnbinom(*NegBinomial*), 1035
- RNG(*Random*), 283
- RNGkind, 287
- RNGkind(*Random*), 283
- RNGversion(*Random*), 283
- rnorm, 286, 1215
- rnorm(*Normal*), 1048
- rock, 483
- Round, 314
- round, 100, 177, 316
- round(*Round*), 314
- round.Date, 85
- round.Date(*round.POSIXt*), 316
- round.difftime(*difftime*), 99
- round.POSIXt, 87, 316
- row, 57, 317, 331, 341
- row.names, 83, 318, 319

- row.names<- (row.names), 318
- row/colnames, 319
- rowMeans (colSums), 57
- rownames, 102, 318
- rownames (row/colnames), 319
- rownames<- (row/colnames), 319
- rowsum, 58, 320
- rowSums, 320
- rowSums (colSums), 57
- rpart, 811
- rpois (Poisson), 1078
- Rprof, 357, 1111, 1320, 1328, 1329
- Rprofile (Startup), 356
- rsignrank (SignRank), 1145
- rstandard (influence.measures), 965
- rstudent, 995
- rstudent (influence.measures), 965
- rt (TDist), 1197
- Rtangle, 1321, 1324, 1331, 1332
- RtangleSetup (Rtangle), 1321
- rug, 187, 645, 1199
- runif, 286, 1049
- runif (Uniform), 1214
- runmed, 1135, 1152
- RweaveLatex, 1322, 1322, 1331, 1332
- RweaveLatexSetup (RweaveLatex), 1322
- rweibull (Weibull), 1222
- rwilcox (Wilcoxon), 1228
  
- S3Methods, 1302
- S3Methods (UseMethod), 417
- SafePrediction, 1097, 1100
- SafePrediction (makepredictcall), 1012
- sammon, 874
- sample, 321
- sapply, 216, 217, 394
- sapply (lapply), 191
- save, 29, 107, 108, 205, 322, 429, 1269, 1272
- savehistory, 1324
- scale, 324, 376, 1013, 1092
- scan, 76, 228, 260, 275, 295–297, 302, 325, 348, 429, 1315
- scatter.smooth, 1137
- SClassExtension-class, 747, 784
- screen, 646
- screeplot, 1106, 1107, 1139
  
- sd, 885, 1140, 1197
- Sd2Rd (RdUtils), 293
- se.contrast, 1026, 1140
- se.contrast.aovlist, 913
- sealClass (setClass), 785
- SealedMethodDefinition-class (MethodDefinition-class), 771
- search, 23, 27, 29, 30, 66, 96, 119, 157, 196, 199, 213, 309, 328, 509, 761, 1256, 1298
- searchpaths (search), 328
- Seatbelts (UKDriverDeaths), 496
- seek, 70, 329
- seekViewport, 679, 733
- seekViewport (Working with Viewports), 734
- segments, 552, 553, 642, 644, 648, 896, 897, 1076
- segmentsGrob (grid.segments), 713
- selectMethod, 743, 748, 802, 803
- selectMethod (getMethod), 757
- selfStart, 946, 1047, 1048, 1142, 1153, 1162–1165, 1167–1170, 1172
- selfStart.default, 946
- selfStart.formula, 946
- seq, 311, 313, 330, 332–334
- seq.Date, 80, 85, 332
- seq.POSIXt, 80, 87, 331, 333, 590
- sequence, 311, 331, 334
- sessionInfo, 1325, 1333
- set.seed (Random), 283
- setAs, 739, 765
- setAs (as), 738
- setCConverterStatus (getNumCConverters), 163
- setChildren (grid.add), 680
- setClass, 739, 745–747, 755–757, 765, 767, 770, 771, 775, 782, 783, 785, 799, 800, 808, 809, 1295
- setClassUnion, 748, 788, 789, 800
- setdiff (sets), 335
- setequal (sets), 335
- setGeneric, 751, 755, 774, 775, 791, 796
- setGrob, 675
- setGrob (grid.set), 714
- setGroupGeneric, 751
- setGroupGeneric (setGeneric), 791

