

# Package ‘pracma’

November 10, 2023

**Type** Package

**Version** 2.4.4

**Date** 2023-11-08

**Title** Practical Numerical Math Functions

**Depends** R (>= 3.1.0)

**Imports** graphics, grDevices, stats, utils

**Suggests** NlcOptim, quadprog

**Description** Provides a large number of functions from numerical analysis and linear algebra, numerical optimization, differential equations, time series, plus some well-known special mathematical functions. Uses 'MATLAB' function names where appropriate to simplify porting.

**License** GPL (>= 3)

**ByteCompile** true

**LazyData** yes

**NeedsCompilation** no

**Author** Hans W. Borchers [aut, cre]

**Maintainer** Hans W. Borchers <hwborchers@googlemail.com>

**Repository** CRAN

**Date/Publication** 2023-11-10 00:10:02 UTC

## R topics documented:

pracma-package . . . . .	8
abm3pc . . . . .	11
accumarray . . . . .	12
agmean . . . . .	14
aitken . . . . .	16
akimaInterp . . . . .	17
and, or . . . . .	19
andrewsplot . . . . .	20
angle . . . . .	21

anms . . . . .	22
approx_entropy . . . . .	24
arclength . . . . .	26
arnoldi . . . . .	28
barylag . . . . .	29
barylag2d . . . . .	31
bernoulli . . . . .	32
bernstein . . . . .	34
bisect . . . . .	35
bits . . . . .	37
blanks . . . . .	38
blkdiag . . . . .	38
brentDekker . . . . .	39
brown72 . . . . .	40
broyden . . . . .	41
bsxfun . . . . .	43
bulirsch-stoer . . . . .	44
bvp . . . . .	46
cart2sph . . . . .	47
cd, pwd, what . . . . .	48
ceil . . . . .	49
charpoly . . . . .	50
chebApprox . . . . .	51
chebCoeff . . . . .	52
chebPoly . . . . .	53
circlefit . . . . .	55
clear, who(s), ver . . . . .	56
clenshaw_curtis . . . . .	57
combs . . . . .	58
compan . . . . .	58
complexstep . . . . .	59
cond . . . . .	61
conv . . . . .	62
cot,csc,sec, etc. . . . .	63
cotes . . . . .	64
coth,csch,sech, etc. . . . .	65
cranknic . . . . .	66
cross . . . . .	68
crossn . . . . .	69
cubicspline . . . . .	70
curvefit . . . . .	71
cutpoints . . . . .	73
dblquad . . . . .	74
deconv . . . . .	76
deeve . . . . .	77
deg2rad . . . . .	78
detrend . . . . .	78
deval . . . . .	79

Diag . . . . .	80
disp,beep . . . . .	81
distmat . . . . .	82
dot . . . . .	83
eig . . . . .	84
eigjacobi . . . . .	85
einsteinF . . . . .	86
ellipke,ellipj . . . . .	87
eps . . . . .	89
erf . . . . .	90
errorbar . . . . .	91
eta . . . . .	92
euler_heun . . . . .	93
expint . . . . .	94
expm . . . . .	96
eye . . . . .	97
ezcontour,ezsurf,ezmesh . . . . .	98
ezplot . . . . .	99
ezpolar . . . . .	101
fact . . . . .	102
factors . . . . .	103
fderiv . . . . .	104
fibsearch . . . . .	105
figure . . . . .	106
findintervals . . . . .	107
findmins . . . . .	108
findpeaks . . . . .	109
finds . . . . .	110
findzeros . . . . .	111
fletcher_powell . . . . .	112
flipdim . . . . .	114
fminbnd . . . . .	115
fmincon . . . . .	116
fminsearch . . . . .	118
fminunc . . . . .	119
fnorm . . . . .	121
fornberg . . . . .	122
fprintf . . . . .	123
fractalcurve . . . . .	124
fresnelS/C . . . . .	126
fsolve . . . . .	127
fzero . . . . .	129
fzsolve . . . . .	130
gammainc . . . . .	131
gammaz . . . . .	132
gaussHermite . . . . .	133
gaussLaguerre . . . . .	134
gaussLegendre . . . . .	136

gaussNewton	137
gauss_kronrod	139
gcd, lcm	140
geomean, harmmean	141
geo_median	142
givens	143
gmres	144
golden_ratio	145
grad	146
gradient	147
gramSchmidt	149
hadamard	150
halley	151
hampel	152
hankel	153
hausdorff_dist	154
haversine	155
hessenberg	156
hessian	157
Hessian utilities	158
hilb	160
hisc	161
histss	162
hooke_jeeves	163
horner	165
householder	166
humps	167
hurstexp	168
hypot	170
iff	171
inpolygon	172
integral	174
integral2	176
interp1	179
interp2	180
inv	182
invlap	183
isempty	184
isposdef	185
isprime	186
itersolve	187
jacobian	188
kriging	189
kron	191
L1linreg	192
laguerre	193
lambertWp	194
laplacian	196

lebesgue . . . . .	197
legendre . . . . .	198
linearproj, affineproj . . . . .	199
line_integral . . . . .	202
linprog . . . . .	203
linspace . . . . .	206
logspace . . . . .	207
lsqin . . . . .	208
lsqincon . . . . .	210
lsqnonlin . . . . .	211
lu . . . . .	215
magic . . . . .	217
matlab . . . . .	218
meshgrid . . . . .	218
mexfit . . . . .	219
mldivide . . . . .	221
mod, rem . . . . .	222
Mode . . . . .	223
moler . . . . .	224
movavg . . . . .	225
muller . . . . .	226
nchoosek . . . . .	228
ndims . . . . .	229
nearest_spd . . . . .	230
nelder_mead . . . . .	231
neville . . . . .	233
newmark . . . . .	234
newtonHorner . . . . .	235
newtonInterp . . . . .	237
newtonRaphson . . . . .	238
newtonsys . . . . .	239
nextpow2 . . . . .	240
nnz . . . . .	241
Norm . . . . .	242
normest . . . . .	243
nthroot . . . . .	244
nullspace . . . . .	245
numderiv . . . . .	246
numel . . . . .	247
ode23 . . . . .	248
odregress . . . . .	251
orth . . . . .	252
pade . . . . .	253
pascal . . . . .	255
pchip . . . . .	255
peaks . . . . .	257
perms . . . . .	258
piecewise . . . . .	259

pinv . . . . .	260
plotyy . . . . .	261
poisson2disk . . . . .	262
polar . . . . .	263
Poly . . . . .	264
poly2str . . . . .	265
polyadd . . . . .	266
polyApprox . . . . .	267
polyarea . . . . .	268
polyder . . . . .	270
polyfit,polyfix . . . . .	271
polyint . . . . .	273
polylog . . . . .	273
polymul, polydiv . . . . .	275
polypow . . . . .	276
polytrans, polygcf . . . . .	277
polyval, polyvalm . . . . .	278
pow2 . . . . .	279
ppfit . . . . .	279
ppval . . . . .	281
primes . . . . .	282
procrustes . . . . .	283
psi . . . . .	285
qpspecial, qpsolve . . . . .	286
qrSolve . . . . .	288
quad . . . . .	289
quad2d . . . . .	290
quadcc . . . . .	291
quadgk . . . . .	292
quadgr . . . . .	293
quadinf . . . . .	294
quadl . . . . .	295
quadprog . . . . .	297
quadv . . . . .	299
quiver . . . . .	300
rand . . . . .	301
randcomb . . . . .	303
randortho . . . . .	304
randperm . . . . .	305
Rank . . . . .	306
rat . . . . .	307
ratinterp . . . . .	308
rationalfit . . . . .	309
rectint . . . . .	311
refindall . . . . .	312
regexp . . . . .	313
regexprep . . . . .	314
repmat . . . . .	315

Reshape . . . . .	316
ridders . . . . .	316
rk4, rk4sys . . . . .	319
rkf54 . . . . .	320
rmserr . . . . .	322
romberg . . . . .	323
roots, polyroots . . . . .	324
rosser . . . . .	325
rot90 . . . . .	326
rref . . . . .	327
runge . . . . .	328
savgol . . . . .	329
segm_distance . . . . .	330
segm_intersect . . . . .	331
semilogx,semilogy . . . . .	333
shooting . . . . .	334
shubert . . . . .	335
Si, Ci . . . . .	337
sigmoid . . . . .	338
simpadpt . . . . .	339
simpson2d . . . . .	340
sind,cosd,tand, etc. . . . .	341
size . . . . .	343
softline . . . . .	344
sorting . . . . .	345
sortrows . . . . .	347
spinterp . . . . .	348
sqrtn,rootm . . . . .	350
squareform . . . . .	352
std . . . . .	353
std_err . . . . .	354
steep_descent . . . . .	355
stereographic . . . . .	356
str2num . . . . .	358
strcat . . . . .	359
strcmp . . . . .	360
strfind . . . . .	361
strjust . . . . .	362
strRep . . . . .	363
strTrim . . . . .	364
subspace . . . . .	365
sumalt . . . . .	366
taylor . . . . .	367
tic,toc . . . . .	368
titanium . . . . .	369
Toeplitz . . . . .	370
Trace . . . . .	371
trapz . . . . .	371

tri . . . . .	373
trigApprox . . . . .	374
trigPoly . . . . .	375
triquad . . . . .	376
trisolve . . . . .	378
vander . . . . .	379
vectorfield . . . . .	380
whittaker . . . . .	381
wilkinson . . . . .	382
zeta . . . . .	383

<b>Index</b>	<b>385</b>
--------------	------------

---

pracma-package

*Practical Numerical Math Routines*

---

## Description

This package provides R implementations of more advanced functions in numerical analysis, with a special view on optimization and time series routines. Uses Matlab/Octave function names where appropriate to simplify porting.

Some of these implementations are the result of courses on Scientific Computing (“Wissenschaftliches Rechnen”) and are mostly intended to demonstrate how to implement certain algorithms in R/S. Others are implementations of algorithms found in textbooks.

## Details

The package encompasses functions from all areas of numerical analysis, for example:

- Root finding and minimization of univariate functions, e.g. Newton-Raphson, Brent-Dekker, Fibonacci or ‘golden ratio’ search.
- Handling polynomials, including roots and polynomial fitting, e.g. Laguerre’s and Muller’s methods.
- Interpolation and function approximation, barycentric Lagrange interpolation, Pade and rational interpolation, Chebyshev or trigonometric approximation.
- Some special functions, e.g. Fresnel integrals, Riemann’s Zeta or the complex Gamma function, and Lambert’s  $W$  computed iteratively through Newton’s method.
- Special matrices, e.g. Hankel, Rosser, Wilkinson
- Numerical differentiation and integration, Richardson approach and “complex step” derivatives, adaptive Simpson and Lobatto integration and adaptive Gauss-Kronrod quadrature.
- Solvers for ordinary differential equations and systems, Euler-Heun, classical Runge-Kutta, ode23, or predictor-corrector method such as the Adams-Bashford-Moulton.



- Some functions from number theory, such as primes and prime factorization, extended Euclidean algorithm.
- Sorting routines, e.g. recursive quickstep.
- Several functions for string manipulation and regular search, all wrapped and named similar to their Matlab analogues.

It serves three main goals:

- Collecting R scripts that can be demonstrated in courses on ‘Numerical Analysis’ or ‘Scientific Computing’ using R/S as the chosen programming language.
- Wrapping functions with appropriate Matlab names to simplify porting programs from Matlab or Octave to R.
- Providing an environment in which R can be used as a full-blown numerical computing system.

Besides that, many of these functions could be called in R applications as they do not have comparable counterparts in other R packages (at least at this moment, as far as I know).

All referenced books have been utilized in one way or another. Web links have been provided where reasonable.

## Note

The following 220 functions are emulations of correspondingly named Matlab functions and bear the same signature as their Matlab cousins if possible:

accumarray, acosd, acot, acotd, acoth, acsc, acscd, acsch, and, angle, ans,  
 arrayfun, asec, asecd, asech, asind, atand, atan2d,  
 beep, bernoulli, blank, blkdiag, bsxfun,  
 cart2pol, cart2sph, cd, ceil, circshift, clear, compan, cond, conv,  
 cosd, cot, cotd, coth, cross, csc, cscd, csch, cumtrapz,  
 dblquad, deblank, deconv, deg2rad, detrend, deval, disp, dot,  
 eig, eigint, ellipj, ellipke, eps, erf, erfc, erfcinv, erfcx, erfi, erfinv,  
 errorbar, expint, expm, eye, ezcontour, ezmesh, ezplot, ezpolar, ezsurf,  
 fact, fftshift, figure, findpeaks, findstr, flipdim, fliplr, flipud,  
 fminbnd, fmincon, fminsearch, fminunc, fplot, fprintf, fsolve, fzero,  
 gammainc, gcd, geomean, gmres, gradient,  
 hadamard, hankel, harmmean, hilb, histc, humps, hypot,  
 idivide, ifft, ifftshift, inpolygon, integral, integral2, integral3,  
 interp1, interp2, inv, isempty, isprime,  
 kron,  
 legendre, linprog, linspace, loglog, logm, logseq, logspace, lsqcurvefit,  
 lsqin, lsqnonlin, lsqnonneg, lu,  
 magic, meshgrid, mkpp, mldivide, mod, mrdivide,  
 nchoosek, ndims, nextpow2, nnz, normest, nthroot, null, num2str, numel,  
 ode23, ode23s, ones, or, orth,  
 pascal, pchip, pdist, pdist2, peaks, perms, piecewise, pinv, plotyy,  
 pol2cart, polar, polyfit, polyint, polylog, polyval, pow2, ppval,  
 primes, psi, pwd,  
 quad, quad2d, quadgk, quadl, quadprog, quadv, quiver,

rad2deg, randi, randn, randsample, rat, rats, regexp, regexp, i,  
 regexpr, rem, repmat, roots, rosser, rot90, rref, runge,  
 sec, secd, sech, semilogx, semilogy, sinc, sind, size, sortrows, sph2cart,  
 sqrtm, squareform, std, str2num, strcat, strcmp, strcmpi,  
 strfind, strfindi, strjust, subspace,  
 tand, tic, toc, trapz, tril, trimmean, triplequad, triu,  
 vander, vectorfield, ver,  
 what, who, whos, wilkinson,  
 zeros, zeta

The following Matlab function names have been capitalized in ‘pracma’ to avoid shadowing functions from R base or one of its recommended packages (on request of Bill Venables and because of Brian Ripley’s CRAN policies):

Diag, factos, finds, Fix, Imag, Lcm, Mode, Norm, nullspace (<- null),  
 Poly, Rank, Real, Reshape, strRep, strTrim, Toeplitz, Trace, uniq (<- unique).

To use “ans” instead of “ans()” – as is common practice in Matlab – type (and similar for other Matlab commands):

```
makeActiveBinding("ans", function() .Last.value, .GlobalEnv)
makeActiveBinding("who", who(), .GlobalEnv)
```

### Author(s)

Hans Werner Borchers

Maintainer: Hans W Borchers <hwborchers@googlemail.com>

### References

- Abramowitz, M., and I. A. Stegun (1972). Handbook of Mathematical Functions (with Formulas, Graphs, and Mathematical Tables). Dover, New York. URL: <https://www.math.ubc.ca/~cbm/aands/notes.htm>
- Arndt, J. (2010). Matters Computational: Ideas, Algorithms, Source Code. Springer-Verlag, Berlin Heidelberg Dordrecht. FXT: a library of algorithms: <https://www.jjj.de/fxt/>.
- Cormen, Th. H., Ch. E. Leiserson, and R. L. Rivest (2009). Introduction to Algorithms. Third Edition, The MIT Press, Cambridge, MA.
- Encyclopedia of Mathematics (2012). Editor-in-Chief: Ulf Rehmann. [https://encyclopediaofmath.org/wiki/Main\\_Page](https://encyclopediaofmath.org/wiki/Main_Page).
- Gautschi, W. (1997). Numerical Analysis: An Introduction. Birkhaeuser, Boston.
- Gentle, J. E. (2009). Computational Statistics. Springer Science+Business Media LCC, New York.
- MathWorld.com (2011). Matlab Central: <https://www.mathworks.com/matlabcentral/>.
- NIST: National Institute of Standards and Technology. Olver, F. W. J., et al. (2010). NIST Handbook of Mathematical Functions. Cambridge University Press. Internet: NIST Digital Library of Mathematical Functions, <https://dlmf.nist.gov/>; Guide to Available Mathematical Software, <https://gams.nist.gov/>.
- Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (2007). Numerical Recipes: The Art of Numerical Computing. Third Edition, incl. Numerical Recipes Software, Cambridge University Press, New York. URL: [numerical.recipes/book/book.html](http://numerical.recipes/book/book.html).

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

Skiena, St. S. (2008). The Algorithm Design Manual. Second Edition, Springer-Verlag, London. The Stony Brook Algorithm Repository: <https://algorist.com/algorist.html>.

Stoer, J., and R. Bulirsch (2002). Introduction to Numerical Analysis. Third Edition, Springer-Verlag, New York.

Strang, G. (2007). Computational Science and Engineering. Wellesley-Cambridge Press.

Weisstein, E. W. (2003). CRC Concise Encyclopedia of Mathematics. Second Edition, Chapman & Hall/CRC Press. Wolfram MathWorld: <https://mathworld.wolfram.com/>.

Zhang, S., and J. Jin (1996). Computation of Special Functions. John Wiley & Sons.

### See Also

The R package ‘matlab’ contains some of the basic routines from Matlab, but unfortunately not any of the higher math routines.

### Examples

```
## Not run:
## See examples in the help files for all functions.

## End(Not run)
```

---

abm3pc

*Adams-Bashford-Moulton*

---

### Description

Third-order Adams-Bashford-Moulton predictor-corrector method.

### Usage

```
abm3pc(f, a, b, y0, n = 50, ...)
```

### Arguments

f	function in the differential equation $y' = f(x, y)$ .
a, b	endpoints of the interval.
y0	starting values at point a.
n	the number of steps from a to b.
...	additional parameters to be passed to the function.

### Details

Combined Adams-Bashford and Adams-Moulton (or: multi-step) method of third order with corrections according to the predictor-corrector approach.

**Value**

List with components `x` for grid points between `a` and `b` and `y` a vector `y` the same length as `x`; additionally an error estimation `est.error` that should be looked at with caution.

**Note**

This function serves demonstration purposes only.

**References**

Fausett, L. V. (2007). Applied Numerical Analysis Using Matlab. Second edition, Prentice Hall.

**See Also**

[rk4](#), [ode23](#)

**Examples**

```
## Attempt on a non-stiff equation
# y' = y^2 - y^3, y(0) = d, 0 <= t <= 2/d, d = 0.01
f <- function(t, y) y^2 - y^3
d <- 1/250
abm1 <- abm3pc(f, 0, 2/d, d, n = 1/d)
abm2 <- abm3pc(f, 0, 2/d, d, n = 2/d)
## Not run:
plot(abm1$x, abm1$y, type = "l", col = "blue")
lines(abm2$x, abm2$y, type = "l", col = "red")
grid()
## End(Not run)
```

---

accumarray

*Accumulate Vector Elements*

---

**Description**

`accumarray` groups elements from a data set and applies a function to each group.

**Usage**

```
accumarray(subs, val, sz = NULL, func = sum, fillval = 0)
```

```
uniq(a, first = FALSE)
```

**Arguments**

subs	vector or matrix of positive integers, used as indices for the result vector.
val	numerical vector.
sz	size of the resulting array.
func	function to be applied to a vector of numbers.
fillval	value used to fill the array when there are no indices pointing to that component.
a	numerical vector.
first	logical, shall the first or last element encountered be used.

**Details**

`A <- accumarray(subs, val)` creates an array `A` by accumulating elements of the vector `val` using the lines of `subs` as indices and applying `func` to that accumulated vector. The size of the array can be predetermined by the size vector `sz`.

`A = uniq(a)` returns a vector `b` identical to `unique(a)` and two other vectors of indices `m` and `n` such that `b == a[m]` and `a == b[n]`.

**Value**

`accumarray` returns an array of size the maximum in each column of `subs`, or by `sz`.

`uniq` returns a list with components

<code>b</code>	vector of unique elements of <code>a</code> .
<code>m</code>	vector of indices such that <code>b = a[m]</code>
<code>n</code>	vector of indices such that <code>a = b[n]</code>

**Note**

The Matlab function `accumarray` can also handle sparse matrices.

**See Also**

[unique](#)

**Examples**

```
## Examples for accumarray
val = 101:105
subs = as.matrix(c(1, 2, 4, 2, 4))
accumarray(subs, val)
# [101; 206; 0; 208]

val = 101:105
subs <- matrix(c(1,2,2,2,2, 1,1,3,1,3, 1,2,2,2,2), ncol = 3)
accumarray(subs, val)
# , , 1
# [,1] [,2] [,3]
```

```

# [1,] 101  0  0
# [2,]  0  0  0
# , , 2
# [1,] [2,] [3,]
# [1,]  0  0  0
# [2,] 206  0 208

val = 101:106
subs <- matrix(c(1, 2, 1, 2, 3, 1, 4, 1, 4, 4, 4, 1), ncol = 2, byrow = TRUE)
accumarray(subs, val, func = function(x) sum(diff(x)))
# [1,] [2,] [3,] [4,]
# [1,]  0  1  0  0
# [2,]  0  0  0  0
# [3,]  0  0  0  0
# [4,]  2  0  0  0

val = 101:105
subs = matrix(c(1, 1, 2, 1, 2, 3, 2, 1, 2, 3), ncol = 2, byrow = TRUE)
accumarray(subs, val, sz = c(3, 3), func = max, fillval = NA)
# [1,] [2,] [3,]
# [1,] 101 NA NA
# [2,] 104 NA 105
# [3,] NA NA NA

## Examples for uniq
a <- c(1, 1, 5, 6, 2, 3, 3, 9, 8, 6, 2, 4)
A <- uniq(a); A
# A$b 1 5 6 2 3 9 8 4
# A$m 2 3 10 11 7 8 9 12
# A$n 1 1 2 3 4 5 5 6 7 3 4 8
A <- uniq(a, first = TRUE); A
# A$m 1 3 4 5 6 8 9 12

## Example: Subset sum problem
# Distribution of unique sums among all combinations of a vectors.
allsums <- function(a) {
  S <- c(); C <- c()
  for (k in 1:length(a)) {
    U <- uniq(c(S, a[k], S + a[k]))
    S <- U$b
    C <- accumarray(U$n, c(C, 1, C))
  }
  o <- order(S); S <- S[o]; C <- C[o]
  return(list(S = S, C = C))
}
A <- allsums(seq(1, 9, by=2)); A
# A$S 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25
# A$C 1 1 1 1 1 1 2 2 2 1 2 2 1 2 2 2 1 1 1 1 1 1 1

```

**Description**

The arithmetic-geometric mean of real or complex numbers.

**Usage**

```
agmean(a, b)
```

**Arguments**

a, b                   vectors of real or complex numbers of the same length (or scalars).

**Details**

The arithmetic-geometric mean is defined as the common limit of the two sequences  $a_{n+1} = (a_n + b_n)/2$  and  $b_{n+1} = \sqrt{a_n b_n}$ .

When used for negative or complex numbers, the complex square root function is applied.

**Value**

Returns a list with components: agm a vector of arithmetic-geometric means, component-wise, ni the number of iterations, and prec the overall estimated precision.

**Note**

Gauss discovered that elliptic integrals can be effectively computed via the arithmetic-geometric mean (see example below), for example:

$$\int_0^{\pi/2} \frac{dt}{\sqrt{1 - m^2 \sin^2(t)}} = \frac{(a + b)\pi}{4 \cdot \text{agm}(a, b)}$$

where  $m = (a - b)/(a + b)$

**References**

<https://mathworld.wolfram.com/Arithmetic-GeometricMean.html>

**See Also**

Arithmetic, geometric, and harmonic mean.

**Examples**

```
## Accuracy test: Gauss constant
1/agmean(1, sqrt(2))$agm - 0.834626841674073186 # 1.11e-16 < eps = 2.22e-16

## Gauss' AGM-based computation of \pi
a <- 1.0
b <- 1.0/sqrt(2)
s <- 0.5
d <- 1L
```

```

while (abs(a-b) > eps()) {
  t <- a
  a <- (a + b)*0.5
  b <- sqrt(t*b)
  c <- (a-t)*(a-t)
  d <- 2L * d
  s <- s - d*c
}
approx_pi <- (a+b)^2 / s / 2.0
abs(approx_pi - pi)          # 8.881784e-16 in 4 iterations

## Example: Approximate elliptic integral
N <- 20
m <- seq(0, 1, len = N+1)[1:N]
E <- numeric(N)
for (i in 1:N) {
  f <- function(t) 1/sqrt(1 - m[i]^2 * sin(t)^2)
  E[i] <- quad(f, 0, pi/2)
}
A <- numeric(2*N-1)
a <- 1
b <- a * (1-m) / (m+1)

## Not run:
plot(m, E, main = "Elliptic Integrals vs. arith.-geom. Mean")
lines(m, (a+b)*pi / 4 / agmean(a, b)$agm, col="blue")
grid()
## End(Not run)

```

---

aitken

*Aitken' Method*


---

## Description

Aitken's acceleration method.

## Usage

```
aitken(f, x0, nmax = 12, tol = 1e-8, ...)
```

## Arguments

f	Function with a fixpoint.
x0	Starting value.
nmax	Maximum number of iterations.
tol	Relative tolerance.
...	Additional variables passed to f.



**Details**

Aitken's acceleration method, or delta-squared process, is used for accelerating the rate of convergence of a sequence (from linear to quadratic), here applied to the fixed point iteration scheme of a function.

**Value**

The fixpoint (as found so far).

**Note**

Sometimes used to accerate Newton-Raphson (Steffensen's method).

**References**

Quarteroni, A., and F. Saleri (2006). Scientific Computing with Matlab and Octave. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[lambertWp](#)

**Examples**

```
# Find a zero of    f(x) = cos(x) - x*exp(x)
# as fixpoint of  phi(x) = x + (cos(x) - x*exp(x))/2
phi <- function(x) x + (cos(x) - x*exp(x))/2
aitken(phi, 0)  #=> 0.5177574
```

---

 akimaInterp

*Univariate Akima Interpolation*


---

**Description**

Interpolate smooth curve through given points on a plane.

**Usage**

```
akimaInterp(x, y, xi)
```

**Arguments**

`x, y`            `x/y`-coordinates of (irregular) grid points defining the curve.  
`xi`                `x`-coordinates of points where to interpolate.

**Details**

Implementation of Akima's univariate interpolation method, built from piecewise third order polynomials. There is no need to solve large systems of equations, and the method is therefore computationally very efficient.

**Value**

Returns the interpolated values at the points  $x_i$  as a vector.

**Note**

There is also a 2-dimensional version in package 'akima'.

**Author(s)**

Matlab code by H. Shamsundar under BSC License; re-implementation in R by Hans W Borchers.

**References**

Akima, H. (1970). A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures. *Journal of the ACM*, Vol. 17(4), pp 589-602.

Hyman, J. (1983). Accurate Monotonicity Preserving Cubic Interpolation. *SIAM J. Sci. Stat. Comput.*, Vol. 4(4), pp. 645-654.

Akima, H. (1996). Algorithm 760: Rectangular-Grid-Data Surface Fitting that Has the Accuracy of a Bicubic Polynomial. *ACM TOMS* Vol. 22(3), pp. 357-361.

Akima, H. (1996). Algorithm 761: Scattered-Data Surface Fitting that Has the Accuracy of a Cubic Polynomial. *ACM TOMS*, Vol. 22(3), pp. 362-371.

**See Also**

[kriging](#), `akima::aspline`, `akima::interp`

**Examples**

```
x <- c( 0, 2, 3, 5, 6, 8, 9, 11, 12, 14, 15)
y <- c(10, 10, 10, 10, 10, 10, 10.5, 15, 50, 60, 85)
xs <- seq(12, 14, 0.5)      # 12.0 12.5 13.0 13.5 14.0
ys <- akimaInterp(x, y, xs) # 50.0 54.57405 54.84360 55.19135 60.0
xs; ys

## Not run:
plot(x, y, col="blue", main = "Akima Interpolation")
xi <- linspace(0,15,51)
yi <- akimaInterp(x, y, xi)
lines(xi, yi, col = "darkred")
grid()
## End(Not run)
```

---

and, or                      *Logical AND, OR (Matlab Style)*

---

### Description

and(1, k) resp. or(1, k) the same as (1 & k) + 0 resp. (1 | k) + 0.

### Usage

```
and(1, k)
or(1, k)
```

### Arguments

1, k                      Arrays.

### Details

Performs a logical operation of arrays 1 and k and returns an array containing elements set to either 1 (TRUE) or 0 (FALSE), that is in Matlab style.

### Value

Logical vector.

### Examples

```
A <- matrix(c(0.5, 0.5, 0, 0.75, 0,
             0.5, 0, 0.75, 0.05, 0.85,
             0.35, 0, 0, 0, 0.01,
             0.5, 0.65, 0.65, 0.05, 0), 4, 5, byrow=TRUE)
B <- matrix(c(0, 1, 0, 1, 0,
             1, 1, 1, 0, 1,
             0, 1, 1, 1, 0,
             0, 1, 0, 0, 1), 4, 5, byrow=TRUE)
```

```
and(A, B)
or(A, B)
```

---

`andrewsplot`*Andrews' Curves*

---

**Description**

Plots Andrews' curves in cartesian or polar coordinates.

**Usage**

```
andrewsplot(A, f, style = "pol", scaled = FALSE, npts = 101)
```

**Arguments**

<code>A</code>	numeric matrix with at least two columns.
<code>f</code>	factor or integer vector with <code>nrow(A)</code> elements.
<code>style</code>	character variable, only possible values 'cart' or 'pol'.
<code>scaled</code>	logical; if true scales each column to have mean 0 and standard deviation 1 (not yet implemented).
<code>npts</code>	number of points to plot.

**Details**

`andrewsplot` creates an Andrews plot of the multivariate data in the matrix `A`, assigning different colors according to the factor or integer vector `f`.

Andrews' plot represent each observation (row) by a periodic function over the interval  $[0, 2\pi]$ . This function for the  $i$ -th observation is defined as ...

The plot can be seen in cartesian or polar coordinates — the latter seems appropriate as all these functions are periodic.

**Value**

Generates a plot, no return value.

**Note**

Please note that a different ordering of the columns will result in quite different functions and overall picture.

There are variants utilizing principal component scores, in order of decreasing eigenvalues.

**References**

R. Khattree and D. N. Naik (2002). Andrews PLOTS for Multivariate Data: Some New Suggestions and Applications. *Journal of Statistical Planning and Inference*, Vol. 100, No. 2, pp. 411–425.

**See Also**

[polar](#), `andrews::andrews`

**Examples**

```
## Not run:
data(iris)
s <- sample(1:4, 4)
A <- as.matrix(iris[, s])
f <- as.integer(iris[, 5])
andrewsplot(A, f, style = "pol")

## End(Not run)
```

---

angle

*Basic Complex Functions*

---

**Description**

Basic complex functions (Matlab style)

**Usage**

```
Real(z)
Imag(z)
angle(z)
```

**Arguments**

`z` Vector or matrix of real or complex numbers

**Details**

These are just Matlab names for the corresponding functions in R. The `angle` function is simply defined as `atan2(Im(z), Re(z))`.

**Value**

returning real or complex values; `angle` returns in radians.

**Note**

The true Matlab names are `real`, `imag`, and `conj`, but as `real` was taken in R, all these beginnings are changed to capitals.

The function `Mod` has no special name in Matlab; use `abs()` instead.

**See Also**

[Mod](#), [abs](#)

### Examples

```
z <- c(0, 1, 1+1i, 1i)
Real(z) # Re(z)
Imag(z) # Im(z)
Conj(z) # Conj(z)
abs(z) # Mod(z)
angle(z)
```

---

anms

*Adaptive Nelder-Mead Minimization*

---

### Description

An implementation of the Nelder-Mead algorithm for derivative-free optimization / function minimization.

### Usage

```
anms(fn, x0, ...,
      tol = 1e-10, maxfeval = NULL)
```

### Arguments

fn	nonlinear function to be minimized.
x0	starting vector.
tol	relative tolerance, to be used as stopping rule.
maxfeval	maximum number of function calls.
...	additional arguments to be passed to the function.

### Details

Also called a ‘simplex’ method for finding the local minimum of a function of several variables. The method is a pattern search that compares function values at the vertices of the simplex. The process generates a sequence of simplices with ever reducing sizes.

anms can be used up to 20 or 30 dimensions (then ‘tol’ and ‘maxfeval’ need to be increased). It applies adaptive parameters for simplicial search, depending on the problem dimension – see Fuchang and Lixing (2012).

With upper and/or lower bounds, anms will apply a transformation of bounded to unbounded regions before utilizing Nelder-Mead. Of course, if the optimum is near to the boundary, results will not be as accurate as when the minimum is in the interior.

### Value

List with following components:

xmin	minimum solution found.
fmin	value of f at minimum.
nfeval	number of function calls performed.

**Note**

Copyright (c) 2012 by F. Gao and L. Han, implemented in Matlab with a permissive license. Implemented in R by Hans W. Borchers. For another elaborate implementation of Nelder-Mead see the package ‘dfoptim’.

**References**

Nelder, J., and R. Mead (1965). A simplex method for function minimization. *Computer Journal*, Volume 7, pp. 308-313.

O’Neill, R. (1971). Algorithm AS 47: Function Minimization Using a Simplex Procedure. *Applied Statistics*, Volume 20(3), pp. 338-345.

J. C. Lagarias et al. (1998). Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal for Optimization*, Vol. 9, No. 1, pp 112-147.

Fuchang Gao and Lixing Han (2012). Implementing the Nelder-Mead simplex algorithm with adaptive parameters. *Computational Optimization and Applications*, Vol. 51, No. 1, pp. 259-277.

**See Also**

[optim](#)

**Examples**

```
## Rosenbrock function
rosenbrock <- function(x) {
  n <- length(x)
  x1 <- x[2:n]
  x2 <- x[1:(n-1)]
  sum(100*(x1-x2^2)^2 + (1-x2)^2)
}

anms(rosenbrock, c(0,0,0,0,0))
# $xmin
# [1] 1 1 1 1 1
# $fmin
# [1] 8.268732e-21
# $nfeval
# [1] 1153

# To add constraints to the optimization problem, use a slightly
# modified objective function. Equality constraints not possible.
# Warning: Avoid a starting value too near to the boundary !

## Not run:
# Example: 0.0 <= x <= 0.5
fun <- function(x) {
  if (any(x < 0) || any(x > 0.5)) 100
  else rosenbrock(x)
}
x0 <- rep(0.1, 5)
```

```

anms(fun, x0)
## $xmin
## [1] 0.500000000 0.263051265 0.079972922 0.016228138 0.000267922
## End(Not run)

```

---

approx\_entropy      *Approximate and Sample Entropy*

---

### Description

Calculates the approximate or sample entropy of a time series.

### Usage

```
approx_entropy(ts, edim = 2, r = 0.2*sd(ts), elag = 1)
```

```
sample_entropy(ts, edim = 2, r = 0.2*sd(ts), tau = 1)
```

### Arguments

ts	a time series.
edim	the embedding dimension, as for chaotic time series; a preferred value is 2.
r	filter factor; work on heart rate variability has suggested setting r to be 0.2 times the standard deviation of the data.
elag	embedding lag; defaults to 1, more appropriately it should be set to the smallest lag at which the autocorrelation function of the time series is close to zero. (At the moment it cannot be changed by the user.)
tau	delay time for subsampling, similar to elag.

### Details

Approximate entropy was introduced to quantify the the amount of regularity and the unpredictability of fluctuations in a time series. A low value of the entropy indicates that the time series is deterministic; a high value indicates randomness.

Sample entropy is conceptually similar with the following differences: It does not count self-matching, and it does not depend that much on the length of the time series.

### Value

The approximate, or sample, entropy, a scalar value.

### Note

This code here derives from Matlab versions at Mathwork's File Exchange, "Fast Approximate Entropy" and "Sample Entropy" by Kijoon Lee under BSD license.



## References

- Pincus, S.M. (1991). Approximate entropy as a measure of system complexity. Proc. Natl. Acad. Sci. USA, Vol. 88, pp. 2297–2301.
- Kaplan, D., M. I. Furman, S. M. Pincus, S. M. Ryan, L. A. Lipsitz, and A. L. Goldberger (1991). Aging and the complexity of cardiovascular dynamics, Biophysics Journal, Vol. 59, pp. 945–949.
- Yentes, J.M., N. Hunt, K.K. Schmid, J.P. Kaipust, D. McGrath, N. Stergiou (2012). The Appropriate use of approximate entropy and sample entropy with short data sets. Ann. Biomed. Eng.

## See Also

RHRV::CalculateApEn

## Examples

```
ts <- rep(61:65, 10)
approx_entropy(ts, edim = 2)           # -0.0004610253
sample_entropy(ts, edim = 2)          # 0

set.seed(8237)
approx_entropy(rnorm(500), edim = 2)   # 1.351439 high, random
approx_entropy(sin(seq(1,100,by=0.2)), edim = 2) # 0.171806 low, deterministic
sample_entropy(sin(seq(1,100,by=0.2)), edim = 2) # 0.2359326

## Not run: (Careful: This will take several minutes.)
# generate simulated data
N <- 1000; t <- 0.001*(1:N)
sint <- sin(2*pi*10*t); sd1 <- sd(sint) # sine curve
whitet <- rnorm(N); sd2 <- sd(whitet) # white noise
chirpt <- sint + 0.1*whitet; sd3 <- sd(chirpt) # chirp signal

# calculate approximate entropy
rnum <- 30; result <- zeros(3, rnum)
for (i in 1:rnum) {
  r <- 0.02 * i
  result[1, i] <- approx_entropy(sint, 2, r*sd1)
  result[2, i] <- approx_entropy(chirpt, 2, r*sd2)
  result[3, i] <- approx_entropy(whitet, 2, r*sd3)
}

# plot curves
r <- 0.02 * (1:rnum)
plot(c(0, 0.6), c(0, 2), type="n",
      xlab = "", ylab = "", main = "Approximate Entropy")
points(r, result[1, ], col="red"); lines(r, result[1, ], col="red")
points(r, result[2, ], col="green"); lines(r, result[2, ], col="green")
points(r, result[3, ], col="blue"); lines(r, result[3, ], col="blue")
grid()
## End(Not run)
```

---

arclength

*Arc Length of a Curve*


---

**Description**

Calculates the arc length of a parametrized curve.

**Usage**

```
arclength(f, a, b, nmax = 20, tol = 1e-05, ...)
```

**Arguments**

f	parametrization of a curve in n-dim. space.
a,b	begin and end of the parameter interval.
nmax	maximal number of iterations.
tol	relative tolerance requested.
...	additional arguments to be passed to the function.

**Details**

Calculates the arc length of a parametrized curve in  $R^n$ . It applies Richardson's extrapolation by refining polygon approximations to the curve.

The parametrization of the curve must be vectorized: if  $t \rightarrow F(t)$  is the parametrization,  $F(c(t_1, t_1, \dots))$  must return  $c(F(t_1), F(t_2), \dots)$ .

Can be directly applied to determine the arc length of a one-dimensional function  $f: R \rightarrow R$  by defining  $F$  (if  $f$  is vectorized) as  $F: t \rightarrow c(t, f(t))$ .

**Value**

Returns a list with components `length` the calculated arc length, `niter` the number of iterations, and `rel.err` the relative error generated from the extrapolation.

**Note**

If by chance certain equidistant points of the curve lie on a straight line, the result may be wrong, then use `polylength` below.

**Author(s)**

HwB <hwborchers@googlemail.com>

**See Also**

[poly\\_length](#)

**Examples**

```

## Example: parametrized 3D-curve with t in 0..3*pi
f <- function(t) c(sin(2*t), cos(t), t)
arclength(f, 0, 3*pi)
# $length: 17.22203          # true length 17.222032...

## Example: length of the sine curve
f <- function(t) c(t, sin(t))
arclength(f, 0, pi)        # true length 3.82019...

## Example: Length of an ellipse with axes a = 1 and b = 0.5
# parametrization x = a*cos(t), y = b*sin(t)
a <- 1.0; b <- 0.5
f <- function(t) c(a*cos(t), b*sin(t))
L <- arclength(f, 0, 2*pi, tol = 1e-10)    #=> 4.84422411027
# compare with elliptic integral of the second kind
e <- sqrt(1 - b^2/a^2)                    # ellipticity
L <- 4 * a * ellipke(e^2)$e                #=> 4.84422411027

## Not run:
## Example: oscillating 1-dimensional function (from 0 to 5)
f <- function(x) x * cos(0.1*exp(x)) * sin(0.1*pi*exp(x))
F <- function(t) c(t, f(t))
L <- arclength(F, 0, 5, tol = 1e-12, nmax = 25)
print(L$length, digits = 16)
# [1] 82.81020372882217          # true length 82.810203728822172...

# Split this computation in 10 steps (run time drops from 2 to 0.2 secs)
L <- 0
for (i in 1:10)
  L <- L + arclength(F, (i-1)*0.5, i*0.5, tol = 1e-10)$length
print(L, digits = 16)
# [1] 82.81020372882216

# Alternative calculation of arc length
f1 <- function(x) sqrt(1 + complexstep(f, x)^2)
L1 <- quadgk(f1, 0, 5, tol = 1e-14)
print(L1, digits = 16)
# [1] 82.81020372882216

## End(Not run)

## Not run:
#-- -----
#   Arc-length parametrization of Fermat's spiral
#-- -----
# Fermat's spiral: r = a * sqrt(t)
f <- function(t) 0.25 * sqrt(t) * c(cos(t), sin(t))

t1 <- 0; t2 <- 6*pi
a <- 0; b <- arclength(f, t1, t2)$length
fParam <- function(w) {

```

```

    fct <- function(u) arclength(f, a, u)$length - w
    urt <- uniroot(fct, c(a, 6*pi))
    urt$root
  }

  ts <- linspace(0, 6*pi, 250)
  plot(matrix(f(ts), ncol=2), type='l', col="blue",
    asp=1, xlab="", ylab = "",
    main = "Fermat's Spiral", sub="20 subparts of equal length")

  for (i in seq(0.05, 0.95, by=0.05)) {
    v <- fParam(i*b); fv <- f(v)
    points(fv[1], f(v)[2], col="darkred", pch=20)
  }
  ## End(Not run)

```

---

 arnoldi

*Arnoldi Iteration*


---

## Description

Arnoldi iteration generates an orthonormal basis of the Krylov space and a Hessenberg matrix.

## Usage

```
arnoldi(A, q, m)
```

## Arguments

A	a square n-by-n matrix.
q	a vector of length n.
m	an integer.

## Details

arnoldi(A, q, m) carries out m iterations of the Arnoldi iteration with n-by-n matrix A and starting vector q (which need not have unit 2-norm). For  $m < n$  it produces an n-by-(m+1) matrix Q with orthonormal columns and an (m+1)-by-m upper Hessenberg matrix H such that  $A*Q[, 1:m] = Q[, 1:m]*H[1:m, 1:m] + H[m+1, m]*Q[, m+1]*t(E_m)$ , where  $E_m$  is the m-th column of the m-by-m identity matrix.

## Value

Returns a list with two elements:

Q A matrix of orthonormal columns that generate the Krylov space (A, A q, A<sup>2</sup> q, ...).

H A Hessenberg matrix such that  $A = Q * H * t(Q)$ .

**References**

Nicholas J. Higham (2008). *Functions of Matrices: Theory and Computation*, SIAM, Philadelphia.

**See Also**

[hessenberg](#)

**Examples**

```
A <- matrix(c(-149,  -50,  -154,
              537,  180,  546,
              -27,   -9,  -25), nrow = 3, byrow = TRUE)
a <- arnoldi(A, c(1,0,0))
a
## $Q
##      [,1]      [,2]      [,3]
## [1,]    1 0.0000000 0.0000000
## [2,]    0 0.9987384 -0.0502159
## [3,]    0 -0.0502159 -0.9987384
##
## $H
##      [,1]      [,2]      [,3]
## [1,] -149.0000 -42.20367124 156.316506
## [2,]  537.6783 152.55114875 -554.927153
## [3,]    0.0000  0.07284727  2.448851

a$Q %*% a$H %*% t(a$Q)
##      [,1] [,2] [,3]
## [1,] -149  -50 -154
## [2,]  537  180  546
## [3,]  -27   -9  -25
```

---

barylag

*Barycentric Lagrange Interpolation*

---

**Description**

Barycentric Lagrange interpolation in one dimension.

**Usage**

```
barylag(xi, yi, x)
```

**Arguments**

`xi, yi`            x- and y-coordinates of supporting nodes.  
`x`                 x-coordinates of interpolation points.

**Details**

barylag interpolates the given data using the barycentric Lagrange interpolation formula (vectorized to remove all loops).

**Value**

Values of interpolated data at points  $x$ .

**Note**

Barycentric interpolation is preferred because of its numerical stability.

**References**

Berrut, J.-P., and L. Nick Trefethen (2004). “Barycentric Lagrange Interpolation”. *SIAM Review*, Vol. 46(3), pp.501–517.

**See Also**

Lagrange or Newton interpolation.