- setHook, 198, 245, 585, 623, 703
- setHook (*UserHooks*), 418
- setIs, 739, 745, 746, 756, 773, 784
- setIs (*is*), 764
- setMethod, 406, 737, 751, 763, 767, 771, 786, 795, 799, 800, 803, 1295
- setNames, 872, 1143
- setOldClass, 787, 788, 796, 798
- setPackageName (*getPackageName*), 760
- setReplaceMethod (*GenericFunctions*), 752
- sets, 335
- setValidity (*validObject*), 808
- setwd, 383
- setwd (*getwd*), 165
- shapiro.test, 983, 1144
- SHLIB, 112, 201, 1268, 1325
- show, 271, 800, 802
- show, ANY-method (*show*), 800
- show, classRepresentation-method (*show*), 800
- show, genericFunction-method (*show*), 800
- show, MethodDefinition-method (*show*), 800
- show, MethodWithNext-method (*show*), 800
- show, ObjectsWithPackage-method (*show*), 800
- show, traceable-method (*show*), 800
- show-methods (*show*), 800
- showClass, 801
- showConnections, 70, 336, 401
- showDefault, 800
- showMethods, 755, 773, 801, 802, 1302
- showMlist, 801
- shQuote, 337
- sign, 338
- signalCondition, 359
- signalCondition (*conditions*), 63
- Signals, 339
- signature (*GenericFunctions*), 752
- signature-class, 803
- signif, 150, 373
- signif (*Round*), 314
- SignRank, 1145
- simpleCondition (*conditions*), 63
- simpleError (*conditions*), 63
- simpleWarning (*conditions*), 63
- sin, 7, 173
- sin (*Trig*), 409
- single, 141
- single (*double*), 103
- single-class (*BasicClasses*), 742
- sinh (*Hyperbolic*), 173
- sink, 45, 336, 339, 1262
- sleep, 483
- slice.index, 341
- slot, 342, 746, 804
- slot<- (*slot*), 804
- slotNames (*slot*), 804
- slotOp, 342
- smooth, 1137, 1146
- smooth.spline, 1090, 1104, 1147, 1148, 1161
- smoothEnds, 1136, 1137, 1152
- socket-class (*setOldClass*), 798
- socketConnection (*connections*), 67
- socketSelect, 342
- solve, 34, 54, 343, 1011
- solve.qr, 278, 344
- solve.qr (*qr*), 276
- sort, 207, 256, 290, 313, 344
- sort.list (*order*), 255
- sortedXyData, 1047, 1048, 1153
- source, 107, 178, 260, 346, 385, 1236, 1237, 1268, 1274, 1330
- spec (*spectrum*), 1158
- spec.ar, 1154, 1159, 1160
- spec.pgram, 1155, 1158–1160
- spec.taper, 1156, 1157
- Special, 7, 20, 348
- spectrum, 977, 1075, 1155, 1156, 1158
- spline, 17
- spline (*splinefun*), 1160
- splinefun, 17, 578, 912, 1076, 1160, 1177
- split, 79, 350
- split.screen, 595, 618, 621
- split.screen (*screen*), 646
- split<- (*split*), 350
- sprintf, 146, 151, 261, 351
- sqrt, 20, 209, 349
- sqrt (*abs*), 6
- sQuote, 338, 353, 370
- SSasymp, 1046, 1162, 1171, 1172