**Examples**

```
## Generates an example with plot.
# Input:
# fun --- function that shall be 'approximated'
# a, b --- interval [a, b] to be used for the example
# n --- number of supporting nodes
# m --- number of interpolation points
# Output
# plot of function, interpolation, and nodes
# return value is NULL (invisible)
## Not run:
barycentricExample <- function(fun, a, b, n, m)
{
  xi <- seq(a, b, len=n)
  yi <- fun(xi)
  x <- seq(a, b, len=m)

  y <- barylag(xi, yi, x)
  plot(xi, yi, col="red", xlab="x", ylab="y",
  main="Example of barycentric interpolation")

  lines(x, fun(x), col="yellow", lwd=2)
  lines(x, y, col="darkred")

  grid()
}

barycentricExample(sin, -pi, pi, 11, 101) # good interpolation
barycentricExample(runge, -1, 1, 21, 101) # bad interpolation
```

```
## End(Not run)
```

---

barylag2d

*2-D Barycentric Lagrange Interpolation*

---

### Description

Two-dimensional barycentric Lagrange interpolation.

### Usage

```
barylag2d(F, xn, yn, xf, yf)
```

### Arguments

F	matrix representing values of a function in two dimensions.
xn, yn	x- and y-coordinates of supporting nodes.
xf, yf	x- and y-coordinates of an interpolating grid..

### Details

Well-known Lagrange interpolation using barycentric coordinates, here extended to two dimensions. The function is completely vectorized.

x-coordinates run downwards in F, y-coordinates to the right. That conforms to the usage in image or contour plots, see the example below.

### Value

Matrix of size  $\text{length}(xf)$ -by- $\text{length}(yf)$  giving the interpolated values at all the grid points  $(xf, yf)$ .

### Note

Copyright (c) 2004 Greg von Winckel of a Matlab function under BSD license; translation to R by Hans W Borchers with permission.

### References

Berrut, J.-P., and L. Nick Trefethen (2004). “Barycentric Lagrange Interpolation”. *SIAM Review*, Vol. 46(3), pp.501–517.

### See Also

[interp2](#), [barylag](#)

**Examples**

```
## Example from R-help
xn <- c(4.05, 4.10, 4.15, 4.20, 4.25, 4.30, 4.35)
yn <- c(60.0, 67.5, 75.0, 82.5, 90.0)
foo <- matrix(c(
  -137.8379, -158.8240, -165.4389, -166.4026, -166.2593,
  -152.1720, -167.3145, -171.1368, -170.9200, -170.4605,
  -162.2264, -172.5862, -174.1460, -172.9923, -172.2861,
  -168.7746, -175.2218, -174.9667, -173.0803, -172.1853,
  -172.4453, -175.7163, -174.0223, -171.5739, -170.5384,
  -173.7736, -174.4891, -171.6713, -168.8025, -167.6662,
  -173.2124, -171.8940, -168.2149, -165.0431, -163.8390),
  nrow = 7, ncol = 5, byrow = TRUE)
xf <- c(4.075, 4.1)
yf <- c(63.75, 67.25)
barylag2d(foo, xn, yn, xf, yf)
# -156.7964 -163.1753
# -161.7495 -167.0424

# Find the minimum of the underlying function
bar <- function(xy) barylag2d(foo, xn, yn, xy[1], xy[2])
optim(c(4.25, 67.5), bar) # "Nelder-Mead"
# $par
# 4.230547 68.522747
# $value
# -175.7959

## Not run:
# Image and contour plots
image(xn, yn, foo)
contour(xn, yn, foo, col="white", add = TRUE)
xs <- seq(4.05, 4.35, length.out = 51)
ys <- seq(60.0, 90.0, length.out = 51)
zz <- barylag2d(foo, xn, yn, xs, ys)
contour(xs, ys, zz, nlevels = 20, add = TRUE)
contour(xs, ys, zz, levels=c(-175, -175.5), add = TRUE)
points(4.23, 68.52)
## End(Not run)
```

bernoulli

*Bernoulli Numbers and Polynomials***Description**

The Bernoulli numbers are a sequence of rational numbers that play an important role for the series expansion of hyperbolic functions, in the Euler-MacLaurin formula, or for certain values of Riemann's function at negative integers.

**Usage**

```
bernoulli(n, x)
```



**Arguments**

n	the index, a whole number greater or equal to 0.
x	real number or vector of real numbers; if missing, the Bernoulli numbers will be given, otherwise the polynomial.

**Details**

The calculation of the Bernoulli numbers uses the values of the zeta function at negative integers, i.e.  $B_n = -n \zeta(1 - n)$ . Bernoulli numbers  $B_n$  for odd  $n$  are 0 except  $B_1$  which is set to -0.5 on purpose.

The Bernoulli polynomials can be directly defined as

$$B_n(x) = \sum_{k=0}^n \binom{n}{k} b_{n-k} x^k$$

and it is immediately clear that the Bernoulli numbers are then given as  $B_n = B_n(0)$ .

**Value**

Returns the first  $n+1$  Bernoulli numbers, if  $x$  is missing, or the value of the Bernoulli polynomial at point(s)  $x$ .

**Note**

The definition uses  $B_1 = -1/2$  in accordance with the definition of the Bernoulli polynomials.

**References**

See the entry on Bernoulli numbers in the Wikipedia.

**See Also**

[zeta](#)

**Examples**

```
bernoulli(10)
# 1.00000000 -0.50000000 0.16666667 0.00000000 -0.03333333
# 0.00000000 0.02380952 0.00000000 -0.03333333 0.00000000 0.07575758
#
## Not run:
x1 <- linspace(0.3, 0.7, 2)
y1 <- bernoulli(1, x1)
plot(x1, y1, type='l', col='red', lwd=2,
      xlim=c(0.0, 1.0), ylim=c(-0.2, 0.2),
      xlab="", ylab="", main="Bernoulli Polynomials")
grid()
xs <- linspace(0, 1, 51)
lines(xs, bernoulli(2, xs), col="green", lwd=2)
lines(xs, bernoulli(3, xs), col="blue", lwd=2)
```

```

lines(xs, bernoulli(4, xs), col="cyan", lwd=2)
lines(xs, bernoulli(5, xs), col="brown", lwd=2)
lines(xs, bernoulli(6, xs), col="magenta", lwd=2)
legend(0.75, 0.2, c("B_1", "B_2", "B_3", "B_4", "B_5", "B_6"),
      col=c("red", "green", "blue", "cyan", "brown", "magenta"),
      lty=1, lwd=2)

## End(Not run)

```

---

 bernstein

*Bernstein Polynomials*


---

### Description

Bernstein base polynomials and approximations.

### Usage

```
bernstein(f, n, x)
```

```
bernsteinb(k, n, x)
```

### Arguments

f	function to be approximated by Bernstein polynomials.
k	integer between 0 and n, the k-th Bernstein polynomial of order n.
n	order of the Bernstein polynomial(s).
x	numeric scalar or vector where the Bernstein polynomials will be calculated.

### Details

The Bernstein basis polynomials  $B_{k,n}(x)$  are defined as

$$B_{k,n}(x) = \binom{n}{k} x^k (1-x)^{n-k}$$

and form a basis for the vector space of polynomials of degree  $n$  over the interval  $[0, 1]$ .

`bernstein(f, n, x)` computes the approximation of function `f` through Bernstein polynomials of degree  $n$ , resp. computes the value of this approximation at `x`. The function is vectorized and applies a brute force calculation.

But if `x` is a scalar, the value will be calculated using De Casteljau's algorithm for higher accuracy. For bigger  $n$  the binomial coefficients may be in for problems.

### Value

Returns a scalar or vector of function values.

**References**

See [https://en.wikipedia.org/wiki/Bernstein\\_polynomial](https://en.wikipedia.org/wiki/Bernstein_polynomial)

**Examples**

```
## Example
f <- function(x) sin(2*pi*x)
xs <- linspace(0, 1)
ys <- f(xs)
## Not run:
plot(xs, ys, type='l', col="blue",
     main="Bernstein Polynomials")
grid()
b10 <- bernstein(f, 10, xs)
b100 <- bernstein(f, 100, xs)
lines(xs, b10, col="magenta")
lines(xs, b100, col="red")
## End(Not run)

# Bernstein basis polynomials
## Not run:
xs <- linspace(0, 1)
plot(c(0,1), c(0,1), type='n',
     main="Bernstein Basis Polynomials")
grid()
n = 10
for (i in 0:n) {
  bs <- bernsteinb(i, n, xs)
  lines(xs, bs, col=i+1)
}
## End(Not run)
```

---

bisect

*Rootfinding Through Bisection or Secant Rule*


---

**Description**

Finding roots of univariate functions in bounded intervals.

**Usage**

```
bisect(fun, a, b, maxiter = 500, tol = NA, ...)
```

```
secant(fun, a, b, maxiter = 500, tol = 1e-08, ...)
```

```
regulaFalsi(fun, a, b, maxiter = 500, tol = 1e-08, ...)
```

**Arguments**

fun	Function or its name as a string.
a, b	interval end points.
maxiter	maximum number of iterations; default 100.
tol	absolute tolerance; default $\text{eps}^{(1/2)}$
...	additional arguments passed to the function.

**Details**

“Bisection” is a well known root finding algorithms for real, univariate, continuous functions. Bisection works in any case if the function has opposite signs at the endpoints of the interval.

bisect stops when floating point precision is reached, attaching a tolerance is no longer needed. This version is trimmed for exactness, not speed. Special care is taken when 0.0 is a root of the function. Argument 'tol' is deprecated and not used anymore.

The “Secant rule” uses a succession of roots of secant lines to better approximate a root of a function. “Regula falsi” combines bisection and secant methods. The so-called ‘Illinois’ improvement is used here.

**Value**

Return a list with components `root`, `f.root`, the function value at the found root, `iter`, the number of iterations done, and `root`, and the estimated accuracy `estim.prec`

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[riders](#)

**Examples**

```
bisect(sin, 3.0, 4.0)
# $root      $f.root      $iter  $estim.prec
# 3.1415926536  1.2246467991e-16  52     4.4408920985e-16

bisect(sin, -1.0, 1.0)
# $root      $f.root      $iter  $estim.prec
# 0          0          2      0

# Legendre polynomial of degree 5
lp5 <- c(63, 0, -70, 0, 15, 0)/8
f <- function(x) polyval(lp5, x)
bisect(f, 0.6, 1)      # 0.9061798453    correct to 15 decimals
secant(f, 0.6, 1)     # 0.5384693      different root
regulaFalsi(f, 0.6, 1) # 0.9061798459    correct to 10 decimals
```





**Details**

Generate a block diagonal matrix from A, B, C, .... All the arguments must be numeric and non-empty matrices.

**Value**

a numeric matrix

**Note**

Vectors as input have to be converted to matrices before. Note that `as.matrix(v)` with `v` a vector will generate a column vector; use `matrix(v, nrow=1)` if a row vector is intended.

**See Also**

[Diag](#)

**Examples**

```
a1 <- matrix(c(1,2), 1)
a2 <- as.matrix(c(1,2))
blkdiag(a1, diag(1, 2, 2), a2)
```

---

brentDekker

*Brent-Dekker Root Finding Algorithm*

---

**Description**

Find root of continuous function of one variable.

**Usage**

```
brentDekker(fun, a, b, maxiter = 500, tol = 1e-12, ...)
brent(fun, a, b, maxiter = 500, tol = 1e-12, ...)
```

**Arguments**

<code>fun</code>	function whose root is to be found.
<code>a, b</code>	left and right end points of an interval; function values need to be of different sign at the endpoints.
<code>maxiter</code>	maximum number of iterations.
<code>tol</code>	relative tolerance.
<code>...</code>	additional arguments to be passed to the function.

### Details

brentDekker implements a version of the Brent-Dekker algorithm, a well known root finding algorithms for real, univariate, continuous functions. The Brent-Dekker approach is a clever combination of secant and bisection with quadratic interpolation.

brent is simply an alias for brentDekker.

### Value

brent returns a list with

root	location of the root.
f.root	function value at the root.
f.calls	number of function calls.
estim.prec	estimated relative precision.

### References

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

### See Also

[ridders](#), [newtonRaphson](#)

### Examples

```
# Legendre polynomial of degree 5
lp5 <- c(63, 0, -70, 0, 15, 0)/8
f <- function(x) polyval(lp5, x)
brent(f, 0.6, 1) # 0.9061798459 correct to 12 places
```

---

brown72

*Brownian Motion*

---

### Description

The Brown72 data set represents a fractal Brownian motion with a prescribed Hurst exponent of 0.72.

### Usage

```
data(brown72)
```

### Format

The format is: one column.



**Details**

Estimating the Hurst exponent for a data set provides a measure of whether the data is a pure random walk or has underlying trends. Brownian walks can be generated from a defined Hurst exponent.

**Examples**

```
## Not run:
data(brown72)
plot(brown72, type = "l", col = "blue")
grid()
## End(Not run)
```

---

broyden

*Broyden's Method*


---

**Description**

Broyden's method for the numerical solution of nonlinear systems of  $n$  equations in  $n$  variables.

**Usage**

```
broyden(Ffun, x0, J0 = NULL, ...,
        maxiter = 100, tol = .Machine$double.eps^(1/2))
```

**Arguments**

Ffun	$n$ functions of $n$ variables.
x0	Numeric vector of length $n$ .
J0	Jacobian of the function at $x_0$ .
...	additional parameters passed to the function.
maxiter	Maximum number of iterations.
tol	Tolerance, relative accuracy.

**Details**

F as a function must return a vector of length  $n$ , and accept an  $n$ -dim. vector or column vector as input. F must not be univariate, that is  $n$  must be greater than 1.

Broyden's method computes the Jacobian and its inverse only at the first iteration, and does a rank-one update thereafter, applying the so-called Sherman-Morrison formula that computes the inverse of the sum of an invertible matrix  $A$  and the dyadic product,  $uv'$ , of a column vector  $u$  and a row vector  $v'$ .

**Value**

List with components: zero the best root found so far, fnorm the square root of sum of squares of the values of  $f$ , and niter the number of iterations needed.

**Note**

Applied to a system of  $n$  linear equations it will stop in  $2n$  steps

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[newtonsys](#), [fsolve](#)

**Examples**

```
## Example from Quarteroni & Saleri
F1 <- function(x) c(x[1]^2 + x[2]^2 - 1, sin(pi*x[1]/2) + x[2]^3)
broyden(F1, x0 = c(1, 1))
# zero: 0.4760958 -0.8793934; fnorm: 9.092626e-09; niter: 13

F <- function(x) {
  x1 <- x[1]; x2 <- x[2]; x3 <- x[3]
  as.matrix(c(x1^2 + x2^2 + x3^2 - 1,
              x1^2 + x3^2 - 0.25,
              x1^2 + x2^2 - 4*x3), ncol = 1)
}
x0 <- as.matrix(c(1, 1, 1))
broyden(F, x0)
# zero: 0.4407629 0.8660254 0.2360680; fnorm: 1.34325e-08; niter: 8

## Find the roots of the complex function  $\sin(z)^2 + \sqrt{z} - \log(z)$ 
F2 <- function(x) {
  z <- x[1] + x[2]*1i
  fz <- sin(z)^2 + sqrt(z) - log(z)
  c(Re(fz), Im(fz))
}
broyden(F2, c(1, 1))
# zero 0.2555197 0.8948303 , i.e.  $z_0 = 0.2555 + 0.8948i$ 
# fnorm 7.284374e-10
# niter 13

## Two more problematic examples
F3 <- function(x)
  c(2*x[1] - x[2] - exp(-x[1]), -x[1] + 2*x[2] - exp(-x[2]))
broyden(F3, c(0, 0))
# $zero 0.5671433 0.5671433 # x = exp(-x)

F4 <- function(x) # Dennis Schnabel
  c(x[1]^2 + x[2]^2 - 2, exp(x[1] - 1) + x[2]^3 - 2)
broyden(F4, c(2.0, 0.5), maxiter = 100)
```

**Description**

Apply a binary function elementwise.

**Usage**

```
bsxfun(func, x, y)
```

```
arrayfun(func, ...)
```

**Arguments**

func	function with two or more input parameters.
x, y	two vectors, matrices, or arrays of the same size.
...	list of arrays of the same size.

**Details**

bsxfun applies element-by-element a binary function to two vectors, matrices, or arrays of the same size. For matrices, sweep is used for reasons of speed, otherwise mapply. (For arrays of more than two dimensions this may become very slow.)

arrayfun applies func to each element of the arrays and returns an array of the same size.

**Value**

The result will be a vector or matrix of the same size as x, y.

**Note**

The underlying function mapply can be applied in a more general setting with many function parameters:

```
mapply(f, x, y, z, ...)
```

but the array structure will not be preserved in this case.

**See Also**

[Vectorize](#)

**Examples**

```
X <- matrix(rep(1:10, each = 10), 10, 10)
Y <- t(X)
bsxfun("x", X, Y) # multiplication table

f <- function(x, y) x[1] * y[1] # function not vectorized
A <- matrix(c(2, 3, 5, 7), 2, 2)
B <- matrix(c(11, 13, 17, 19), 2, 2)
arrayfun(f, A, B)
```

---

bulirsch-stoer

*Bulirsch-Stoer Algorithm*


---

**Description**

Bulirsch-Stoer algorithm for solving Ordinary Differential Equations (ODEs) very accurately.

**Usage**

```
bulirsch_stoer(f, t, y0, ..., tol = 1e-07)

midpoint(f, t0, tfinal, y0, tol = 1e-07, kmax = 25)
```

**Arguments**

f	function describing the differential equation $y' = f(t, y)$ .
t	vector of x-values where the values of the ODE function will be computed; needs to be increasingly sorted.
y0	starting values as column vector.
...	additional parameters to be passed to the function.
tol	relative tolerance in the Ricardson extrapolation.
t0, tfinal	start and end point of the interval.
kmax	maximal number of steps in the Richardson extrapolation.

**Details**

The Bulirsch-Stoer algorithm is a well-known method to obtain high-accuracy solutions to ordinary differential equations with reasonable computational efforts. It exploits the midpoint method to get good accuracy in each step.

The (modified) midpoint method computes the values of the dependent variable  $y(t)$  from  $t_0$  to  $t_{\text{final}}$  by a sequence of substeps, applying Richardson extrapolation in each step.

Bulirsch-Stoer and midpoint shall not be used with non-smooth functions or singularities inside the interval. The best way to get intermediate points  $t = (t[1], \dots, t[n])$  may be to call `ode23` or `ode23s` first and use the x-values returned to start `bulirsch_stoer` on.

**Value**

bulirsch\_stoer returns a list with x the grid points input, and y a vector of function values at the se points.

**Note**

Will be extended to become a full-blown Bulirsch-Stoer implementation.

**Author(s)**

Copyright (c) 2014 Hans W Borchers

**References**

J. Stoer and R. Bulirsch (2002). Introduction to Numerical Analysis. Third Edition, Texts in Applied Mathematics 12, Springer Science + Business, LCC, New York.

**See Also**

[ode23](#), [ode23s](#)

**Examples**

```
## Example: y' = -y
f1 <- function(t, y) as.matrix(c(y[2], -y[1]))
y0 <- as.matrix(c(0.0, 1.0))
tt <- linspace(0, pi, 13)
yy <- bulirsch_stoer(f1, tt, c(0.0, 1.0)) # 13 equally-spaced grid points
yy[nrow(yy), 1] # 1.1e-11

## Not run:
S <- ode23(f1, 0, pi, c(0.0, 1.0))
yy <- bulirsch_stoer(f1, S$t, c(0.0, 1.0)) # S$x 13 irregular grid points
yy[nrow(yy), 1] # 2.5e-11
S$y[nrow(S$y), 1] # -7.1e-04

## Example: y' = -200 x y^2 # y(x) = 1 / (1 + 100 x^2)
f2 <- function(t, y) -200 * t * y^2
y0 < 1
tic(); S <- ode23(f2, 0, 1, y0); toc() # 0.002 sec
tic(); yy <- bulirsch_stoer(f2, S$t, y0); toc() # 0.013 sec
## End(Not run)
```

**Description**

Solves boundary value problems of linear second order differential equations.

**Usage**

```
bvp(f, g, h, x, y, n = 50)
```

**Arguments**

f, g, h	functions on the right side of the differential equation. If f, g or h is a scalar instead of a function, it is assumed to be a constant coefficient in the differential equation.
x	x[1], x[2] are the interval borders where the solution shall be computed.
y	boundary conditions such that $y(x[1]) = y[1]$ , $y(x[2]) = y[2]$ .
n	number of intermediate grid points; default 50.

**Details**

Solves the two-point boundary value problem given as a linear differential equation of second order in the form:

$$y'' = f(x)y' + g(x)y + h(x)$$

with the finite element method. The solution  $y(x)$  shall exist on the interval  $[a, b]$  with boundary conditions  $y(a) = y_a$  and  $y(b) = y_b$ .

**Value**

Returns a list `list(xs, ys)` with the grid points `xs` and the values `ys` of the solution at these points, including the boundary points.

**Note**

Uses a tridiagonal equation solver that may be faster than `qr.solve` for large values of `n`.

**References**

Kutz, J. N. (2005). Practical Scientific Computing. Lecture Notes 98195-2420, University of Washington, Seattle.

**See Also**

[shooting](#)

**Examples**

```
## Solve y'' = 2*x/(1+x^2)*y' - 2/(1+x^2) * y + 1
## with y(0) = 1.25 and y(4) = -0.95 on the interval [0, 4]:
f1 <- function(x) 2*x / (1 + x^2)
f2 <- function(x) -2 / (1 + x^2)
f3 <- function(x) rep(1, length(x))    # vectorized constant function 1
x <- c(0.0, 4.0)
y <- c(1.25, -0.95)
sol <- bvp(f1, f2, f3, x, y)
## Not run:
plot(sol$xs, sol$ys, ylim = c(-2, 2),
      xlab = "", ylab = "", main = "Boundary Value Problem")
# The analytic solution is
sfun <- function(x) 1.25 + 0.4860896526*x - 2.25*x^2 +
                  2*x*atan(x) - 1/2 * log(1+x^2) + 1/2 * x^2 * log(1+x^2)
xx <- linspace(0, 4)
yy <- sfun(xx)
lines(xx, yy, col="red")
grid()
## End(Not run)
```

---

cart2sph

*Coordinate Transformations*


---

**Description**

Transforms between cartesian, spherical, polar, and cylindrical coordinate systems in two and three dimensions.

**Usage**

```
cart2sph(xyz)
sph2cart(tpr)
cart2pol(xyz)
pol2cart(prz)
```

**Arguments**

xyz	cartesian coordinates x, y, z as vector or matrix.
tpr	spherical coordinates theta, phi, and r as vector or matrix.
prz	polar coordinates phi, r or cylindrical coordinates phi, r, z as vector or matrix.

**Details**

The spherical coordinate system used here consists of

- theta, azimuth angle relative to the positive x-axis
- phi, elevation angle measured from the reference plane

- r, radial distance. i.e., distance to the origin

The polar angle, measured with respect from the polar axis, is then calculated as  $\pi/2 - \text{phi}$ . Note that this convention differs slightly from spherical coordinates ( $r$ ,  $\theta$ ,  $\phi$ ) as often used in mathematics, where  $\phi$  is the polar angle.