- SSasymptOff, 1163
- SSasymptOrig, 1164
- SSbiexp, 1165
- SSfol, 491, 1166
- SSfpl, 1167
- SSgompertz, 1168
- SSlogis, 1169
- SSmicmen, 1170
- SSweibull, 1171
- stack, 354, 1133
- stack.loss (stackloss), 484
- stack.x (stackloss), 484
- stackloss, 484
- standardGeneric, 751
- standardGeneric  
(GenericFunctions), 752
- Stangle, 1236, 1321
- Stangle (Sweave), 1329
- stars, 649, 659
- start, 1172, 1203, 1206, 1210
- Startup, 252, 356
- stat.anova, 825, 1173
- state, 485, 500
- stderr (showConnections), 336
- stdin, 275
- stdin (showConnections), 336
- stdout (showConnections), 336
- stem, 589, 636, 652
- step, 817, 918, 919, 1174
- stepAIC, 1175, 1176
- stepfun, 912, 974, 1075, 1176
- stl, 892, 893, 1027, 1178, 1180, 1181
- stlmethods, 1180
- stop, 252, 358, 360, 422
- stopifnot, 9, 359, 360
- storage.mode, 412
- storage.mode (mode), 233
- storage.mode<- (mode), 233
- str, 213, 898, 1258, 1297, 1298, 1326
- str.default, 897
- str.dendrogram (dendrogram), 895
- str.logLik (logLik), 1001
- str.POSIXt (DateTimeClasses), 85
- strftime, 147
- strftime (strptime), 361
- strheight (strwidth), 654
- stringHeight (stringWidth), 726
- stringWidth, 722, 726
- stripchart, 565, 628, 653
- strptime, 25, 87, 207, 208, 361, 557, 590
- strsplit, 50, 239, 261, 305, 309, 364, 371
- StructTS, 976, 1027, 1181, 1209, 1211
- structure, 366
- structure-class  
(StructureClasses), 806
- StructureClasses, 806
- strwidth, 239, 597, 617, 654
- strwrap, 366
- sub, 50, 51, 365
- sub (grep), 167
- Subscript (Extract), 121
- subset, 126, 367, 409
- substitute, 39, 92, 233, 369, 405, 406,  
532, 597
- substr, 6, 50, 239, 261, 365, 370
- substr<- (substr), 370
- substring (substr), 370
- substring<- (substr), 370
- sum, 58, 274, 372
- Summary, 8, 12, 128, 273, 289, 372
- Summary (groupGeneric), 169
- summary, 372, 824, 831, 948, 950, 994, 1041,  
1184, 1187, 1189, 1190, 1298, 1326,  
1327
- summary.aov, 831, 1184
- summary.aovlist (summary.aov),  
1184
- summary.connection (connections),  
67
- Summary.Date (Dates), 85
- summary.Date (Dates), 85
- Summary.difftime (difftime), 99
- summary.glm, 373, 948, 950, 953, 1185
- summary.infl  
(influence.measures), 965
- summary.lm, 373, 990, 994, 995, 1007, 1187
- summary.manova, 1014, 1189
- summary.mlm (summary.lm), 1187
- Summary.package\_version  
(package-version), 258
- summary.packageStatus  
(packageStatus), 1306
- Summary.POSIXct  
(DateTimeClasses), 85
- summary.POSIXct  
(DateTimeClasses), 85