`cart2sph` returns spherical coordinates as ( $\theta$ ,  $\phi$ ,  $r$ ), and `sph2cart` expects them in this sequence.

`cart2pol` returns polar coordinates ( $\phi$ ,  $r$ ) if `length(xyz)==2` and cylindrical coordinates ( $\phi$ ,  $r$ ,  $z$ ) else. `pol2cart` needs them in this sequence and length.

To go from cylindrical to cartesian coordinates, transform to cartesian coordinates first — or write your own function, see the examples.

All transformation functions are vectorized.

### Value

All functions return a (2- or 3-dimensional) vector representing a point in the requested coordinate system, or a matrix with 2 or 3 named columns where is row represents a point. The columns are named accordingly.

### Note

In Matlab these functions accept two or three variables and return two or three values. In R it did not appear appropriate to return coordinates as a list.

These functions should be vectorized in the sense that they accept will accept matrices with number of rows or columns equal to 2 or 3.

### Examples

```
x <- 0.5*cos(pi/6); y <- 0.5*sin(pi/6); z <- sqrt(1 - x^2 - y^2)
(s <-cart2sph(c(x, y, z)))      # 0.5235988 1.0471976 1.0000000
sph2cart(s)                    # 0.4330127 0.2500000 0.8660254

cart2pol(c(1,1))               # 0.7853982 1.4142136
cart2pol(c(1,1,0))             # 0.7853982 1.4142136 0.0000000
pol2cart(c(pi/2, 1))          # 6.123234e-17 1.000000e+00
pol2cart(c(pi/4, 1, 1))       # 0.7071068 0.7071068 1.0000000

## Transform spherical to cylindrical coordinates and vice versa
# sph2cyl <- function(th.ph.r) cart2pol(sph2cart(th.ph.r))
# cyl2sph <- function(phi.r.z) cart2sph(pol2cart(phi.r.z))
```

---

cd, pwd, what

*Directory Functions (Matlab style)*

---

### Description

Displays or changes working directory, or lists files therein.



**Usage**

```
cd(dname)
pwd()

what(dname = getwd())
```

**Arguments**

dname (relative or absolute) directory path.

**Details**

pwd() displays the name of the current directory, and is the same as cd(). cd(dname) changes to directory dname and if successful displays the directory name.

what() lists all files in a directory.

**Value**

Name of the current working directory.

**See Also**

[getwd](#), [setwd](#), [list.files](#)

**Examples**

```
# cd()
# pwd()
# what()
```

---

ceil

*Integer Functions (Matlab Style)*

---

**Description**

Functions for rounding and truncating numeric values towards near integer values.

**Usage**

```
ceil(n)
Fix(n)
```

**Arguments**

n a numeric vector or matrix

**Details**

`ceil()` is an alias for `ceiling()` and rounds to the smallest integer equal to or above  $n$ .

`Fix()` truncates values towards 0 and is an alias for `trunc()`. Uses `m1` prefix to indicate Matlab style.

The corresponding functions `floor()` (rounding to the largest interger equal to or smaller than  $n$ ) and `round()` (rounding to the specified number of digits after the decimal point, default being 0) are already part of R base.

**Value**

integer values

**Examples**

```
x <- c(-1.2, -0.8, 0, 0.5, 1.1, 2.9)
ceil(x)
Fix(x)
```

---

charpoly

*Characteristic Polynomial*

---

**Description**

Computes the characteristic polynomial (and the inverse of the matrix, if requested) using the Faddeew-Leverrier method.

**Usage**

```
charpoly(a, info = FALSE)
```

**Arguments**

`a`                   quadratic matrix; size should not be much larger than 100.  
`info`                 logical; if true, the inverse matrix will also be reported.

**Details**

Computes the characteristic polynomial recursively. In the last step the determinant and the inverse matrix can be determined without any extra cost (if the matrix is not singular).

**Value**

Either the characteristic polynomial as numeric vector, or a list with components `cp`, the characteristic polynomial, `det`, the determinant, and `inv`, the inverse matrix, will be returned.

## References

Hou, S.-H. (1998). Classroom Note: A Simple Proof of the Leverrier–Faddeev Characteristic Polynomial Algorithm, *SIAM Review*, 40(3), pp. 706–709.

## Examples

```
a <- magic(5)
A <- charpoly(a, info = TRUE)
A$cp
roots(A$cp)
A$det
zapsmall(A$inv %*% a)
```

---

chebApprox

*Chebyshev Approximation*

---

## Description

Function approximation through Chebyshev polynomials (of the first kind).

## Usage

```
chebApprox(x, fun, a, b, n)
```

## Arguments

x	Numeric vector of points within interval [a, b].
fun	Function to be approximated.
a, b	Endpoints of the interval.
n	An integer $\geq 0$ .

## Details

Return approximate y-coordinates of points at x by computing the Chebyshev approximation of degree n for fun on the interval [a, b].

## Value

A numeric vector of the same length as x.

## Note

TODO: Evaluate the Chebyshev approximative polynomial by using the Clenshaw recurrence formula. (Not yet vectorized, that's why we still use the Horner scheme.)

**References**

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). Numerical Recipes in C: The Art of Scientific Computing. Second Edition, Cambridge University Press.

**See Also**

[polyApprox](#)

**Examples**

```
# Approximate sin(x) on [-pi, pi] with a polynomial of degree 9 !
# This polynomial has to be beaten:
# P(x) = x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9

# Compare these polynomials
p1 <- rev(c(0, 1, 0, -1/6, 0, 1/120, 0, -1/5040, 0, 1/362880))
p2 <- chebCoeff(sin, -pi, pi, 9)

# Estimate the maximal distance
x <- seq(-pi, pi, length.out = 101)
ys <- sin(x)
yp <- polyval(p1, x)
yc <- chebApprox(x, sin, -pi, pi, 9)
max(abs(ys-yp))           # 0.006925271
max(abs(ys-yc))           # 1.151207e-05

## Not run:
# Plot the corresponding curves
plot(x, ys, type = "l", col = "gray", lwd = 5)
lines(x, yp, col = "navy")
lines(x, yc, col = "red")
grid()
## End(Not run)
```

---

chebCoeff

*Chebyshev Polynomials*

---

**Description**

Chebyshev Coefficients for Chebyshev polynomials of the first kind.

**Usage**

```
chebCoeff(fun, a, b, n)
```

**Arguments**

fun	function to be approximated.
a, b	endpoints of the interval.
n	an integer $\geq 0$ .

**Details**

For a function fun on the interval [a, b] determines the coefficients of the Chebyshev polynomials up to degree n that will approximate the function (in L2 norm).

**Value**

Vector of coefficients for the Chebyshev polynomials, from low to high degrees (see the example).

**Note**

See the “Chebfun Project” <<https://www.chebfun.org/>> by Nick Trefethen.

**References**

Weisstein, Eric W. “Chebyshev Polynomial of the First Kind.” From MathWorld — A Wolfram Web Resource. <https://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>

**See Also**

[chebPoly](#), [chebApprox](#)

**Examples**

```
## Chebyshev coefficients for x^2 + 1
n <- 4
f2 <- function(x) x^2 + 1
cC <- chebCoeff(f2, -1, 1, n) # 3.0  0  0.5  0  0
cC[1] <- cC[1]/2             # correcting the absolute Chebyshev term
                               # i.e. 1.5*T_0 + 0.5*T_2
cP <- chebPoly(n)           # summing up the polynomial coefficients
p <- cC %*% cP              # 0 0 1 0 1
```

---

chebPoly

*Chebyshev Polynomials*

---

**Description**

Chebyshev polynomials and their values.

**Usage**

```
chebPoly(n, x = NULL)
```

**Arguments**

n                    an integer  $\geq 0$ .  
x                    a numeric vector, possibly empty; default NULL.

**Details**

Determines an  $(n+1)$ -by- $(n+1)$ -Matrix of Chebyshev polynomials up to degree  $n$ .

The coefficients of the first  $n$  Chebyshev polynomials are computed using the recursion formula. For computing any values at points the well known Horner schema is applied.

**Value**

If  $x$  is NULL, returns an  $(n+1)$ -by- $(n+1)$  matrix with the coefficients of the first Chebyshev polynomials from 0 to  $n$ , one polynomial per row with coefficients from highest to lowest order.

If  $x$  is a numeric vector, returns the values of the  $n$ -th Chebyshev polynomial at the points of  $x$ .

**Note**

See the “Chebfun Project” <<https://www.chebfun.org/>> by Nick Trefethen.

**References**

Carothers, N. L. (1998). A Short Course on Approximation Theory. Bowling Green State University.

**See Also**

[chebCoeff](#), [chebApprox](#)

**Examples**

```
chebPoly(6)

## Not run:
## Plot 6 Chebyshev Polynomials
plot(0, 0, type="n", xlim=c(-1, 1), ylim=c(-1.2, 1.2),
     main="Chebyshev Polynomials for n=1..6", xlab="x", ylab="y")
grid()
x <- seq(-1, 1, length.out = 101)
for (i in 1:6) {
  y <- chebPoly(i, x)
  lines(x, y, col=i)
}
legend(x = 0.55, y = 1.2, c("n=1", "n=2", "n=3", "n=4", "n=5", "n=6"),
      col = 1:6, lty = 1, bg="whitesmoke", cex = 0.75)

## End(Not run)
```

---

`circlefit`*Fitting a Circle*

---

**Description**

Fitting a circle from points in the plane

**Usage**

```
circlefit(xp, yp)
```

**Arguments**

`xp, yp`            Vectors representing the x and y coordinates of plane points

**Details**

This routine finds an ‘algebraic’ solution based on a linear fit. The value to be minimized is the distance of the given points to the nearest point on the circle along a radius.

**Value**

Returns x- and y-coordinates of the center and the radius as a vector of length 3.

Writes the RMS error of the (radial) distance of the original points to the circle directly onto the console.

**References**

Gander, W., G. H. Golub, and R. Strebler (1994). Fitting of Circles and Ellipses — Least Squares Solutions. ETH Zürich, Technical Report 217, Institut für Wissenschaftliches Rechnen.

**Examples**

```
# set.seed(8421)
n <- 20
w <- 2*pi*runif(n)
xp <- cos(w) + 1 + 0.25 * (runif(n) - 0.5)
yp <- sin(w) + 1 + 0.25 * (runif(n) - 0.5)

circe <- circlefit(xp, yp) #=> 0.9899628 1.0044920 1.0256633
# RMS error: 0.07631986

## Not run:
x0 <- circe[1]; y0 <- circe[2]; r0 <- circe[3]
plot(c(-0.2, 2.2), c(-0.2, 2.2), type="n", asp=1)
grid()
abline(h=0, col="gray"); abline(v=0, col="gray")
points(xp, yp, col="darkred")

w <- seq(0, 2*pi, len=100)
```

```
xx <- r0 * cos(w) + x0
yy <- r0 * sin(w) + y0
lines(xx, yy, col="blue")
## End(Not run)
```

---

clear, who(s), ver      *Clear function (Matlab style)*

---

## Description

List or remove items from workspace, or display system information.

## Usage

```
clear(lst)
ver()

who()
whos()
```

## Arguments

lst                      Character vector of names of variables in the global environment.

## Details

Remove these or all items from the workspace, i.e. the global environment, and freeing up system memory.

who() lists all items on the workspace.

whos() lists all items and their class and size.

ver() displays version and license information for R and all the loaded packages.

## Value

Invisibly NULL.

## See Also

[ls](#), [rm](#), [sessionInfo](#)

## Examples

```
# clear() # DON'T
# who()
# whos()
# ver()
```



---

clenshaw\_curtis      *Clenshaw-Curtis Quadrature Formula*

---

## Description

Clenshaw-Curtis Quadrature Formula

## Usage

```
clenshaw_curtis(f, a = -1, b = 1, n = 1024, ...)
```

## Arguments

f	function, the integrand, without singularities.
a, b	lower and upper limit of the integral; must be finite.
n	Number of Chebyshev nodes to account for.
...	Additional parameters to be passed to the function

## Details

Clenshaw-Curtis quadrature is based on sampling the integrand on Chebyshev points, an operation that can be implemented using the Fast Fourier Transform.

## Value

Numerical scalar, the value of the integral.

## References

Trefethen, L. N. (2008). Is Gauss Quadrature Better Than Clenshaw-Curtis? *SIAM Review*, Vol. 50, No. 1, pp 67–87.

## See Also

[gaussLegendre](#), [gauss\\_kronrod](#)

## Examples

```
## Quadrature with Chebyshev nodes and weights
f <- function(x) sin(x+cos(10*exp(x)))/3
## Not run: ezplot(f, -1, 1, fill = TRUE)
cc <- clenshaw_curtis(f, n = 64) #=> 0.0325036517151 , true error > 1.3e-10
```

---

combs *Generate Combinations*

---

**Description**

Generates all combinations of length  $m$  of a vector  $a$ .

**Usage**

```
combs(a, m)
```

**Arguments**

$a$  numeric vector of some length  $n$   
 $m$  integer with  $0 \leq m \leq n$

**Details**

combs generates combinations of length  $n$  of the elements of the vector  $a$ .

**Value**

matrix representing combinations of the elements of  $a$

**See Also**

[perms](#), [randcomb](#)

**Examples**

```
combs(seq(2, 10, by=2), m = 3)
```

---

compan *Companion Matrix*

---

**Description**

Computes the companion matrix of a real or complex vector.

**Usage**

```
compan(p)
```

**Arguments**

$p$  vector representing a polynomial

**Details**

Computes the companion matrix corresponding to the vector  $p$  with  $-p[2:\text{length}(p)]/p[1]$  as first row.

The eigenvalues of this matrix are the roots of the polynomial.

**Value**

A square matrix of  $\text{length}(p)-1$  rows and columns

**See Also**

[roots](#)

**Examples**

```
p <- c(1, 0, -7, 6)
compan(p)
# 0 7 -6
# 1 0 0
# 0 1 0
```

---

complexstep

*Complex Step Derivatives*

---

**Description**

Complex step derivatives of real-valued functions, including gradients, Jacobians, and Hessians.

**Usage**

```
complexstep(f, x0, h = 1e-20, ...)

grad_csd(f, x0, h = 1e-20, ...)
jacobian_csd(f, x0, h = 1e-20, ...)
hessian_csd(f, x0, h = 1e-20, ...)
laplacian_csd(f, x0, h = 1e-20, ...)
```

**Arguments**

$f$	Function that is to be differentiated.
$x0$	Point at which to differentiate the function.
$h$	Step size to be applied; shall be <i>very</i> small.
$\dots$	Additional variables to be passed to $f$ .

## Details

Complex step derivation is a fast and highly exact way of numerically differentiating a function. If the following conditions are satisfied, there will be no loss of accuracy between computing a function value and computing the derivative at a certain point.

- $f$  must have an analytical (i.e., complex differentiable) continuation into an open neighborhood of  $x_0$ .
- $x_0$  **and**  $f(x_0)$  must be real.
- $h$  is real and *very* small:  $0 < h \ll 1$ .

complexstep handles differentiation of univariate functions, while grad\_csd and jacobian\_csd compute gradients and Jacobians by applying the complex step approach iteratively. Please understand that these functions are not vectorized, but complexstep is.

As complex step cannot be applied twice (the first derivative does not fulfill the conditions), hessian\_csd works differently. For the first derivation, complex step is used, to the one time derived function Richardson's method is applied. The same applies to lapalacian\_csd.

## Value

complexstep( $f$ ,  $x_0$ ) returns the derivative  $f'(x_0)$  of  $f$  at  $x_0$ . The function is vectorized in  $x_0$ .

## Note

This surprising approach can be easily deduced from the complex-analytic Taylor formula.

## Author(s)

HwB <hwborchers@googlemail.com>

## References

Martins, J. R. R. A., P. Sturdza, and J. J. Alonso (2003). The Complex-step Derivative Approximation. ACM Transactions on Mathematical Software, Vol. 29, No. 3, pp. 245–262.

## See Also

[numderiv](#)

## Examples

```
## Example from Martins et al.
f <- function(x) exp(x)/sqrt(sin(x)^3 + cos(x)^3) # derivative at x0 = 1.5
# central diff formula # 4.05342789402801, error 1e-10
# numDeriv::grad(f, 1.5) # 4.05342789388197, error 1e-12 Richardson
# pracma::numderiv # 4.05342789389868, error 5e-14 Richardson
complexstep(f, 1.5) # 4.05342789389862, error 1e-15
# Symbolic calculation: # 4.05342789389862

jacobian_csd(f, 1.5)
```

```

f1 <- function(x) sum(sin(x))
grad_csd(f1, rep(2*pi, 3))
## [1] 1 1 1

laplacian_csd(f1, rep(pi/2, 3))
## [1] -3

f2 <- function(x) c(sin(x[1]) * exp(-x[2]))
hessian_csd(f2, c(0.1, 0.5, 0.9))
##           [,1]      [,2] [,3]
## [1,] -0.06055203 -0.60350053  0
## [2,] -0.60350053  0.06055203  0
## [3,]  0.00000000  0.00000000  0

f3 <- function(u) {
  x <- u[1]; y <- u[2]; z <- u[3]
  matrix(c(exp(x^2+y^2), sin(x+y), sin(x)*cos(y), x^2 - y^2), 2, 2)
}
jacobian_csd(f3, c(1,1,1))
##           [,1]      [,2] [,3]
## [1,]  2.7182818  0.0000000  0
## [2,] -0.4161468 -0.4161468  0
## [3,]  0.2919266 -0.7080734  0
## [4,]  2.0000000 -2.0000000  0

```

---

cond

*Matrix Condition*


---

## Description

Condition number of a matrix.

## Usage

```
cond(M, p = 2)
```

## Arguments

M	Numeric matrix; vectors will be considered as column vectors.
p	Indicates the p-norm. At the moment, norms other than p=2 are not implemented.

## Details

The condition number of a matrix measures the sensitivity of the solution of a system of linear equations to small errors in the data. Values of  $\text{cond}(M)$  and  $\text{cond}(M, p)$  near 1 are indications of a well-conditioned matrix.

**Value**

`cond(M)` returns the 2-norm condition number, the ratio of the largest singular value of  $M$  to the smallest.

`c = cond(M, p)` returns the matrix condition number in  $p$ -norm:

`norm(X, p) * norm(inv(X), p)`.

(Not yet implemented.)

**Note**

Not feasible for large or sparse matrices as `svd(M)` needs to be computed. The Matlab/Octave function `condest` for condition estimation has not been implemented.

**References**

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Philadelphia.

**See Also**

[normest](#), [svd](#)

**Examples**

```
cond(hilb(8))
```

---

conv

*Polynomial Convolution*

---

**Description**

Convolution and polynomial multiplication.

**Usage**

```
conv(x, y)
```

**Arguments**

`x, y`                    real or complex vectors.

**Details**

`r = conv(p, q)` convolves vectors  $p$  and  $q$ . Algebraically, convolution is the same operation as multiplying the polynomials whose coefficients are the elements of  $p$  and  $q$ .

**Value**

Another vector.

**Note**

conv utilizes fast Fourier transformation.

**See Also**

[deconv](#), [polyadd](#)

**Examples**

```
conv(c(1, 1, 1), 1)
conv(c(1, 1, 1), c(0, 0, 1))
conv(c(-0.5, 1, -1), c(0.5, 0, 1))
```

---

cot, csc, sec, etc.      *More Trigonometric Functions*

---

**Description**

More trigonometric functions not available in R.

**Usage**

```
cot(z)
csc(z)
sec(z)
acot(z)
acsc(z)
asec(z)
```

**Arguments**

z                    numeric or complex scalar or vector.

**Details**

The usual trigonometric cotangens, cosecans, and secans functions and their inverses, computed through the other well known – in R – sine, cosine, and tangens functions.

**Value**

Result vector of numeric or complex values.

**Note**

These function names are available in Matlab, that is the reason they have been added to the ‘pracma’ package.

**See Also**

Trigonometric and hyperbolic functions in R.

**Examples**

```

cot(1+1i)      # 0.2176 - 0.8680i
csc(1+1i)      # 0.6215 - 0.3039i
sec(1+1i)      # 0.4983 + 0.5911i
acot(1+1i)     # 0.5536 - 0.4024i
acsc(1+1i)     # 0.4523 - 0.5306i
asec(1+1i)     # 1.1185 + 0.5306i

```

---

cotes

---

*Newton-Cotes Formulas*


---

**Description**

Closed composite Newton-Cotes formulas of degree 2 to 8.

**Usage**

```
cotes(f, a, b, n, nodes, ...)
```

**Arguments**

f	the integrand as function of two variables.
a, b	lower and upper limit of the integral.
n	number of subintervals (grid points).
nodes	number of nodes in the Newton-Cotes formula.
...	additional parameters to be passed to the function.

**Details**

2 to 8 point closed and summed Newton-Cotes numerical integration formulas.

These formulas are called ‘closed’ as they include the endpoints. They are called ‘composite’ insofar as they are combined with a Lagrange interpolation over subintervals.

**Value**

The integral as a scalar.

**Note**

It is generally recommended not to apply Newton-Cotes formula of degrees higher than 6, instead increase the number n of subintervals used.



**Author(s)**

Standard Newton-Cotes formulas can be found in every textbook. Copyright (c) 2005 Greg von Winckel of nicely vectorized Matlab code, available from MatlabCentral, for 2 to 11 grid points. R version by Hans W Borchers, with permission.

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[simpadpt](#), [trapz](#)

**Examples**

```
cotes(sin, 0, pi/2, 20, 2)      # 0.999485905248533
cotes(sin, 0, pi/2, 20, 3)      # 1.000000211546591
cotes(sin, 0, pi/2, 20, 4)      # 1.000000391824184
cotes(sin, 0, pi/2, 20, 5)      # 0.99999999501637
cotes(sin, 0, pi/2, 20, 6)      # 0.99999998927507
cotes(sin, 0, pi/2, 20, 7)      # 1.000000000000363  odd degree is better
cotes(sin, 0, pi/2, 20, 8)      # 1.000000000002231
```

---

coth,csch,sech, etc. *More Hyperbolic Functions*

---

**Description**

More hyperbolic functions not available in R.

**Usage**

```
coth(z)
csch(z)
sech(z)
acoth(z)
acsch(z)
asech(z)
```

**Arguments**

`z` numeric or complex scalar or vector.

**Details**

The usual hyperbolic cotangens, cosecans, and secans functions and their inverses, computed through the other well known – in R – hyperbolic sine, cosine, and tangens functions.

**Value**

Result vector of numeric or complex values.

**Note**

These function names are available in Matlab, that is the reason they have been added to the 'pracma' package.

**See Also**

Trigonometric and hyperbolic functions in R.