- Summary.POSIXlt
  - (*DateTimeClasses*), 85
- summary.POSIXlt
  - (*DateTimeClasses*), 85
- summary.prcomp (*prcomp*), 1091
- summary.princomp, 1107, 1190
- summary.stepfun (*stepfun*), 1176
- summary.table (*table*), 390
- summary.table-class
  - (*setOldClass*), 798
- summaryRprof, 1320, 1321, 1328
- sunflowerplot, 656, 659
- sunspot.month, 487, 489
- sunspot.year, 487, 488, 488
- sunspots, 487, 488
- suppressWarnings (*warning*), 421
- supsmu, 1090, 1147, 1191
- survival, 25, 147
- survreg, 1199
- svd, 53, 114, 188, 278, 374, 865, 1093, 1106
- Sweave, 1233, 1236, 1322, 1324, 1329, 1331
- SweaveSyntaxLatex (*Sweave*), 1329
- SweaveSyntaxNoweb (*Sweave*), 1329
- SweaveSyntConv, 1331
- sweep, 15, 222, 325, 376, 885
- swiss, 489
- switch, 75, 377
- symbol.C (*dyn.load*), 109
- symbol.For (*dyn.load*), 109
- symbols, 658
- symnum, 1186, 1188, 1192
- Syntax, 20, 61, 75, 123, 210, 259, 378
- sys.call, 238
- sys.call (*sys.parent*), 381
- sys.calls (*sys.parent*), 381
- Sys.Date, 85
- Sys.Date (*Sys.time*), 385
- sys.frame, 27, 117, 119, 157, 213, 309
- sys.frame (*sys.parent*), 381
- sys.frames (*sys.parent*), 381
- sys.function (*sys.parent*), 381
- Sys.getenv, 379, 383
- Sys.getlocale (*locales*), 207
- Sys.getpid (*getpid*), 164
- Sys.info, 4, 380
- sys.load.image (*save*), 322
- Sys.localeconv, 208
- Sys.localeconv (*localeconv*), 206
- sys.nframe (*sys.parent*), 381
- sys.on.exit, 250
- sys.on.exit (*sys.parent*), 381
- sys.parent, 381, 1272
- sys.parents (*sys.parent*), 381
- Sys.putenv, 379, 383
- sys.save.image (*save*), 322
- Sys.setlocale, 206
- Sys.setlocale (*locales*), 207
- Sys.sleep, 384
- sys.source, 246, 385
- sys.status (*sys.parent*), 381
- Sys.time, 84, 87, 385
- Sys.timezone (*Sys.time*), 385
- system, 4, 184, 386
- system.file, 387
- system.time, 272, 273, 388, 1203
- T (*logical*), 211
- t, 13, 389, 1205
- t.test, 981, 1052, 1064, 1086, 1195, 1227
- t.ts (*ts*), 1204
- table, 79, 268, 390, 392, 393, 636, 637, 818, 941, 943, 1004, 1232
- table-class (*setOldClass*), 798
- tabulate, 79, 392
- tail (*head*), 1286
- tan, 173
- tan (*Trig*), 409
- tanh, 1000
- tanh (*Hyperbolic*), 173
- tapply, 15, 41, 191, 320, 393, 820
- taskCallback, 395
- taskCallbackManager, 395, 396, 397, 399
- taskCallbackNames, 398
- TDist, 1197
- tempdir, 1299
- tempdir (*tempfile*), 399
- tempfile, 399
- termplot, 1071, 1198
- terms, 631, 893, 937, 949, 989, 1022, 1024, 1200, 1202, 1203, 1219
- terms.formula, 1200, 1201, 1201, 1202
- terms.object, 1200, 1201, 1202, 1202
- terrain.colors, 524, 592, 593
- terrain.colors (*Palettes*), 525
- texi2dvi, 1233, 1251

- text, 121, 519, 521, 523, 530, 532, 537, 573, 574, 598, 609, 615, 616, 655, 660, 662, 1077, 1078
- textConnection, 70, 400, 1262
- textGrob(*grid.text*), 717
- Theoph, 490
- tilde, 402
- time, 389, 664, 1173, 1203, 1206, 1210, 1231
- Titanic, 491
- title, 519, 532, 561, 569, 573, 577, 580, 581, 604, 609, 615, 622, 627, 631, 661, 662, 954, 1069, 1071
- toBibtex(*toLatex*), 1332
- toBibtex.citation(*citation*), 1263
- toBibtex.citationList(*citation*), 1263
- toBibtex.person(*person*), 1308
- toBibtex.personList(*person*), 1308
- toeplitz, 1204
- toLatex, 1332
- toLatex.sessionInfo(*sessionInfo*), 1325
- tolower, 168
- tolower(*chartr*), 51
- tools-deprecated, 1251
- ToothGrowth, 493
- topenv, 385
- topenv(*ns-topenv*), 246
- topicName(*help*), 1287
- topo.colors, 512, 592, 593
- topo.colors(*Palettes*), 525
- toString, 403
- toupper, 168
- toupper(*chartr*), 51
- trace, 404, 763, 806, 807
- traceable-class, 763
- traceable-class(*TraceClasses*), 806
- traceback, 40, 90, 359, 407
- TraceClasses, 806
- tracingState(*trace*), 404
- transform, 368, 408
- treering, 493
- trees, 494
- Trig, 209, 409
- trigamma(*Special*), 348
- TRUE, 210, 360, 1078
- TRUE(*logical*), 211
- truehist, 589
- trunc, 177
- trunc(*Round*), 314
- trunc.Date(*round.POSIXt*), 316
- trunc.POSIXt, 87
- trunc.POSIXt(*round.POSIXt*), 316
- truncate(*seek*), 329
- try, 132, 198, 252, 359, 407, 410, 419
- tryCatch, 407
- tryCatch(*conditions*), 63
- ts, 98, 170, 252, 1027, 1078, 1109, 1173, 1203, 1204, 1206, 1210, 1230, 1231
- ts-class(*StructureClasses*), 806
- ts-methods, 1206
- ts.intersect, 990
- ts.intersect(*ts.union*), 1208
- ts.plot, 1207
- ts.union, 1208
- tsdiag, 840, 845, 1209
- tsp, 1172, 1173, 1203, 1205, 1206, 1210, 1230
- tsp<-(*tsp*), 1210
- tsSmooth, 976, 1183, 1211
- Tukey, 1212
- TukeyHSD, 831, 1026, 1184, 1213
- type.convert, 296, 297, 411
- typeof, 20, 233, 234, 236, 412
- UCBAdmissions, 495
- ucv, 851
- UKDriverDeaths, 496
- UKgas, 497
- UKLungDeaths, 498
- unclass, 131
- unclass(*class*), 55
- undebug(*debug*), 89
- undoc, 1239, 1252
- Uniform, 1214
- union(*sets*), 335
- unique, 109, 413
- uniroot, 264, 1039, 1059, 1083, 1084, 1086, 1216
- unit, 668, 676, 679, 686, 687, 701, 722, 726, 726, 729
- unit.c, 728, 728
- unit.length, 728, 729
- unit.pmax(*unit.pmin*), 730
- unit.pmin, 730
- unit.rep, 730

- units, [546](#)
- unix(*system*), [386](#)
- unix.time(*system.time*), [388](#)
- unlink, [137](#), [400](#), [414](#)
- unlist, [42](#), [179](#), [415](#)
- unloadNamespace, [244](#)
- unloadNamespace(*ns-load*), [245](#)
- unname, [416](#)
- unsplit(*split*), [350](#)
- unstack(*stack*), [354](#)
- untrace, [806](#)
- untrace(*trace*), [404](#)
- unz(*connections*), [67](#)
- update, [1217](#)
- update.formula, [1175](#), [1218](#), [1218](#)
- update.packages, [253](#), [1246](#), [1296](#), [1306](#), [1307](#), [1333](#)
- update.packageStatus  
(*packageStatus*), [1306](#)
- upgrade(*packageStatus*), [1306](#)
- upper.tri, [98](#)
- upper.tri(*lower.tri*), [212](#)
- upViewport, [679](#), [733](#)
- upViewport(*Working with Viewports*), [734](#)
- url, [44](#), [205](#), [1276](#), [1335](#)
- url(*connections*), [67](#)
- url.show, [1276](#), [1335](#)
- USAccDeaths, [499](#)
- USArrests, [499](#)
- UseMethod, [47](#), [56](#), [235](#), [417](#)
- UserHooks, [418](#)
- USJudgeRatings, [500](#)
- USPersonalExpenditure, [501](#)
- uspop, [501](#)
- utils-deprecated, [1336](#)
  