**Examples**

```
coth(1+1i)      # 0.8680 - 0.2176i
csch(1+1i)     # 0.3039 - 0.6215i
sech(1+1i)     # 0.4983 - 0.5911i
acoth(1+1i)    # 0.4024 - 0.5536i
acsch(1+1i)    # 0.5306 - 0.4523i
asech(1+1i)    # 0.5306 - 1.1185i
```

---

cranknic

*Crank-Nicolson Method*


---

**Description**

The Crank-Nicolson method for solving ordinary differential equations is a combination of the generic steps of the forward and backward Euler methods.

**Usage**

```
cranknic(f, t0, t1, y0, ..., N = 100)
```

**Arguments**

f	function in the differential equation $y' = f(x, y)$ ; defined as a function $R \times R^m \rightarrow R^m$ , where $m$ is the number of equations.
t0, t1	start and end points of the interval.
y0	starting values as row or column vector; for $m$ equations y0 needs to be a vector of length $m$ .
N	number of steps.
...	Additional parameters to be passed to the function.

**Details**

Adding together forward and backward Euler method in the cranknic method is by finding the root of the function merging these two formulas.

No attempt is made to catch any errors in the root finding functions.

**Value**

List with components `t` for grid (or ‘time’) points between `t0` and `t1`, and `y` an `n`-by-`m` matrix with solution variables in columns, i.e. each row contains one time stamp.

**Note**

This is for demonstration purposes only; for real problems or applications please use `ode23` or `rkf54`.

**References**

Quarteroni, A., and F. Saleri (2006). *Scientific Computing With MATLAB and Octave*. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[ode23](#), [newmark](#)

**Examples**

```
## Newton's example
f <- function(x, y) 1 - 3*x + y + x^2 + x*y
sol100 <- cranknic(f, 0, 1, 0, N = 100)
sol1000 <- cranknic(f, 0, 1, 0, N = 1000)

## Not run:
# Euler's forward approach
feuler <- function(f, t0, t1, y0, n) {
  h <- (t1 - t0)/n; x <- seq(t0, t1, by = h)
  y <- numeric(n+1); y[1] <- y0
  for (i in 1:n) y[i+1] <- y[i] + h * f(x[i], y[i])
  return(list(x = x, y = y))
}

solode <- ode23(f, 0, 1, 0)
soleul <- feuler(f, 0, 1, 0, 100)

plot(soleul$x, soleul$y, type = "l", col = "blue",
     xlab = "", ylab = "", main = "Newton's example")
lines(solode$t, solode$y, col = "gray", lwd = 3)
lines(sol100$t, sol100$y, col = "red")
lines(sol1000$t, sol1000$y, col = "green")
grid()

## System of differential equations
# "Herr und Hund"
fhh <- function(x, y) {
  y1 <- y[1]; y2 <- y[2]
  s <- sqrt(y1^2 + y2^2)
  dy1 <- 0.5 - 0.5*y1/s
  dy2 <- -0.5*y2/s
}
```

```
    return(c(dy1, dy2))
}

sol <- cranknic(fhh, 0, 60, c(0, 10))
plot(sol$y[, 1], sol$y[, 2], type = "l", col = "blue",
      xlab = "", ylab = "", main = "'Herr und Hund'")
grid()
## End(Not run)
```

---

cross

*Vector Cross Product*

---

## Description

Vector or cross product

## Usage

```
cross(x, y)
```

## Arguments

x	numeric vector or matrix
y	numeric vector or matrix

## Details

Computes the cross (or: vector) product of vectors in 3 dimensions. In case of matrices it takes the first dimension of length 3 and computes the cross product between corresponding columns or rows.

The more general cross product of  $n-1$  vectors in  $n$ -dimensional space is realized as `crossn`.

## Value

3-dim. vector if `x` and `y` are vectors, a matrix of 3-dim. vectors if `x` and `y` are matrices themselves.

## See Also

[dot](#), [crossn](#)

## Examples

```
cross(c(1, 2, 3), c(4, 5, 6)) # -3 6 -3
```

---

`crossn`*n-dimensional Vector Cross Product*

---

**Description**

Vector cross product of  $n-1$  vectors in  $n$ -dimensional space

**Usage**

```
crossn(A)
```

**Arguments**

A                    matrix of size  $(n-1) \times n$  where  $n \geq 2$ .

**Details**

The rows of the matrix A are taken as  $(n-1)$  vectors in  $n$ -dimensional space. The cross product generates a vector in this space that is orthogonal to all these rows in A and its length is the volume of the geometric hypercube spanned by the vectors.

**Value**

a vector of length  $n$

**Note**

The 'scalar triple product' in  $R^3$  can be defined as

```
spatproduct <- function(a, b, c) dot(a, crossn(b, c))
```

It represents the volume of the parallelepiped spanned by the three vectors.

**See Also**

[cross](#), [dot](#)

**Examples**

```
A <- matrix(c(1,0,0, 0,1,0), nrow=2, ncol=3, byrow=TRUE)
crossn(A) #=> 0 0 1

x <- c(1.0, 0.0, 0.0)
y <- c(1.0, 0.5, 0.0)
z <- c(0.0, 0.0, 1.0)
identical(dot(x, crossn(rbind(y, z))), det(rbind(x, y, z)))
```

---

`cubicspline`*Interpolating Cubic Spline*

---

**Description**

Computes the natural interpolation cubic spline.

**Usage**

```
cubicspline(x, y, xi = NULL, endp2nd = FALSE, der = c(0, 0))
```

**Arguments**

<code>x, y</code>	x- and y-coordinates of points to be interpolated.
<code>xi</code>	x-coordinates of points at which the interpolation is to be performed.
<code>endp2nd</code>	logical; if true, the derivatives at the endpoints are prescribed by <code>der</code> .
<code>der</code>	a two-components vector prescribing derivatives at endpoints.

**Details**

`cubicspline` computes the values at `xi` of the natural interpolating cubic spline that interpolate the values `y` at the nodes `x`. The derivatives at the endpoints can be prescribed.

**Value**

Returns either the interpolated values at the points `xi` or, if `is.null(xi)`, the piecewise polynomial that represents the spline.

**Note**

From the piecewise polynomial returned one can easily generate the spline function, see the examples.

**References**

Quarteroni, Q., and F. Saleri (2006). *Scientific Computing with Matlab and Octave*. Springer-Verlag Berlin Heidelberg.

**See Also**

[spline](#)

**Examples**

```
## Example: Average temperatures at different latitudes
x <- seq(-55, 65, by = 10)
y <- c(-3.25, -3.37, -3.35, -3.20, -3.12, -3.02, -3.02,
      -3.07, -3.17, -3.32, -3.30, -3.22, -3.10)
xs <- seq(-60, 70, by = 1)

# Generate a function for this
pp <- cubicspline(x, y)
ppfun <- function(xs) ppval(pp, xs)

## Not run:
# Plot with and without endpoint correction
plot(x, y, col = "darkblue",
      xlim = c(-60, 70), ylim = c(-3.5, -2.8),
      xlab = "Latitude", ylab = "Temp. Difference",
      main = "Earth Temperatures per Latitude")
lines(spline(x, y), col = "darkgray")
grid()

ys <- cubicspline(x, y, xs, endp2nd = TRUE) # der = 0 at endpoints
lines(xs, ys, col = "red")
ys <- cubicspline(x, y, xs) # no endpoint condition
lines(xs, ys, col = "darkred")

## End(Not run)
```

---

curvefit

*Parametric Curve Fit*


---

**Description**

Polynomial fitting of parametrized points on 2D curves, also requiring to meet some points exactly.

**Usage**

```
curvefit(u, x, y, n, U = NULL, V = NULL)
```

**Arguments**

u	the parameter vector.
x, y	x-, y-coordinates for each parameter value.
n	order of the polynomials, the same in x- and y-direction.
U	parameter values where points will be fixed.
V	matrix with two columns and length(U) rows; first column contains the x-, the second the y-values of those points kept fixed.

**Details**

This function will attempt to fit two polynomials to parametrized curve points using the linear least squares approach with linear equality constraints in `lsqlin`. The requirement to meet exactly some fixed points is interpreted as a linear equality constraint.

**Value**

Returns a list with 4 components, `xp` and `yp` coordinates of the fitted points, and `px` and `py` the coefficients of the fitting polynomials in x- and y-direction.

**Note**

In the same manner, derivatives/directions could be prescribed at certain points.

**See Also**

[circlefit](#), [lsqlin](#)

**Examples**

```
## Approximating half circle arc with small perturbations
N <- 50
u <- linspace(0, pi, N)
x <- cos(u) + 0.05 * randn(1, N)
y <- sin(u) + 0.05 * randn(1, N)
n <- 8
cfit1 <- curvefit(u, x, y, n)
## Not run:
plot(x, y, col = "darkgray", pch = 19, asp = 1)
xp <- cfit1$xp; yp <- cfit1$yp
lines(xp, yp, col="blue")
grid()
## End(Not run)

## Fix the end points at t = 0 and t = pi
U <- c(0, pi)
V <- matrix(c(1, 0, -1, 0), 2, 2, byrow = TRUE)
cfit2 <- curvefit(u, x, y, n, U, V)
## Not run:
xp <- cfit2$xp; yp <- cfit2$yp
lines(xp, yp, col="red")
## End(Not run)

## Not run:
## Archimedian spiral
n <- 8
u <- linspace(0, 3*pi, 50)
a <- 1.0
x <- as.matrix(a*u*cos(u))
y <- as.matrix(a*u*sin(u))
plot(x, y, type = "p", pch = 19, col = "darkgray", asp = 1)
```



```
lines(x, y, col = "darkgray", lwd = 3)
cfit <- curvefit(u, x, y, n)
px <- c(cfit$px); py <- c(cfit$py)
v <- linspace(0, 3*pi, 200)
xs <- polyval(px, v)
ys <- polyval(py, v)
lines(xs, ys, col = "navy")
grid()
## End(Not run)
```

---

cutpoints

*Find Cutting Points*

---

### Description

Finds cutting points for vector  $s$  of real numbers.

### Usage

```
cutpoints(x, nmax = 8, quant = 0.95)
```

### Arguments

<code>x</code>	vector of real values.
<code>nmax</code>	the maximum number of cutting points to choose
<code>quant</code>	quantile of the gaps to consider for cuts.

### Details

Finds cutting points for vector  $s$  of real numbers, based on the gaps in the values of the vector. The number of cutting points is derived from a quantile of gaps in the values. The user can set a lower limit for this number of gaps.

### Value

Returns a list with components `cutp`, the cutting points selected, and `cutd`, the gap between values of  $x$  at this cutting point.

### Note

Automatically finding cutting points is often requested in Data Mining. If a target attribute is available, Quinlan's C5.0 does a very good job here. Unfortunately, the 'C5.0' package (of the R-Forge project "Rulebased Models") is quite cumbersome to use.

### References

Witten, I. H., and E. Frank (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers, San Francisco.

**See Also**[cut](#)**Examples**

```

N <- 100; x <- sort(runif(N))
cp <- cutpoints(x, 6, 0.9)
n <- length(cp$cutp)

# Print out
nocp <- rle(findInterval(x, c(-Inf, cp$cutp, Inf)))$lengths
cbind(c(-Inf, cp$cutp), c(cp$cutp, Inf), nocp)

# Define a factor from the cutting points
fx <- cut(x, breaks = c(-Inf, cp$cutp, Inf))

## Not run:
# Plot points and cutting points
plot(x, rep(0, N), col="gray", ann = FALSE)
points(cp$cutp, rep(0, n), pch="|", col=2)

# Compare with k-means clustering
km <- kmeans(x, n)
points(x, rep(0, N), col = km$cluster, pch = "+")

## A 2-dimensional example
x <- y <- c()
for (i in 1:9) {
  for (j in 1:9) {
    x <- c(x, i + rnorm(20, 0, 0.2))
    y <- c(y, j + rnorm(20, 0, 0.2))
  }
}
cpx <- cutpoints(x, 8, 0)
cpy <- cutpoints(y, 8, 0)

plot(x, y, pch = 18, col=rgb(0.5,0.5,0.5), axes=FALSE, ann=FALSE)
for (xi in cpx$cutp) abline(v=xi, col=2, lty=2)
for (yi in cpy$cutp) abline(h=yi, col=2, lty=2)

km <- kmeans(cbind(x, y), 81)
points(x, y, col=km$cluster)

## End(Not run)

```

---

dblquad*Double and Triple Integration*

---

**Description**

Numerically evaluate double integral over rectangle.

**Usage**

```
dblquad(f, xa, xb, ya, yb, dim = 2, ...,
        subdivs = 300, tol = .Machine$double.eps^0.5)

triplequad(f, xa, xb, ya, yb, za, zb,
           subdivs = 300, tol = .Machine$double.eps^0.5, ...)
```

**Arguments**

f	function of two variables, the integrand.
xa, xb	left and right endpoint for first variable.
ya, yb	left and right endpoint for second variable.
za, zb	left and right endpoint for third variable.
dim	which variable to integrate first.
subdivs	number of subdivisions to use.
tol	relative tolerance to use in integrate.
...	additional parameters to be passed to the integrand.

**Details**

Function `dblquad` applies the internal single variable integration function `integrate` two times, once for each variable.

Function `triplequad` reduces the problem to `dblquad` by first integrating over the innermost variable.

**Value**

Numerical scalar, the value of the integral.

**See Also**

[integrate](#), [quad2d](#), [simpson2d](#)

**Examples**

```
f1 <- function(x, y) x^2 + y^2
dblquad(f1, -1, 1, -1, 1)      # 2.666666667 , i.e. 8/3 . err = 0

f2 <- function(x, y) y*sin(x)+x*cos(y)
dblquad(f2, pi, 2*pi, 0, pi)  # -9.869604401 , i.e. -pi^2, err = 0

# f3 <- function(x, y) sqrt((1 - (x^2 + y^2)) * (x^2 + y^2 <= 1))
f3 <- function(x, y) sqrt(pmax(0, 1 - (x^2 + y^2)))
dblquad(f3, -1, 1, -1, 1)     # 2.094395124 , i.e. 2/3*pi , err = 2e-8

f4 <- function(x, y, z) y*sin(x)+z*cos(x)
triplequad(f4, 0,pi, 0,1, -1,1) # - 2.0 => -2.220446e-16
```

---

`deconv`*Deconvolution*

---

**Description**

Deconvolution and polynomial division.

**Usage**

```
deconv(b, a)
```

**Arguments**

`b`, `a`            real or complex vectors.

**Details**

`deconv(b,a)` deconvolves vector `a` out of vector `b`. The quotient is returned in vector `q` and the remainder in vector `r` such that  $b = \text{conv}(a, q) + r$ .

If `b` and `a` are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials, and deconvolution is polynomial division.

**Value**

List with elements named `q` and `r`.

**Note**

TODO: Base `deconv` on some `filter1d` function.

**See Also**

[conv](#), [polymul](#)

**Examples**

```
b <- c(10, 40, 100, 160, 170, 120)
a <- c(1, 2, 3, 4)

p <- deconv(b, a)
p$q            #=> 10 20 30
p$r            #=> 0 0 0
```

**Description**

Detect events in solutions of a differential equation.

**Usage**

```
deeve(x, y, yv = 0, idx = NULL)
```

**Arguments**

x	vector of (time) points at which the differential equation has been solved.
y	values of the function(s) that have been computed for the given (time) points.
yv	point or numeric vector at which the solution is wanted.
idx	index of functions whose vales shall be returned.

**Details**

Determines when (in x coordinates) the idx-th solution function will take on the value yv. The interpolation is linear for the moment. For points outside the x interval NA is returned.

**Value**

A (time) point  $x_0$  at which the event happens.

**Note**

The interpolation is linear only for the moment.

**See Also**

[deval](#)

**Examples**

```
## Damped pendulum: y'' = -0.3 y' - sin(y)
# y1 = y, y2 = y': y1' = y2, y2' = -0.3*y2 - sin(y1)
f <- function(t, y) {
  dy1 <- y[2]
  dy2 <- -0.3*y[2] - sin(y[1])
  return(c(dy1, dy2))
}
sol <- rk4sys(f, 0, 10, c(pi/2, 0), 100)
deeve(sol$x, sol$y[,1]) # y1 = 0 : elongation in [sec]
# [1] 2.073507 5.414753 8.650250
# matplot(sol$x, sol$y); grid()
```

deg2rad

*Degrees to Radians*

---

**Description**

Transforms between angles in degrees and radians.

**Usage**

```
deg2rad(deg)
rad2deg(rad)
```

**Arguments**

deg            (array of) angles in degrees.  
rad            (array of) angles in radians.

**Details**

This is a simple calculation back and forth. Note that angles greater than 360 degrees are allowed and will be returned. This may appear incorrect but follows a corresponding discussion on Matlab Central.

**Value**

The angle in degrees or radians.

**Examples**

```
deg2rad(c(0, 10, 20, 30, 40, 50, 60, 70, 80, 90))
rad2deg(seq(-pi/2, pi/2, length = 19))
```

---

detrend*Remove Linear Trends*

---

**Description**

Removes the mean value or (piecewise) linear trend from a vector or from each column of a matrix.

**Usage**

```
detrend(x, tt = 'linear', bp = c())
```

**Arguments**

x	vector or matrix, columns considered as the time series.
tt	trend type, 'constant' or 'linear', default is 'linear'.
bp	break points, indices between 1 and nrow(x).

**Details**

detrend computes the least-squares fit of a straight line (or composite line for piecewise linear trends) to the data and subtracts the resulting function from the data.

To obtain the equation of the straight-line fit, use `polyfit`.

**Value**

removes the mean or (piecewise) linear trend from x and returns it in `y=detrend(x)`, that is `x-y` is the linear trend.

**Note**

Detrending is often used for FFT processing.

**See Also**

[polyfit](#)

**Examples**

```
t <- 1:9
x <- c(0, 2, 0, 4, 4, 4, 0, 2, 0)
x - detrend(x, 'constant')
x - detrend(x, 'linear')

y <- detrend(x, 'linear', 5)
## Not run:
plot(t, x, col="blue")
lines(t, x - y, col="red")
grid()
## End(Not run)
```

---

deval

*Evaluate ODE Solution*


---

**Description**

Evaluate solution of a differential equation solver.

**Usage**

```
deval(x, y, xp, idx = NULL)
```

**Arguments**

x	vector of (time) points at which the differential equation has been solved.
y	values of the function(s) that have been computed for the given (time) points.
xp	point or numeric vector at which the solution is wanted; must be sorted.
idx	index of functions whose vales shall be returned.

**Details**

Determines where the points xp lie within the vector x and interpolates linearly.

**Value**

An length(xp)-by-length(idx) matrix of values at points xp.

**Note**

The interpolation is linear only for the moment.

**See Also**

[deeve](#)

**Examples**

```
## Free fall: v' = -g - cw abs(v)^1.1, cw = 1.6 drag coefficient
f <- function(t, y) -9.81 + 1.6*abs(y)^1.1
sol <- rk4(f, 0, 10, 0, 100)
# speed after 0.5, 1, 1.5, 2 seconds
cbind(c(0.5,1,1.5,2), -deval(sol$x, sol$y, c(0.5, 1, 1.5, 2)))
# 0.5 3.272267 m/s
# 1.0 4.507677
# 1.5 4.953259
# 2.0 5.112068
# plot(sol$x, -sol$y, type="l", col="blue"); grid()
```

---

Diag

*Matrix Diagonal*

---

**Description**

Generate diagonal matrices or return diagonal of a matrix

**Usage**

Diag(x, k = 0)



**Arguments**

x                    vector or matrix  
k                    integer indicating a secondary diagonal

**Details**

If x is a vector, `Diag(x, k)` generates a matrix with x as the (k-th secondary) diagonal.

If x is a matrix, `Diag(x, k)` returns the (k-th secondary) diagonal of x.

The k-th secondary diagonal is above the main diagonal for  $k > 0$  and below the main diagonal for  $k < 0$ .

**Value**

matrix or vector

**Note**

In Matlab/Octave this function is called `diag()` and has a different signature than the corresponding function in R.

**See Also**

[diag](#), [Trace](#)

**Examples**

```
Diag(matrix(1:12,3,4), 1)  
Diag(matrix(1:12,3,4), -1)
```

```
Diag(c(1,5,9), 1)  
Diag(c(1,5,9), -1)
```

---

disp,beep

*Utility functions (Matlab style)*

---

**Description**

Display text or array, or produce beep sound.

**Usage**

```
disp(...)  
beep()
```

**Arguments**

...                    any R object that can be printed.

**Details**

Display text or array, or produces the computer's default beep sound using 'cat' with closing new-line.

**Value**

beep() returns NULL invisibly, disp() displays with newline.

**Examples**

```
disp("Some text, and numbers:", pi, exp(1))
# beep()
```

---

 distmat

*Distance Matrix*


---

**Description**

Computes the Euclidean distance between rows of two matrices.

**Usage**

```
distmat(X, Y)
pdist(X)
pdist2(X, Y)
```

**Arguments**

X                    matrix of some size m x k; vector will be taken as row matrix.  
 Y                    matrix of some size n x k; vector will be taken as row matrix.

**Details**

Computes Euclidean distance between two vectors A and B as:

$$\|A-B\| = \sqrt{ \|A\|^2 + \|B\|^2 - 2*A.B }$$

and vectorizes to rows of two matrices (or vectors).

pdist2 is an alias for distmat, while pdist(X) is the same as distmat(X, X).

**Value**

matrix of size m x n if x is of size m x k and y is of size n x k.

**Note**

If  $a$  is  $m \times r$  and  $b$  is  $n \times r$  then

```
apply(outer(a, t(b), "-"), c(1, 4), function(x) sqrt(sum(diag(x*x))))
```

is the  $m \times n$  matrix of distances between the  $m$  rows of  $a$  and  $n$  rows of  $b$ .

This can be modified as necessary, if one wants to apply distances other than the euclidean.

BUT: The code shown here is 10-100 times faster, utilizing the similarity between Euclidean distance and matrix operations.

**References**

Copyright (c) 1999 Roland Bunschoten for a Matlab version on MatlabCentral under the name `distance.m`. Translated to R by Hans W Borchers.

**See Also**

[dist](#)

**Examples**

```
A <- c(0.0, 0.0)
B <- matrix(c(
  0,0, 1,0, 0,1, 1,1), nrow=4, ncol = 2, byrow = TRUE)
distmat(A, B) #=> 0 1 1 sqrt(2)

X <- matrix(rep(0.5, 5), nrow=1, ncol=5)
Y <- matrix(runif(50), nrow=10, ncol=5)
distmat(X, Y)

# A more vectorized form of distmat:
distmat2 <- function(x, y) {
  sqrt(outer(rowSums(x^2), rowSums(y^2), '+') - tcrossprod(x, 2 * y))
}
```

---

dot

*Scalar Product*

---

**Description**

'dot' or 'scalar' product of vectors or pairwise columns of matrices.

**Usage**

```
dot(x, y)
```

**Arguments**

<code>x</code>	numeric vector or matrix
<code>y</code>	numeric vector or matrix

**Details**

Returns the 'dot' or 'scalar' product of vectors or columns of matrices. Two vectors must be of same length, two matrices must be of the same size. If  $x$  and  $y$  are column or row vectors, their dot product will be computed as if they were simple vectors.

**Value**

A scalar or vector of length the number of columns of  $x$  and  $y$ .

**See Also**

[cross](#)

**Examples**

```
dot(1:5, 1:5) #=> 55
# Length of space diagonal in 3-dim- cube:
sqrt(dot(c(1,1,1), c(1,1,1))) #=> 1.732051
```

---

eig

*Eigenvalue Function (Matlab Style)*

---

**Description**

Eigenvalues of a matrix

**Usage**

```
eig(a)
```

**Arguments**

$a$                       real or complex square matrix

**Details**

Computes the eigenvalues of a square matrix of real or complex numbers, using the R routine `eigen` without computing the eigenvectors.

**Value**

Vector of eigenvalues

**See Also**

[compan](#)

**Examples**

```
eig(matrix(c(1,-1,-1,1), 2, 2)) #=> 2 0
eig(matrix(c(1,1,-1,1), 2, 2)) # complex values
eig(matrix(c(0,1i,-1i,0), 2, 2)) # real values
```

---

**eigjacobi***Jacobi Eigenvalue Method*

---

**Description**

Jacobi's iteration method for eigenvalues and eigenvectors.

**Usage**

```
eigjacobi(A, tol = .Machine$double.eps^(2/3))
```

**Arguments**

A	a real symmetric matrix.
tol	requested tolerance.

**Details**

The Jacobi eigenvalue method repeatedly performs (Givens) transformations until the matrix becomes almost diagonal.

**Value**

Returns a list with components V, a matrix containing the eigenvectors as columns, and D a vector of the eigenvalues.

**Note**

This R implementation works well up to 50x50-matrices.

**References**

Mathews, J. H., and K. D. Fink (2004). Numerical Methods Using Matlab. Fourth edition, Pearson education, Inc., New Jersey.

**See Also**

[eig](#)

**Examples**

```

A <- matrix(c( 1.06, -0.73,  0.77, -0.67,
              -0.73,  2.64,  1.04,  0.72,
              0.77,  1.04,  3.93, -2.14,
              -0.67,  0.72, -2.14,  2.04), 4, 4, byrow = TRUE)

eigjacobi(A)
# $V
#           [,1]      [,2]      [,3]      [,4]
# [1,] 0.87019414 -0.3151209  0.1975473 -0.3231656
# [2,] 0.11138094  0.8661855  0.1178032 -0.4726938
# [3,] 0.07043799  0.1683401  0.8273261  0.5312548
# [4,] 0.47475776  0.3494040 -0.5124734  0.6244140
#
# $D
# [1] 0.66335457 3.39813189 5.58753257 0.02098098

```

---

einsteinF

*Einstein Functions*


---

**Description**

Einstein functions.

**Usage**

```
einsteinF(d, x)
```

**Arguments**

x                    numeric or complex vector.  
d                    parameter to select one of the Einstein functions E1, E2, E3, E4.

**Details**

The Einstein functions are sometimes used for the Planck-Einstein oscillator in one degree of freedom.

The functions are defined as:

$$E1(x) = \frac{x^2 e^x}{(e^x - 1)^2}$$

$$E2(x) = \frac{x}{e^x - 1}$$

$$E3(x) = \ln(1 - e^{-x})$$

$$E4(x) = \frac{x}{e^x - 1} - \ln(1 - e^{-x})$$

E1 has an inflection point as  $x=2.34694130\dots$

**Value**

Numeric/complex scalar or vector.

**Examples**

```
## Not run:
x1 <- seq(-4, 4, length.out = 101)
y1 <- einsteinF(1, x1)
plot(x1, y1, type = "l", col = "red",
      xlab = "", ylab = "", main = "Einstein Function E1(x)")
grid()

y2 <- einsteinF(2, x1)
plot(x1, y2, type = "l", col = "red",
      xlab = "", ylab = "", main = "Einstein Function E2(x)")
grid()

x3 <- seq(0, 5, length.out = 101)
y3 <- einsteinF(3, x3)
plot(x3, y3, type = "l", col = "red",
      xlab = "", ylab = "", main = "Einstein Function E3(x)")
grid()

y4 <- einsteinF(4, x3)
plot(x3, y4, type = "l", col = "red",
      xlab = "", ylab = "", main = "Einstein Function E4(x)")
grid()
## End(Not run)
```

---

ellipke,ellipj

*Elliptic and Jacobi Elliptic Integrals*

---

**Description**

Complete elliptic integrals of the first and second kind, and Jacobi elliptic integrals.

**Usage**

```
ellipke(m, tol = .Machine$double.eps)
```

```
ellipj(u, m, tol = .Machine$double.eps)
```

**Arguments**

**u** numeric vector.  
**m** input vector, all input elements must satisfy  $0 \leq x \leq 1$ .  
**tol** tolerance; default is machine precision.

**Details**

ellipke computes the complete elliptic integrals to accuracy tol, based on the algebraic-geometric mean.

ellipj computes the Jacobi elliptic integrals sn, cn, and dn. For instance, *sn* is the inverse function for

$$u = \int_0^\phi dt / \sqrt{1 - m \sin^2 t}$$

with  $sn(u) = \sin(\phi)$ .

Some definitions of the elliptic functions use the modules *k* instead of the parameter *m*. They are related by  $k^2 = m = \sin^2(a)$  where *a* is the ‘modular angle’.

**Value**

ellipke returns list with two components, *k* the values for the first kind, *e* the values for the second kind.

ellipj returns a list with components the three Jacobi elliptic integrals sn, cn, and dn.

**References**

Abramowitz, M., and I. A. Stegun (1965). Handbook of Mathematical Functions. Dover Publications, New York.

**See Also**

elliptic::sn,cn,dn

**Examples**

```
x <- linspace(0, 1, 20)
ke <- ellipke(x)

## Not run:
plot(x, ke$k, type = "l", col = "darkblue", ylim = c(0, 5),
     main = "Elliptic Integrals")
lines(x, ke$e, col = "darkgreen")
legend( 0.01, 4.5,
       legend = c("Elliptic integral of first kind",
                  "Elliptic integral of second kind"),
       col = c("darkblue", "darkgreen"), lty = 1)
grid()
## End(Not run)

## ellipse circumference with axes a, b
ellipse_cf <- function(a, b) {
  return(4*a*ellipke(1 - (b^2/a^2))$e)
}
print(ellipse_cf(1.0, 0.8), digits = 10)
# [1] 5.672333578
```



```
## Jacobi elliptic integrals
u <- c(0, 1, 2, 3, 4, 5)
m <- seq(0.0, 1.0, by = 0.2)
je <- ellipj(u, m)
# $sn      0.0000  0.8265  0.9851  0.7433  0.4771  0.9999
# $cn      1.0000  0.5630 -0.1720 -0.6690 -0.8789  0.0135
# $dn      1.0000  0.9292  0.7822  0.8176  0.9044  0.0135
je$sn^2 + je$cn^2      # 1 1 1 1 1 1
je$dn^2 + m * je$sn^2  # 1 1 1 1 1 1
```

---

eps

*Floating Point Relative Accuracy*


---

### Description

Distance from 1.0 to the next largest double-precision number.

### Usage

```
eps(x = 1.0)
```

### Arguments

x scalar or numerical vector or matrix.

### Details

$d = \text{eps}(x)$  is the positive distance from  $\text{abs}(x)$  to the next larger floating point number in double precision.

If x is an array,  $\text{eps}(x)$  will return  $\text{eps}(\max(\text{abs}(x)))$ .

### Value

Returns a scalar.

### Examples

```
for (i in -5:5) cat(eps(10^i), "\n")
# 1.694066e-21
# 1.355253e-20
# 2.168404e-19
# 1.734723e-18
# 1.387779e-17
# 2.220446e-16
# 1.776357e-15
# 1.421085e-14
# 1.136868e-13
# 1.818989e-12
# 1.455192e-11
```

---

erf

*Error Functions and Inverses (Matlab Style)*

---

### Description

The error or Phi function is a variant of the cumulative normal (or Gaussian) distribution.

### Usage

erf(x)  
erfinv(y)  
erfc(x)  
erfcinv(y)  
erfcx(x)  
  
erfz(z)  
erfi(z)

### Arguments

x, y                vector of real numbers.  
z                    real or complex number; must be a scalar.

### Details

erf and erfinv are the error and inverse error functions.  
erfc and erfcinv are the complementary error function and its inverse.  
erfcx is the scaled complementary error function.  
erfz is the complex, erfi the imaginary error function.

### Value

Real or complex number(s), the value(s) of the function.

### Note

For the complex error function we used Fortran code from the book S. Zhang & J. Jin “Computation of Special Functions” (Wiley, 1996).

### Author(s)

First version by Hans W Borchers; vectorized version of erfz by Michael Lachmann.

### See Also

[pnorm](#)

**Examples**

```

x <- 1.0
erf(x); 2*pnorm(sqrt(2)*x) - 1
# [1] 0.842700792949715
# [1] 0.842700792949715
erfc(x); 1 - erf(x); 2*pnorm(-sqrt(2)*x)
# [1] 0.157299207050285
# [1] 0.157299207050285
# [1] 0.157299207050285
erfz(x)
# [1] 0.842700792949715
erfi(x)
# [1] 1.650425758797543

```

---

errorbar

*Plot Error Bars*


---

**Description**

Draws symmetric error bars in x- and/or y-direction.

**Usage**

```

errorbar(x, y, xerr = NULL, yerr = NULL,
         bar.col = "red", bar.len = 0.01,
         grid = TRUE, with = TRUE, add = FALSE, ...)

```

**Arguments**

x, y	x-, y-coordinates
xerr, yerr	length of the error bars, relative to the x-, y-values.
bar.col	color of the error bars; default: red
bar.len	length of the cross bars orthogonal to the error bars; default: 0.01.
grid	logical; should the grid be plotted?; default: true
with	logical; whether to end the error bars with small cross bars.
add	logical; should the error bars be added to an existing plot?; default: false.
...	additional plotting parameters that will be passed to the plot function.

**Details**

errorbar plots y versus x with symmetric error bars, with a length determined by xerr resp. yerr in x- and/or y-direction. If xerr or yerr is NULL error bars in this direction will not be drawn.

A future version will allow to draw unsymmetric error bars by specifying upper and lower limits when xerr or yerr is a matrix of size (2 x length(x)).

**Value**

Generates a plot, no return value.

**See Also**

plotrix::plotCI, Hmisc::errbar

**Examples**

```
## Not run:
x <- seq(0, 2*pi, length.out = 20)
y <- sin(x)
xe <- 0.1
ye <- 0.1 * y
errorbar(x, y, xe, ye, type = "l", with = FALSE)

cnt <- round(100*randn(20, 3))
y <- apply(cnt, 1, mean)
e <- apply(cnt, 1, sd)
errorbar(1:20, y, yerr = e, bar.col = "blue")

## End(Not run)
```

---

eta

*Dirichlet Eta Function*

---

**Description**

Dirichlet's eta function valid in the entire complex plane.

**Usage**

```
eta(z)
```

**Arguments**

z                      Real or complex number or a numeric or complex vector.

**Details**

Computes the eta function for complex arguments using a series expansion.

Accuracy is about 13 significant digits for  $abs(z) < 100$ , drops off with higher absolute values.

**Value**

Returns a complex vector of function values.

**Note**

Copyright (c) 2001 Paul Godfrey for a Matlab version available on Mathwork's Matlab Central under BSD license.

**References**

Zhang, Sh., and J. Jin (1996). Computation of Special Functions. Wiley-Interscience, New York.

**See Also**

[gammaz](#), [zeta](#)

**Examples**

```
z <- 0.5 + (1:5)*1i
eta(z)
z <- c(0, 0.5+1i, 1, 1i, 2+2i, -1, -2, -1-1i)
eta(z)
```

---

euler\_heun

*Euler-Heun ODE Solver*


---

**Description**

Euler and Euler-Heun ODE solver.

**Usage**

```
euler_heun(f, a, b, y0, n = 100, improved = TRUE, ...)
```

**Arguments**

f	function in the differential equation $y' = f(x, y)$ .
a, b	start and end points of the interval.
y0	starting value at a.
n	number of grid points.
improved	logical; shall the Heun method be used; default TRUE.
...	additional parameters to be passed to the function.

**Details**

euler\_heun is an integration method for ordinary differential equations using the simple Euler resp. the improved Euler-Heun Method.

**Value**

List with components t for grid (or 'time') points, and y the vector of predicted values at those grid points.

**References**

Quarteroni, A., and F. Saleri (). Scientific Computing with MATLAB and Octave. Second Edition, Springer-Verlag, Berlin Heidelberg, 2006.

**See Also**

[cranknic](#)

**Examples**

```
## Flame-up process
f <- function(x, y) y^2 - y^3
s1 <- cranknic(f, 0, 200, 0.01)
s2 <- euler_heun(f, 0, 200, 0.01)
## Not run:
plot(s1$t, s1$y, type="l", col="blue")
lines(s2$t, s2$y, col="red")
grid()
## End(Not run)
```

---

expint

*Exponential and Logarithmic Integral*

---

**Description**

The exponential integral functions  $E1$  and  $Ei$  and the logarithmic integral  $Li$ .

The exponential integral is defined for  $x > 0$  as

$$\int_x^{\infty} \frac{e^{-t}}{t} dt$$

and by analytic continuation in the complex plane. It can also be defined as the Cauchy principal value of the integral

$$\int_{-\infty}^x \frac{e^t}{t} dt$$

This is denoted as  $Ei(x)$  and the relationship between  $Ei$  and  $\text{expint}(x)$  for  $x$  real,  $x > 0$  is as follows:

$$Ei(x) = -E1(-x) - i\pi$$

The logarithmic integral  $li(x)$  for real  $x$ ,  $x > 0$ , is defined as

$$li(x) = \int_0^x \frac{dt}{\log(t)}$$

and the Eulerian logarithmic integral as  $Li(x) = li(x) - li(2)$ .

The integral  $Li$  approximates the prime number function  $\pi(n)$ , i.e., the number of primes below or equal to  $n$  (see the examples).

**Usage**

```
expint(x)
expint_E1(x)

expint_Ei(x)
li(x)
```

**Arguments**

`x` vector of real or complex numbers.

**Details**

For  $x$  in  $[-38, 2]$  we use a series expansion, otherwise a continued fraction, see the references below, chapter 5.

**Value**

Returns a vector of real or complex numbers, the vectorized exponential integral, resp. the logarithmic integral.

**Note**

The logarithmic integral  $\text{li}(10^i) - \text{li}(2)$  is an approximation of the number of primes below  $10^i$ , i.e.,  $\text{Pi}(10^i)$ , see “?primes”.

**References**

Abramowitz, M., and I.A. Stegun (1965). Handbook of Mathematical Functions. Dover Publications, New York.

**See Also**

`gsl::expint_E1`, `expint_Ei`, [primes](#)

**Examples**

```
expint_E1(1:10)
# 0.2193839 0.0489005 0.0130484 0.0037794 0.0011483
# 0.0003601 0.0001155 0.0000377 0.0000124 0.0000042
expint_Ei(1:10)

## Not run:
estimPi <- function(n) round(Re(li(n) - li(2))) # estimated number of primes
primesPi <- function(n) length(primes(n))      # true number of primes <= n
N <- 1e6
(estimPi(N) - primesPi(N)) / estimPi(N)      # deviation is 0.16 percent!
## End(Not run)
```

---

`expm`*Matrix Exponential*

---

**Description**

Computes the exponential of a matrix.

**Usage**`expm(A, np = 128)``logm(A)`**Arguments**

`A` numeric square matrix.  
`np` number of points to use on the unit circle.

**Details**

For an analytic function  $f$  and a matrix  $A$  the expression  $f(A)$  can be computed by the Cauchy integral

$$f(A) = (2\pi i)^{-1} \int_G (zI - A)^{-1} f(z) dz$$

where  $G$  is a closed contour around the eigenvalues of  $A$ .

Here this is achieved by taking  $G$  to be a circle and approximating the integral by the trapezoid rule.

`logm` is a fake at the moment as it computes the matrix logarithm through taking the logarithm of its eigenvalues; will be replaced by an approach using Pade interpolation.

Another more accurate and more reliable approach for computing these functions can be found in the R package ‘`expm`’.

**Value**

Matrix of the same size as  $A$ .

**Note**

This approach could be used for other analytic functions, but a point to consider is which branch to take (e.g., for the `logm` function).

**Author(s)**

Idea and Matlab code for a cubic root by Nick Trefethen in his “10 digits 1 page” project.



## References

Moler, C., and Ch. Van Loan (2003). Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. *SIAM Review*, Vol. 1, No. 1, pp. 1–46.

N. J. Higham (2008). *Matrix Functions: Theory and Computation*. SIAM Society for Industrial and Applied Mathematics.

## See Also

`expm::expm`

## Examples

```
## The Ward test cases described in the help for expm::expm agree up to
## 10 digits with the values here and with results from Matlab's expm !
A <- matrix(c(-49, -64, 24, 31), 2, 2)
expm(A)
# -0.7357588 0.5518191
# -1.4715176 1.1036382

A1 <- matrix(c(10, 7, 8, 7,
              7, 5, 6, 5,
              8, 6, 10, 9,
              7, 5, 9, 10), nrow = 4, ncol = 4, byrow = TRUE)
expm(logm(A1))
logm(expm(A1))

## System of linear differential equations: y' = M y (y = c(y1, y2, y3))
M <- matrix(c(2,-1,1, 0,3,-1, 2,1,3), 3, 3, byrow=TRUE)
M
C1 <- 0.5; C2 <- 1.0; C3 <- 1.5
t <- 2.0; Mt <- expm(t * M)
yt <- Mt
```

---

eye

*Some Basic Matrices*

---

## Description

Create basic matrices.

## Usage

```
eye(n, m = n)
ones(n, m = n)
zeros(n, m = n)
```

## Arguments

`m, n` numeric scalars specifying size of the matrix

**Value**

Matrix of size  $n \times m$ . Defaults to a square matrix if  $m$  is missing.

No dropping of dimensions; if  $n = 1$ , still returns a matrix and not a vector.

**See Also**

[Diag](#),

**Examples**

```
eye(3)
ones(3, 1)
zeros(1, 3)
```

---

ezcontour,ezsurf,ezmesh

*Contour, Surface, and Mesh Plotter*

---

**Description**

Easy-to-use contour and 3-D surface resp mesh plotter.

**Usage**

```
ezcontour(f, xlim = c(-pi,pi), ylim = c(-pi,pi),
          n = 60, filled = FALSE, col = NULL)
```

```
ezsurf(f, xlim = c(-pi, pi), ylim = c(-pi, pi),
        n = 60, ...)
```

```
ezmesh(f, xlim = c(-pi,pi), ylim = c(-pi,pi),
        n = 60, ...)
```

**Arguments**

<code>f</code>	2-D function to be plotted, must accept $(x,y)$ as a vector.
<code>xlim, ylim</code>	defines x- and y-ranges as intervals.
<code>n</code>	number of grid points in each direction.
<code>col</code>	colour of isolines lines, resp. the surface color.
<code>filled</code>	logical; shall the contour plot be
<code>...</code>	parameters to be passed to the persp function.

**Details**

ezcontour generates a contour plot of the function  $f$  using contour (and image if filled=TRUE is chosen). If filled=TRUE is chosen, col should be a color scheme, the default is heat.colors(12).

ezsurf resp. ezmesh generates a surface/mesh plot of the function  $f$  using persp.

The function  $f$  needs not be vectorized in any form.

**Value**

Plots the function graph and invisibly returns NULL.

**Note**

Mimicks Matlab functions of the same names; Matlab's ezcontourf can be generated with filled=TRUE.

**See Also**

[contour](#), [image](#), [persp](#)

**Examples**

```
## Not run:
f <- function(xy) {
  x <- xy[1]; y <- xy[2]
  3*(1-x)^2 * exp(-(x^2) - (y+1)^2) -
  10*(x/5 - x^3 - y^5) * exp(-x^2 - y^2) -
  1/3 * exp(-(x+1)^2 - y^2)
}
ezcontour(f, col = "navy")
ezcontour(f, filled = TRUE)
ezmesh(f)
ezmesh(f, col="lightblue", theta = -15, phi = 30)

## End(Not run)
```

---

 ezplot

*Easy Function Plot*


---

**Description**

Easy function plot w/o the need to define x, y coordinates.

**Usage**

```
fplot(f, interval, ...)
```

```
ezplot( f, a, b, n = 101, col = "blue", add = FALSE,
  lty = 1, lwd = 1, marker = 0, pch = 1,
  grid = TRUE, gridcol = "gray",
```

```
fill = FALSE, fillcol = "lightgray",
xlab = "x", ylab = "f (x)", main = "Function Plot", ...)
```

### Arguments

f	Function to be plotted.
interval	interval [a, b] to plot the function in
a, b	Left and right endpoint for the plot.
n	Number of points to plot.
col	Color of the function graph.
add	logical; shall the plot be added to an existing plot.
lty	line type; default 1.
lwd	line width; default 1.
marker	no. of markers to be added to the curve; default: none.
pch	point character; default circle.
grid	Logical; shall a grid be plotted?; default TRUE.
gridcol	Color of grid points.
fill	Logical; shall the area between function and axis be filled?; default: FALSE.
fillcol	Color of fill area.
xlab	Label on the x-axis.
ylab	Label on the y-axis.
main	Title of the plot
...	More parameters to be passed to plot.

### Details

Calculates the x, y coordinates of points to be plotted and calls the plot function.

If fill is TRUE, also calls the polygon function with the x, y coordinates in appropriate order.

If the no. of markers is greater than 2, this number of markers will be added to the curve, with equal distances measured along the curve.

### Value

Plots the function graph and invisibly returns NULL.

### Note

fplot is almost an alias for ezplot as all ez... will be replaced by MATLAB with function names f... in 2017.

ezplot should mimick the Matlab function of the same name, has more functionality, misses the possibility to plot several functions.

**See Also**[curve](#)**Examples**

```
## Not run:  
fun <- function(x) x * cos(0.1*exp(x)) * sin(0.1*pi*exp(x))  
ezplot(fun, 0, 5, n = 1001, fill = TRUE)  
  
## End(Not run)
```

---

ezpolar

*Easy Polar Plot*

---

**Description**

Easy function plot w/o the need to define x, y coordinates.

**Usage**

```
ezpolar(fun, interv = c(0, 2*pi))
```

**Arguments**

fun	function to be plotted.
interv	left and right endpoint for the plot.

**Details**

Calculates the x, y coordinates of points to be plotted and calls the polar function.

**Value**

Plots the function graph and invisibly returns NULL.

**Note**

Mimick the Matlab function of the same name.