- VADeaths, [502](#)
- validDetails, [731](#)
- validObject, [747](#), [770](#), [786](#), [808](#)
- var, [889](#), [1010](#), [1011](#), [1140](#)
- var(*cor*), [884](#)
- var.test, [830](#), [853](#), [935](#), [1029](#), [1219](#)
- variable.names, [319](#)
- variable.names  
(*case/variable.names*), [865](#)
- varimax, [922](#), [1220](#)
- vcov, [839](#), [990](#), [1041](#), [1222](#)
- vector, [149](#), [203](#), [420](#)
- vector-class(*BasicClasses*), [742](#)
- version(*R.Version*), [281](#)
- vi, [1271](#)
- vi(*edit*), [1276](#)
- viewport, [670](#), [676](#), [682](#), [685](#), [696](#), [699–701](#), [703](#), [708](#), [709](#), [712](#), [714](#), [716–718](#), [720](#), [721](#), [723](#), [725](#), [733](#), [735](#)
- viewport(*Grid Viewports*), [676](#)
- vignette, [1336](#)
- vignetteDepends, [1253](#)
- VIRTUAL-class(*BasicClasses*), [742](#)
- volcano, [503](#)
- vpList(*Grid Viewports*), [676](#)
- vpPath, [732](#), [735](#)
- vpStack(*Grid Viewports*), [676](#)
- vpTree(*Grid Viewports*), [676](#)
  
- warning, [20](#), [47](#), [253](#), [359](#), [360](#), [421](#), [423](#)
- warnings, [252](#), [422](#), [422](#)
- warpbreaks, [504](#)
- weekdays, [85](#), [423](#)
- weekdays.POSIXt, [87](#)
- Weibull, [1222](#)
- weighted.mean, [227](#), [1224](#)
- weighted.residuals, [995](#), [1224](#)
- weights, [1225](#)
- weights(*lm.summaries*), [994](#)
- weights.glm(*glm*), [947](#)
- which, [424](#), [426](#)
- which.is.max, [426](#)
- which.max, [226](#)
- which.max(*which.min*), [426](#)
- which.min, [129](#), [425](#), [426](#)
- while(*Control*), [74](#)
- while-class(*language-class*), [768](#)
- width.SJ, [851](#)
- widthDetails, [667](#), [733](#)
- wilcox.test, [981](#), [1065](#), [1146](#), [1225](#), [1229](#)
- Wilcoxon, [1228](#)
- WinAnsi(*connections*), [67](#)
- window, [1203](#), [1206](#), [1230](#)
- window<-(*window*), [1230](#)
- with, [30](#), [427](#), [1262](#)
- withCallingHandlers(*conditions*), [63](#)
- withRestarts(*conditions*), [63](#)
- women, [505](#)
- Working with Viewports, [734](#)

WorldPhones, 505  
write, 106, 108, 269, 328, 428, 430  
write.dcf (*dcf*), 88  
write.ftable (*read.ftable*), 1125  
write.matrix, 430  
write.socket (*read.socket*), 1316  
write.table, 89, 297, 429, 429  
writeBin, 70, 431  
writeBin (*readBin*), 298  
writeChar, 431  
writeChar (*readBin*), 298  
writeLines, 70, 299, 302, 431  
wsbrowser (*browseEnv*), 1257  
WWWusage, 506

X11, 253, 518, 533, 534, 549, 591, 602  
X11 (*x11*), 547  
x11, 547, 621, 1257  
X11Font (*X11Fonts*), 548  
X11Fonts, 548  
xaxisGrob (*grid.xaxis*), 719  
xedit (*edit*), 1276  
xemacs (*edit*), 1276  
xfig, 517, 549  
xinch (*units*), 546  
xor (*Logic*), 209  
xtabs, 391, 504, 941, 1126, 1231  
xy.coords, 16, 571, 572, 597, 598, 601,  
    629, 630, 638–640, 642, 656, 658,  
    660, 663, 666, 974, 1160  
xyinch (*units*), 546  
xyz.coords, 665

yaxisGrob (*grid.yaxis*), 720  
yinch (*units*), 546

zapsmall, 1110  
zapsmall (*Round*), 314  
zip.file.extract, 431  
zpackages, 432