**See Also**[ezplot](#)**Examples**

```
## Not run:  
fun <- function(x) 1 + cos(x)  
ezpolar(fun)  
  
## End(Not run)
```

---

fact	<i>Factorial Function</i>
------	---------------------------

---

**Description**

Factorial for non-negative integers  $n \leq 170$ .

**Usage**

```
fact(n)
```

```
factorial2(n)
```

**Arguments**

`n`                      Vector of integers, for `fact`, resp. a single integer for `factorial2`.

**Details**

The factorial is computed by brute force; factorials for  $n \geq 171$  are not representable as ‘double’ anymore.

**Value**

`fact` returns the factorial of each element in `n`. If  $n < 0$  the value is `NaN`, and for  $n > 170$  it is `Inf`. Non-integers will be reduced to integers through `floor(n)`.

`factorial2` returns the product of all even resp. odd integers, depending on whether `n` is even or odd.

**Note**

The R core function `factorial` uses the gamma function, whose implementation is not accurate enough for larger input values.

**See Also**

[factorial](#)

**Examples**

```
fact(c(-1, 0, 1, NA, 171)) #=> NaN 1 1 NA Inf
fact(100)                  #=> 9.332621544394410e+157
factorial(100)            #=> 9.332621544394225e+157
# correct value:          9.332621544394415e+157
# Stirling's approximation: 9.324847625269420e+157
# n! ~ sqrt(2*pi*n) * (n/e)^n

factorial2(8); factorial2(9); factorial2(10) # 384 945 3840
factorial(10) / factorial2(10)                # => factorial2(9)
```

---

factors

*Prime Factors*

---

## Description

Returns a vector containing the prime factors of  $n$ .

## Usage

```
factors(n)
```

## Arguments

$n$                       nonnegative integer

## Details

Computes the prime factors of  $n$  in ascending order, each one as often as its multiplicity requires, such that  $n == \text{prod}(\text{factors}(n))$ .

The corresponding Matlab function is called ‘factor’, but because factors have a special meaning in R and the `factor()` function in R could not (or should not) be shadowed, the number theoretic function has been renamed here.

## Value

Vector containing the prime factors of  $n$ .

## See Also

[isprime](#), [primes](#)

## Examples

```
## Not run:
factors(1002001)        # 7 7 11 11 13 13
factors(65537)        # is prime
# Euler's calculation
factors(2^32 + 1)     # 641 6700417
## End(Not run)
```

fderiv

*Numerical Differentiation***Description**

Numerical function differentiation for orders  $n=1 \dots 4$  using finite difference approximations.

**Usage**

```
fderiv(f, x, n = 1, h = 0,
      method = c("central", "forward", "backward"), ...)
```

**Arguments**

f	function to be differentiated.
x	point(s) where differentiation will take place.
n	order of derivative, should only be between 1 and 8; for $n=0$ function values will be returned.
h	step size: if $h=0$ step size will be set automatically.
method	one of "central", "forward", or "backward".
...	more variables to be passed to function f.

**Details**

Derivatives are computed applying central difference formulas that stem from the Taylor series approximation. These formulas have a convergence rate of  $O(h^2)$ .

Use the 'forward' (right side) or 'backward' (left side) method if the function can only be computed or is only defined on one side. Otherwise, always use the central difference formulas.

Optimal step sizes depend on the accuracy the function can be computed with. Assuming internal functions with an accuracy  $2.2e-16$ , appropriate step sizes might be  $5e-6$ ,  $1e-4$ ,  $5e-4$ ,  $2.5e-3$  for  $n=1, \dots, 4$  and precisions of about  $10^{-10}$ ,  $10^{-8}$ ,  $5 \times 10^{-7}$ ,  $5 \times 10^{-6}$  (at best).

For  $n > 4$  a recursion (or finite difference) formula will be applied, cd. the Wikipedia article on "finite difference".

**Value**

Vector of the same length as x.

**Note**

Numerical differentiation suffers from the conflict between round-off and truncation errors.

**References**

Kiusalaas, J. (2005). Numerical Methods in Engineering with Matlab. Cambridge University Press.



**See Also**

[numderiv](#), [taylor](#)

**Examples**

```
## Not run:
f <- sin
xs <- seq(-pi, pi, length.out = 100)
ys <- f(xs)
y1 <- fderiv(f, xs, n = 1, method = "backward")
y2 <- fderiv(f, xs, n = 2, method = "backward")
y3 <- fderiv(f, xs, n = 3, method = "backward")
y4 <- fderiv(f, xs, n = 4, method = "backward")
plot(xs, ys, type = "l", col = "gray", lwd = 2,
      xlab = "", ylab = "", main = "Sinus and its Derivatives")
lines(xs, y1, col=1, lty=2)
lines(xs, y2, col=2, lty=3)
lines(xs, y3, col=3, lty=4)
lines(xs, y4, col=4, lty=5)
grid()
## End(Not run)
```

---

fibsearch

*Fibonacci Search*


---

**Description**

Fibonacci search for function minimum.

**Usage**

```
fibsearch(f, a, b, ..., endp = FALSE, tol = .Machine$double.eps^(1/2))
```

**Arguments**

f	Function or its name as a string.
a, b	endpoints of the interval
endp	logical; shall the endpoints be considered as possible minima?
tol	absolute tolerance; default $\text{eps}^{(1/2)}$ .
...	Additional arguments to be passed to f.

**Details**

Fibonacci search for a univariate function minimum in a bounded interval.

**Value**

Return a list with components `xmin`, `fmin`, the function value at the minimum, `niter`, the number of iterations done, and the estimated precision `estim.prec`

**See Also**[uniroot](#)**Examples**

```
f <- function(x) x * cos(0.1*exp(x)) * sin(0.1*pi*exp(x))
fibsearch(f, 0, 4, tol=10^-10) # $xmin = 3.24848329403424
optimize(f, c(0,4), tol=10^-10) # $minimum = 3.24848328971188
```

---

**figure***Control Plot Devices (Matlab Style)*

---

**Description**

Open, activate, and close graphics devices.

**Usage**

```
figure(figno, title = "")
```

**Arguments**

**figno** (single) number of plot device.  
**title** title of the plot device; not yet used.

**Details**

The number of a graphics device cannot be 0 or 1. The function will work for the operating systems Mac OS, MS Windows, and most Linux systems.

If **figno** is negative and a graphics device with that number does exist, it will be closed.

**Value**

No return value, except when a device of that number does not exist, in which case it returns a list of numbers of open graphics devices.

**Note**

Does not bring the activated graphics device in front.

**See Also**

[dev.set](#), [dev.off](#), [dev.list](#)

**Examples**

```
## Not run:  
figure()  
figure(-2)  
  
## End(Not run)
```

---

findintervals	<i>Find Interval Indices</i>
---------------	------------------------------

---

**Description**

Find indices  $i$  in vector  $xs$  such that either  $x=xs[i]$  or such that  $xs[i]<x<xs[i+1]$  or  $xs[i]>x>xs[i+1]$ .

**Usage**

```
findintervals(x, xs)
```

**Arguments**

$x$	single number.
$xs$	numeric vector, not necessarily sorted.

**Details**

Contrary to `findInterval`, the vector  $xs$  in `findintervals` need not be sorted.

**Value**

Vector of indices in  $1..length(xs)$ . If none is found, returns `integer(0)`.

**Note**

If  $x$  is equal to the last element in  $xs$ , the index `length(xs)` will also be returned.

**Examples**

```
xs <- zapsmall(sin(seq(0, 10*pi, len=100)))  
findintervals(0, xs)  
# 1 10 20 30 40 50 60 70 80 90 100
```

findmins

*Find All Minima***Description**

Finding all local(!) minima of a univariate function in an interval by splitting the interval in many small subintervals.

**Usage**

```
findmins(f, a, b, n = 100, tol = .Machine$double.eps^(2/3), ...)
```

**Arguments**

f	functions whose minima shall be found.
a, b	endpoints of the interval.
n	number of subintervals to generate and search.
tol	has no effect at this moment.
...	Additional parameters to be passed to the function.

**Details**

Local minima are found by looking for one minimum in each subinterval. It will be found by applying `optimize` to any two adjacent subinterval where the first slope is negative and the second one positive.

If the function is minimal on a whole subinterval, this will cause problems. If some minima are apparently not found, increase the number of subintervals.

Note that the endpoints of the interval will never be considered to be local minima. The function need not be vectorized.

**Value**

Numeric vector with the x-positions of all minima found in the interval.

**See Also**

[optimize](#)

**Examples**

```
fun <- function(x) x * cos(0.1*exp(x)) * sin(0.1*pi*exp(x))
## Not run: ezplot(fun, 0, 5, n = 1001)

# If n is smaller, the rightmost minimum will not be found.
findmins(fun, 0, 5, n= 1000)
# 2.537727 3.248481 3.761840 4.023021 4.295831
# 4.455115 4.641481 4.756263 4.897461 4.987802
```

---

 findpeaks

*Find Peaks*


---

### Description

Find peaks (maxima) in a time series.

### Usage

```
findpeaks(x, nups = 1, ndowns = nups, zero = "0", peakpat = NULL,
          minpeakheight = -Inf, minpeakdistance = 1,
          threshold = 0, npeaks = 0, sortstr = FALSE)
```

### Arguments

x	numerical vector taken as a time series (no NAs allowed)
nups	minimum number of increasing steps before a peak is reached
ndowns	minimum number of decreasing steps after the peak
zero	can be '+', '-', or '0'; how to interpret succeeding steps of the same value: increasing, decreasing, or special
peakpat	define a peak as a regular pattern, such as the default pattern <code>[+]{1,}[ -]{1,}</code> ; if a pattern is provided, parameters <code>nups</code> and <code>ndowns</code> are not taken into account
minpeakheight	the minimum (absolute) height a peak has to have to be recognized as such
minpeakdistance	the minimum distance (in indices) peaks have to have to be counted
threshold	the minimum
npeaks	the number of peaks to return
sortstr	logical; should the peaks be returned sorted in decreasing order of their maximum value

### Details

This function is quite general as it relies on regular patterns to determine where a peak is located, from beginning to end.

### Value

Returns a matrix where each row represents one peak found. The first column gives the height, the second the position/index where the maximum is reached, the third and fourth the indices of where the peak begins and ends — in the sense of where the pattern starts and ends.

**Note**

On Matlab Central there are several realizations for finding peaks, for example “peakfinder”, “peakseek”, or “peakdetect”. And “findpeaks” is also the name of a function in the Matlab ‘signal’ toolbox.

The parameter names are taken from the “findpeaks” function in ‘signal’, but the implementation utilizing regular expressions is unique and fast.

**See Also**

[hampel](#)

**Examples**

```
x <- seq(0, 1, len = 1024)
pos <- c(0.1, 0.13, 0.15, 0.23, 0.25, 0.40, 0.44, 0.65, 0.76, 0.78, 0.81)
hgt <- c(4, 5, 3, 4, 5, 4.2, 2.1, 4.3, 3.1, 5.1, 4.2)
wdt <- c(0.005, 0.005, 0.006, 0.01, 0.01, 0.03, 0.01, 0.01, 0.005, 0.008, 0.005)

pSignal <- numeric(length(x))
for (i in seq(along=pos)) {
  pSignal <- pSignal + hgt[i]/(1 + abs((x - pos[i])/wdt[i]))^4
}
findpeaks(pSignal, npeaks=3, threshold=4, sortstr=TRUE)

## Not run:
plot(pSignal, type="l", col="navy")
grid()
x <- findpeaks(pSignal, npeaks=3, threshold=4, sortstr=TRUE)
points(x[, 2], x[, 1], pch=20, col="maroon")
## End(Not run)
```

---

finds

*find function (Matlab Style)*

---

**Description**

Finds indices of nonzero elements.

**Usage**

```
finds(v)
```

**Arguments**

v                      logical or numeric vector or array

**Details**

Finds indices of true or nonzero elements of argument v; can be used with a logical expression.

**Value**

Indices of elements matching the expression x.

**Examples**

```
finds(-3:3 >= 0)
finds(c(0, 1, 0, 2, 3))
```

---

findzeros

*Find All Roots*

---

**Description**

Finding all roots of a univariate function in an interval by splitting the interval in many small subintervals.

**Usage**

```
findzeros(f, a, b, n = 100, tol = .Machine$double.eps^(2/3), ...)
```

**Arguments**

f	functions whose roots shall be found.
a, b	endpoints of the interval.
n	number of subintervals to generate and search.
tol	tolerance for identifying zeros.
...	Additional parameters to be passed to the function.

**Details**

Roots, i.e. zeros in a subinterval will be found by applying `uniroot` to any subinterval where the sign of the function changes. The endpoints of the interval will be tested separately.

If the function points are both positive or negative and the slope in this interval is high enough, the minimum or maximum will be determined with `optimize` and checked for a possible zero.

The function need not be vectorized.

**Value**

Numeric vector with the x-positions of all roots found in the interval.

**See Also**

[findmins](#)

**Examples**

```

f1 <- function(x) sin(pi/x)
findzeros(f1, 1/10, 1)
# 0.1000000 0.1111028 0.1250183 0.1428641 0.1666655
# 0.2000004 0.2499867 0.3333441 0.4999794 1.0000000

f2 <- function(x) 0.5*(1 + sin(10*pi*x))
findzeros(f2, 0, 1)
# 0.15 0.35 0.55 0.75 0.95

f3 <- function(x) sin(pi/x) + 1
findzeros(f3, 0.1, 0.5)
# 0.1052632 0.1333333 0.1818182 0.2857143

f4 <- function(x) sin(pi/x) - 1
findzeros(f4, 0.1, 0.5)
# 0.1176471 0.1538462 0.2222222 0.4000000

## Not run:
# Dini function
Dini <- function(x) x * besselJ(x, 1) + 3 * besselJ(x, 0)
findzeros(Dini, 0, 100, n = 128)
ezplot(Dini, 0, 100, n = 512)

## End(Not run)

```

---

fletcher\_powell

*Fletcher-Powell Conjugate Gradient Minimization*


---

**Description**

Conjugate Gradient (CG) minimization through the Davidon-Fletcher-Powell approach for function minimization.

The Davidon-Fletcher-Powell (DFP) and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) methods are the first quasi-Newton minimization methods developed. These methods differ only in some details; in general, the BFGS approach is more robust.

**Usage**

```

fletcher_powell(x0, f, g = NULL,
               maxiter = 1000, tol = .Machine$double.eps^(2/3))

```

**Arguments**

x0	start value.
f	function to be minimized.
g	gradient function of f; if NULL, a numerical gradient will be calculated.
maxiter	max. number of iterations.
tol	relative tolerance, to be used as stopping rule.



**Details**

The starting point is Newton's method in the multivariate case, when the estimate of the minimum is updated by the following equation

$$x_{new} = x - H^{-1}(x)grad(g)(x)$$

where  $H$  is the Hessian and  $grad$  the gradient.

The basic idea is to generate a sequence of good approximations to the inverse Hessian matrix, in such a way that the approximations are again positive definite.

**Value**

List with following components:

xmin	minimum solution found.
fmin	value of f at minimum.
niter	number of iterations performed.

**Note**

Used some Matlab code as described in the book "Applied Numerical Analysis Using Matlab" by L. V.Fausett.

**References**

J. F. Bonnans, J. C. Gilbert, C. Lemarechal, and C. A. Sagastizabal. Numerical Optimization: Theoretical and Practical Aspects. Second Edition, Springer-Verlag, Berlin Heidelberg, 2006.

**See Also**

[steep\\_descent](#)

**Examples**

```
## Rosenbrock function
rosenbrock <- function(x) {
  n <- length(x)
  x1 <- x[2:n]
  x2 <- x[1:(n-1)]
  sum(100*(x1-x2^2)^2 + (1-x2)^2)
}
fletcher_powell(c(0, 0), rosenbrock)
# $xmin
# [1] 1 1
# $fmin
# [1] 1.774148e-27
# $niter
# [1] 14
```

---

`flipdim`*Matrix Flipping (Matlab Style)*

---

**Description**

Flip matrices up and down or left and right; or circulating indices per dimension.

**Usage**

```
flipdim(a, dim)
flipud(a)
fliplr(a)
circshift(a, sz)
```

**Arguments**

<code>a</code>	numeric or complex matrix
<code>dim</code>	flipping dimension; can only be 1 (default) or 2
<code>sz</code>	integer vector of length 1 or 2.

**Details**

`flipdim` will flip a matrix along the `dim` dimension, where `dim=1` means flipping rows, and `dim=2` flipping the columns.

`flipud` and `fliplr` are simply shortcuts for `flipdim(a, 1)` resp. `flipdim(a, 2)`.

`circshift(a, sz)` circulates each dimension (should be applicable to arrays).

**Value**

the original matrix somehow flipped or circularly shifted.

**Examples**

```
a <- matrix(1:12, nrow=3, ncol=4, byrow=TRUE)
flipud(a)
fliplr(a)

circshift(a, c(1, -1))
v <- 1:10
circshift(v, 5)
```

---

fminbnd *Finding Function Minimum*

---

**Description**

Find minimum of single-variable function on fixed interval.

**Usage**

```
fminbnd(f, a, b, maxiter = 1000, maximum = FALSE,  
        tol = 1e-07, rel.tol = tol, abs.tol = 1e-15, ...)
```

**Arguments**

f	function whose minimum or maximum is to be found.
a, b	endpoints of the interval to be searched.
maxiter	maximal number of iterations.
maximum	logical; shall maximum or minimum be found; default FALSE.
tol	relative tolerance; left over for compatibility.
rel.tol, abs.tol	relative and absolute tolerance.
...	additional variables to be passed to the function.

**Details**

fminbnd finds the minimum of a function of one variable within a fixed interval. It applies Brent's algorithm, based on golden section search and parabolic interpolation.

fminbnd may only give local solutions. fminbnd never evaluates f at the endpoints.

**Value**

List with

xmin	location of the minimum resp. maximum.
fmin	function value at the optimum.
niter	number of iterations used.
estim.prec	estimated precision.

**Note**

fminbnd mimics the Matlab function of the same name.

**References**

R. P. Brent (1973). Algorithms for Minimization Without Derivatives. Dover Publications, reprinted 2002.

**See Also**

[fibsearch](#), [golden\\_ratio](#)

**Examples**

```
## CHEBFUN example by Trefethen
f <- function(x) exp(x)*sin(3*x)*tanh(5*cos(30*x))
fminbnd(f, -1, 1) # fourth local minimum (from left)
g <- function(x) complexstep(f, x) # complex-step derivative
xs <- findzeros(g, -1, 1) # local minima and maxima
ys <- f(xs); n0 <- which.min(ys) # index of global minimum
fminbnd(f, xs[n0-1], xs[n0+1]) # xmin:0.7036632, fmin: -1.727377

## Not run:
ezplot(f, -1, 1, n = 1000, col = "darkblue", lwd = 2)
ezplot(function(x) g(x)/150, -1, 1, n = 1000, col = "darkred", add = TRUE)
grid()
## End(Not run)
```

---

fmincon

---

*Minimize Nonlinear Constrained Multivariable Function.*


---

**Description**

Find minimum of multivariable functions with nonlinear constraints.

**Usage**

```
fmincon(x0, fn, gr = NULL, ..., method = "SQP",
        A = NULL, b = NULL, Aeq = NULL, beq = NULL,
        lb = NULL, ub = NULL, hin = NULL, heq = NULL,
        tol = 1e-06, maxfeval = 10000, maxiter = 5000)
```

**Arguments**

x0	starting point.
fn	objective function to be minimized.
gr	gradient function of the objective; not used for SQP method.
...	additional parameters to be passed to the function.
method	method options 'SQP', 'auglag'; only 'SQP' is implemented.
A, b	linear inequality constraints of the form $A x \leq b$ .
Aeq, beq	linear equality constraints of the form $Aeq x = beq$ .
lb, ub	bounds constraints of the form $lb \leq x \leq ub$ .
hin	nonlinear inequality constraints of the form $hin(x) \leq 0$ .
heq	nonlinear equality constraints of the form $heq(x) = 0$ .

tol	relative tolerance.
maxiter	maximum number of iterations.
maxfeval	maximum number of function evaluations.

### Details

Wraps the function `soln1` in the 'NlcOptim' package. The underlying method is a Sequential Quadratic Programming (SQP) approach.

Constraints can be defined in different ways, as linear constraints in matrix form, as nonlinear functions, or as bounds constraints.

### Value

List with the following components:

par	the best minimum found.
value	function value at the minimum.
convergence	integer indicating the terminating situation.
info	parameter list describing the final situation.

### Note

`fmincon` mimics the Matlab function of the same name.

### Author(s)

Xiyan Chen for the package NlcOptim.

### References

J. Nocedal and S. J. Wright (2006). Numerical Optimization. Second Edition, Springer Science+Business Media, New York.

### See Also

[fminsearch](#), [fminunc](#),

### Examples

```
# Classical Rosenbrock function
n <- 10; x0 <- rep(1/n, n)
fn <- function(x) {n <- length(x)
  x1 <- x[2:n]; x2 <- x[1:(n - 1)]
  sum(100 * (x1 - x2^2)^2 + (1 - x2)^2)
}
# Equality and inequality constraints
heq1 <- function(x) sum(x)-1.0
hin1 <- function(x) -1 * x
hin2 <- function(x) x - 0.5
```

```

ub <- rep(0.5, n)

# Apply constraint minimization
res <- fmincon(x0, fn, hin = hin1, heq = heq1)
res$par; res$value

```

---

fminsearch

*Derivative-free Nonlinear Function Minimization*


---

### Description

Find minimum of multivariable functions using derivative-free methods.

### Usage

```

fminsearch(fn, x0, ..., lower = NULL, upper = NULL,
           method = c("Nelder-Mead", "Hooke-Jeeves"),
           minimize = TRUE, maxiter = 1000, tol = 1e-08)

```

### Arguments

fn	function whose minimum or maximum is to be found.
x0	point considered near to the optimum.
...	additional variables to be passed to the function.
lower, upper	lower and upper bounds constraints.
method	"Nelder-Mead" (default) or "Hooke-Jeeves"; can be abbreviated.
minimize	logical; shall a minimum or a maximum be found.
maxiter	maximal number of iterations
tol	relative tolerance.

### Details

fminsearch finds the minimum of a nonlinear scalar multivariable function, starting at an initial estimate and returning a value x that is a local minimizer of the function. With minimize=FALSE it searches for a maximum, by default for a (local) minimum.

As methods/solvers "Nelder-Mead" and "Hooke-Jeeves" are available. Only Hooke-Jeeves can handle bounds constraints. For nonlinear constraints see fmincon, and for methods using gradients see fminunc.

Important: fminsearch may only give local solutions.

**Value**

List with

xopt	location of the location of minimum resp. maximum.
fmin	function value at the optimum.
count	number of function calls.
convergence	info about convergence: not used at the moment.
info	special information from the solver.

**Note**

fminsearch mimics the Matlab function of the same name.

**References**

Nocedal, J., and S. Wright (2006). Numerical Optimization. Second Edition, Springer-Verlag, New York.

**See Also**

[nelder\\_mead](#), [hooke\\_jeeves](#)

**Examples**

```
# Rosenbrock function
rosena <- function(x, a) 100*(x[2]-x[1]^2)^2 + (a-x[1])^2 # min: (a, a^2)

fminsearch(rosena, c(-1.2, 1), a = sqrt(2), method="Nelder-Mead")
## $xmin          $fmin
## [1] 1.414292 2.000231 [1] 1.478036e-08

fminsearch(rosena, c(-1.2, 1), a = sqrt(2), method="Hooke-Jeeves")
## $xmin          $fmin
## [1] 1.414215 2.000004 [1] 1.79078e-12
```

---

fminunc

*Minimize Unconstrained Multivariable Function*


---

**Description**

Find minimum of unconstrained multivariable functions.

**Usage**

```
fminunc(x0, fn, gr = NULL, ...,
        tol = 1e-08, maxiter = 0, maxfeval = 0)
```

**Arguments**

<code>x0</code>	starting point.
<code>fn</code>	objective function to be minimized.
<code>gr</code>	gradient function of the objective.
<code>...</code>	additional parameters to be passed to the function.
<code>tol</code>	relative tolerance.
<code>maxiter</code>	maximum number of iterations.
<code>maxfeval</code>	maximum number of function evaluations.

**Details**

The method used here for unconstrained minimization is a variant of a "variable metric" resp. quasi-Newton approach.

**Value**

List with the following components:

<code>par</code>	the best minimum found.
<code>value</code>	function value at the minimum.
<code>counts</code>	number of function and gradient calls.
<code>convergence</code>	integer indicating the terminating situation.
<code>message</code>	description of the final situation.

**Note**

`fminunc` mimics the Matlab function of the same name.

**Author(s)**

The "variable metric" code provided by John Nash (package `Rvmmmin`), stripped-down version by Hans W. Borchers.

**References**

J. Nocedal and S. J. Wright (2006). Numerical Optimization. Second Edition, Springer Science+Business Media, New York.

**See Also**

[fminsearch](#), [fmincon](#),

**Examples**

```
fun = function(x)
    x[1]*exp(-(x[1]^2 + x[2]^2)) + (x[1]^2 + x[2]^2)/20
fminunc(x0 = c(1, 2), fun)
## xmin: c(-0.6691, 0.0000); fmin: -0.4052
```



---

fnorm	<i>Function Norm</i>
-------	----------------------

---

### Description

The `fnorm` function calculates several different types of function norms for depending on the argument `p`.

### Usage

```
fnorm(f, g, x1, x2, p = 2, npoints = 100)
```

### Arguments

<code>f, g</code>	functions given by name or string.
<code>x1, x2</code>	endpoints of the interval.
<code>p</code>	Numeric scalar or <code>Inf</code> , <code>-Inf</code> ; default is 2.
<code>npoints</code>	number of points to be considered in the interval.

### Details

`fnorm` returns a scalar that gives some measure of the distance of two functions `f` and `g` on the interval `[x1, x2]`.

It takes `npoints` equidistant points in the interval, computes the function values for `f` and `g` and applies `Norm` to their difference.

Especially `p=Inf` returns the maximum norm, while `fnorm(f, g, x1, x2, p = 1, npoints) / npoints` would return some estimate of the mean distance.

### Value

Numeric scalar (or `Inf`), or `NA` if one of these functions returns `NA`.

### Note

Another kind of ‘mean’ distance could be calculated by integrating the difference `f-g` and dividing through the length of the interval.

### See Also

[Norm](#)

**Examples**

```

xp <- seq(-1, 1, length.out = 6)
yp <- runge(xp)
p5 <- polyfit(xp, yp, 5)
f5 <- function(x) polyval(p5, x)
fnorm(runge, f5, -1, 1, p = Inf)           #=> 0.4303246
fnorm(runge, f5, -1, 1, p = Inf, npoints = 1000) #=> 0.4326690

# Compute mean distance using fnorm:
fnorm(runge, f5, -1, 1, p = 1, 1000) / 1000   #=> 0.1094193

# Compute mean distance by integration:
fn <- function(x) abs(runge(x) - f5(x))
integrate(fn, -1, 1)$value / 2                #=> 0.1095285

```

fornberg

*Fornberg's Finite Difference Approximation***Description**

Finite difference approximation using Fornberg's method for the derivatives of order 1 to k based on irregular grid values.

**Usage**

```
fornberg(x, y, xs, k = 1)
```

**Arguments**

x	grid points on the x-axis, must be distinct.
y	discrete values of the function at the grid points.
xs	point at which to approximate (not vectorized).
k	order of derivative, $k \leq \text{length}(x) - 1$ required.

**Details**

Compute coefficients for finite difference approximation for the derivative of order k at xs based on grid values at points in x. For  $k=0$  this will evaluate the interpolating polynomial itself, but call it with  $k=1$ .

**Value**

Returns a matrix of size  $(\text{length}(xs))$ , where the  $(k+1)$ -th column gives the value of the k-th derivative. Especially the first column returns the polynomial interpolation of the function.

**Note**

Fornberg's method is considered to be numerically more stable than applying Vandermonde's matrix.

**References**

LeVeque, R. J. (2007). Finite Difference Methods for Ordinary and Partial Differential Equations. Society for Industrial and Applied Mathematics (SIAM), Philadelphia.

**See Also**

[neville](#), [newtonInterp](#)

**Examples**

```
x <- 2 * pi * c(0.0, 0.07, 0.13, 0.2, 0.28, 0.34, 0.47, 0.5, 0.71, 0.95, 1.0)
y <- sin(0.9*x)
xs <- linspace(0, 2*pi, 51)
fornb <- fornberg(x, y, xs, 10)
## Not run:
matplot(xs, fornb, type="l")
grid()
## End(Not run)
```

---

fprintf *Formatted Printing (Matlab style)*

---

**Description**

Formatted printing to stdout or a file.

**Usage**

```
fprintf(fmt, ..., file = "", append = FALSE)
```

**Arguments**

fmt	a character vector of format strings.
...	values passed to the format string.
file	a connection or a character string naming the file to print to; default is "" which means standard output.
append	logical; shall the output be appended to the file; default is FALSE.

**Details**

fprintf applies the format string `fmt` to all input data `...` and writes the result to standard output or a file. The usual C-style string formatting commands are used-

**Value**

Returns invisibly the number of bytes printed (using nchar).

**See Also**

[sprintf](#)

**Examples**

```
## Examples:
nbytes <- fprintf("Results are:\n", file = "")
for (i in 1:10) {
  fprintf("%4d %15.7f\n", i, exp(i), file = "")
}
```

---

fractalcurve

*Fractal Curves*

---

**Description**

Generates the following fractal curves: Dragon Curve, Gosper Flowsnake Curve, Hexagon Molecule Curve, Hilbert Curve, Koch Snowflake Curve, Sierpinski Arrowhead Curve, Sierpinski (Cross) Curve, Sierpinski Triangle Curve.

**Usage**

```
fractalcurve(n, which = c("hilbert", "sierpinski", "snowflake",
  "dragon", "triangle", "arrowhead", "flowsnake", "molecule"))
```

**Arguments**

n                    integer, the ‘order’ of the curve  
 which                character string, which curve to compute.

**Details**

The Hilbert curve is a continuous curve in the plane with  $4^N$  points.

The Sierpinski (cross) curve is a closed curve in the plane with  $4^{(N+1)+1}$  points.

His arrowhead curve is a continuous curve in the plane with  $3^N+1$  points, and his triangle curve is a closed curve in the plane with  $2*3^N+2$  points.

The Koch snowflake curve is a closed curve in the plane with  $3*2^N+1$  points.

The dragon curve is a continuous curve in the plane with  $2^{(N+1)}$  points.

The flowsnake curve is a continuous curve in the plane with  $7^N+1$  points.

The hexagon molecule curve is a closed curve in the plane with  $6*3^N+1$  points.

**Value**

Returns a list with  $x$ ,  $y$  the  $x$ - resp.  $y$ -coordinates of the generated points describing the fractal curve.

**Author(s)**

Copyright (c) 2011 Jonas Lundgren for the Matlab toolbox fractal curves available on Matlab-Central under BSD license; here re-implemented in R with explicit allowance from the author.

**References**

Peitgen, H.O., H. Juergens, and D. Saupe (1993). Fractals for the Classroom. Springer-Verlag Berlin Heidelberg.

**Examples**

```
## The Hilbert curve transforms a 2-dim. function into a time series.
z <- fractalcurve(4, which = "hilbert")

## Not run:
f1 <- function(x, y) x^2 + y^2
plot(f1(z$x, z$y), type = 'l', col = "darkblue", lwd = 2,
     ylim = c(-1, 2), main = "Functions transformed by Hilbert curves")

f2 <- function(x, y) x^2 - y^2
lines(f2(z$x, z$y), col = "darkgreen", lwd = 2)

f3 <- function(x, y) x^2 * y^2
lines(f3(z$x, z$y), col = "darkred", lwd = 2)
grid()
## End(Not run)

## Not run:
## Show some more fractal curves
n <- 8
opar <- par(mfrow=c(2,2), mar=c(2,2,1,1))

z <- fractalcurve(n, which="dragon")
x <- z$x; y <- z$y
plot(x, y, type='l', col="darkgrey", lwd=2)
title("Dragon Curve")

z <- fractalcurve(n, which="molecule")
x <- z$x; y <- z$y
plot(x, y, type='l', col="darkblue")
title("Molecule Curve")

z <- fractalcurve(n, which="arrowhead")
x <- z$x; y <- z$y
plot(x, y, type='l', col="darkgreen")
title("Arrowhead Curve")
```

```

z <- fractalcurve(n, which="snowflake")
x <- z$x; y <- z$y
plot(x, y, type='l', col="darkred", lwd=2)
title("Snowflake Curve")

par(opar)
## End(Not run)

```

---

fresnelS/C

*Fresnel Integrals*


---

### Description

(Normalized) Fresnel integrals  $S(x)$  and  $C(x)$

### Usage

```

fresnelS(x)
fresnelC(x)

```

### Arguments

$x$  numeric vector.

### Details

The *normalized* Fresnel integrals are defined as

$$S(x) = \int_0^x \sin(\pi/2 t^2) dt$$

$$C(x) = \int_0^x \cos(\pi/2 t^2) dt$$

This program computes the Fresnel integrals  $S(x)$  and  $C(x)$  using Fortran code by Zhang and Jin. The accuracy is almost up to Machine precision.

The functions are not (yet) truly vectorized, but use a call to ‘apply’. The underlying function `.fresnel` (not exported) computes single values of  $S(x)$  and  $C(x)$  at the same time.

### Value

Numeric vector of function values.

### Note

Copyright (c) 1996 Zhang and Jin for the Fortran routines, converted to Matlab using the open source project ‘f2matlab’ by Ben Barrowes, posted to MatlabCentral in 2004, and then translated to R by Hans W. Borchers.

**References**

Zhang, S., and J. Jin (1996). *Computation of Special Functions*. Wiley-Interscience.

**See Also**

[gaussLegendre](#)

**Examples**

```
## Compute Fresnel integrals through Gauss-Legendre quadrature
f1 <- function(t) sin(0.5 * pi * t^2)
f2 <- function(t) cos(0.5 * pi * t^2)
for (x in seq(0.5, 2.5, by = 0.5)) {
  cgl <- gaussLegendre(51, 0, x)
  fs <- sum(cgl$w * f1(cgl$x))
  fc <- sum(cgl$w * f2(cgl$x))
  cat(formatC(c(x, fresnelS(x), fs, fresnelC(x), fc),
    digits = 8, width = 12, flag = " ----"), "\n")
}

## Not run:
xs <- seq(0, 7.5, by = 0.025)
ys <- fresnelS(xs)
yc <- fresnelC(xs)

## Function plot of the Fresnel integrals
plot(xs, ys, type = "l", col = "darkgreen",
  xlim = c(0, 8), ylim = c(0, 1),
  xlab = "", ylab = "", main = "Fresnel Integrals")
lines(xs, yc, col = "blue")
legend(6.25, 0.95, c("S(x)", "C(x)"), col = c("darkgreen", "blue"), lty = 1)
grid()

## The Cornu (or Euler) spiral
plot(c(-1, 1), c(-1, 1), type = "n",
  xlab = "", ylab = "", main = "Cornu Spiral")
lines(ys, yc, col = "red")
lines(-ys, -yc, col = "red")
grid()
## End(Not run)
```

**Description**

Solve a system of  $m$  nonlinear equations of  $n$  variables.

**Usage**

```
fsolve(f, x0, J = NULL,
       maxiter = 100, tol = .Machine$double.eps^(0.5), ...)
```

**Arguments**

f	function describing the system of equations.
x0	point near to the root.
J	Jacobian function of f, or NULL.
maxiter	maximum number of iterations in gaussNewton.
tol	tolerance to be used in Gauss-Newton.
...	additional variables to be passed to the function.

**Details**

fsolve tries to solve the components of function f simultaneously and uses the Gauss-Newton method with numerical gradient and Jacobian. If  $m = n$ , it uses broyden. Not applicable for univariate root finding.

**Value**

List with

x	location of the solution.
fval	function value at the solution.

**Note**

fsolve mimics the Matlab function of the same name.

**References**

Antoniou, A., and W.-S. Lu (2007). Practical Optimization: Algorithms and Engineering Applications. Springer Science+Business Media, New York.

**See Also**

[broyden](#), [gaussNewton](#)

**Examples**

```
## Not run:
# Find a matrix X such that X * X * X = [1, 2; 3, 4]
F <- function(x) {
  a <- matrix(c(1, 3, 2, 4), nrow = 2, ncol = 2, byrow = TRUE)
  X <- matrix(x, nrow = 2, ncol = 2, byrow = TRUE)
  return(c(X %*% X %*% X - a))
}
x0 <- matrix(1, 2, 2)
```



```

X <- matrix(fsolve(F, x0)$x, 2, 2)
X
# -0.1291489  0.8602157
#  1.2903236  1.1611747

## End(Not run)

```

---

fzero *Root Finding Algorithm*

---

### Description

Find root of continuous function of one variable.

### Usage

```
fzero(fun, x, maxiter = 500, tol = 1e-12, ...)
```

### Arguments

fun	function whose root is sought.
x	a point near the root or an interval giving end points.
maxiter	maximum number of iterations.
tol	relative tolerance.
...	additional arguments to be passed to the function.

### Details

fzero tries to find a zero of  $f$  near  $x$ , if  $x$  is a scalar. Expands the interval until different signs are found at the endpoints or the maximum number of iterations is exceeded. If  $x$  is a vector of length two, fzero assumes  $x$  is an interval where the sign of  $x[1]$  differs from the sign of  $x[2]$ . An error occurs if this is not the case.

“This is essentially the ACM algorithm 748. The structure of the algorithm has been transformed non-trivially: it implement here a FSM version using one interior point determination and one bracketing per iteration, thus reducing the number of temporary variables and simplifying the structure.”

This approach will not find zeroes of quadratic order.

### Value

fzero returns a list with

x	location of the root.
fval	function value at the root.

**Note**

fzero mimics the Matlab function of the same name, but is translated from Octave's fzero function, copyrighted (c) 2009 by Jaroslav Hajek.

**References**

Alefeld, Potra and Shi (1995). Enclosing Zeros of Continuous Functions. ACM Transactions on Mathematical Software, Vol. 21, No. 3.

**See Also**

[uniroot](#), [brent](#)

**Examples**

```
fzero(sin, 3)           # 3.141593
fzero(cos,c(1, 2))     # 1.570796
fzero(function(x) x^3-2*x-5, 2) # 2.094551
```

---

fzsolve

*Complex Root Finding*

---

**Description**

Find the root of a complex function

**Usage**

```
fzsolve(fz, z0)
```

**Arguments**

fz	complex(-analytic) function.
z0	complex point near the assumed root.

**Details**

fzsolve tries to find the root of the complex and relatively smooth (i.e., analytic) function near a starting point.

The function is considered as real function  $R^2 \rightarrow R^2$  and the newtonsys function is applied.

**Value**

Complex point with sufficiently small function value.

**See Also**

[newtonsys](#)

**Examples**

```
fz <- function(z) sin(z)^2 + sqrt(z) - log(z)
fzsolve(fz, 1+1i)
# 0.2555197+0.8948303i
```

gammainc

*Incomplete Gamma Function***Description**

Lower and upper incomplete gamma function.

**Usage**

```
gammainc(x, a)
```

```
incgam(x, a)
```

**Arguments**

x                    positive real number.  
a                    real number.

**Details**

gammainc computes the lower and upper incomplete gamma function, including the regularized gamma function. The lower and upper incomplete gamma functions are defined as

$$\gamma(x, a) = \int_0^x e^{-t} t^{a-1} dt$$

and

$$\Gamma(x, a) = \int_x^\infty e^{-t} t^{a-1} dt$$

while the regularized incomplete gamma function is  $\gamma(x, a)/\Gamma(a)$ .

incgam (a name used in Pari/GP) computes the upper incomplete gamma function alone, applying the R function pgamma. The accuracy is thus much higher. It works for  $a \geq -1$ , for even smaller values a recursion will give the result.

**Value**

gammainc returns a list with the values of the lower, the upper, and regularized lower incomplete gamma function. incgam only returns the value of the incomplete upper gamma function.

**Note**

Directly converting Fortran code is often easier than translating Matlab code generated with f2matlab.

**References**

Zhang, Sh., and J. Jin (1996). Computation of Special Functions. Wiley-Interscience, New York.

**See Also**

[gamma](#), [pgamma](#)

**Examples**

```
gammainc( 1.5, 2)
gammainc(-1.5, 2)

incgam(3, 1.2)
incgam(3, 0.5); incgam(3, -0.5)
```

---

gammaz

*Complex Gamma Function*

---

**Description**

Gamma function valid in the entire complex plane.

**Usage**

```
gammaz(z)
```

**Arguments**

**z** Real or complex number or a numeric or complex vector.

**Details**

Computes the Gamma function for complex arguments using the Lanczos series approximation.

Accuracy is 15 significant digits along the real axis and 13 significant digits elsewhere.

To compute the logarithmic Gamma function use `log(gammaz(z))`.

**Value**

Returns a complex vector of function values.

**Note**

Copyright (c) 2001 Paul Godfrey for a Matlab version available on Mathwork's Matlab Central under BSD license.

Numerical Recipes used a 7 terms formula for a less effective approximation.

**References**

Zhang, Sh., and J. Jin (1996). Computation of Special Functions. Wiley-Interscience, New York.

**See Also**

[gamma](#), [gsl::lngamma\\_complex](#)

**Examples**

```
max(gamma(1:10) - gammaz(1:10))
gammaz(-1)
gammaz(c(-2-2i, -1-1i, 0, 1+1i, 2+2i))

# Euler's reflection formula
z <- 1+1i
gammaz(1-z) * gammaz(z) # == pi/sin(pi*z)
```

---

gaussHermite

*Gauss-Hermite Quadrature Formula*


---

**Description**

Nodes and weights for the n-point Gauss-Hermite quadrature formula.

**Usage**

```
gaussHermite(n)
```

**Arguments**

n                      Number of nodes in the interval ]-Inf, Inf[.

**Details**

Gauss-Hermite quadrature is used for integrating functions of the form

$$\int_{-\infty}^{\infty} f(x)e^{-x^2} dx$$

over the infinite interval ] -∞, ∞[.

x and w are obtained from a tridiagonal eigenvalue problem. The value of such an integral is then sum(w\*f(x)).

**Value**

List with components x, the nodes or points in ]-Inf, Inf[, and w, the weights applied at these nodes.

**Note**

The basic quadrature rules are well known and can, e. g., be found in Gautschi (2004) — and explicit Matlab realizations in Trefethen (2000). These procedures have also been implemented in Matlab by Geert Van Damme, see his entries at MatlabCentral since 2010.

**References**

Gautschi, W. (2004). Orthogonal Polynomials: Computation and Approximation. Oxford University Press.

Trefethen, L. N. (2000). Spectral Methods in Matlab. SIAM, Society for Industrial and Applied Mathematics.

**See Also**

[gaussLegendre](#), [gaussLaguerre](#)

**Examples**

```
cc <- gaussHermite(17)
# Integrate exp(-x^2) from -Inf to Inf
sum(cc$w)                #=> 1.77245385090552 == sqrt(pi)
# Integrate x^2 exp(-x^2)
sum(cc$w * cc$x^2)       #=> 0.88622692545276 == sqrt(pi) / 2
# Integrate cos(x) * exp(-x^2)
sum(cc$w * cos(cc$x))    #=> 1.38038844704314 == sqrt(pi)/exp(1)^0.25
```

---

gaussLaguerre

*Gauss-Laguerre Quadrature Formula*

---

**Description**

Nodes and weights for the n-point Gauss-Laguerre quadrature formula.

**Usage**

```
gaussLaguerre(n, a = 0)
```

**Arguments**

n                    Number of nodes in the interval  $[0, \text{Inf}]$ .

a                    exponent of  $x$  in the integrand: must be greater or equal to 0, otherwise the integral would not converge.

**Details**

Gauss-Laguerre quadrature is used for integrating functions of the form

$$\int_0^{\infty} f(x)x^a e^{-x} dx$$

over the infinite interval  $]0, \infty[$ .

$x$  and  $w$  are obtained from a tridiagonal eigenvalue problem. The value of such an integral is then  $\text{sum}(w*f(x))$ .

**Value**

List with components  $x$ , the nodes or points in  $[\theta, \text{Inf}[$ , and  $w$ , the weights applied at these nodes.

**Note**

The basic quadrature rules are well known and can, e. g., be found in Gautschi (2004) — and explicit Matlab realizations in Trefethen (2000). These procedures have also been implemented in Matlab by Geert Van Damme, see his entries at MatlabCentral since 2010.

**References**

- Gautschi, W. (2004). Orthogonal Polynomials: Computation and Approximation. Oxford University Press.
- Trefethen, L. N. (2000). Spectral Methods in Matlab. SIAM, Society for Industrial and Applied Mathematics.

**See Also**

[gaussLegendre](#), [gaussHermite](#)

**Examples**

```
cc <- gaussLaguerre(7)
# integrate exp(-x) from 0 to Inf
sum(cc*w)           # 1
# integrate x^2 * exp(-x)   # integral x^n * exp(-x) is n!
sum(cc*w * cc*x^2)   # 2
# integrate sin(x) * exp(-x)
cc <- gaussLaguerre(17, 0) # we need more nodes
sum(cc*w * sin(cc*x))    #=> 0.499999999994907 , should be 0.5
```

gaussLegendre

*Gauss-Legendre Quadrature Formula***Description**

Nodes and weights for the n-point Gauss-Legendre quadrature formula.

**Usage**

```
gaussLegendre(n, a, b)
```

**Arguments**

n	Number of nodes in the interval [a, b].
a, b	lower and upper limit of the integral; must be finite.

**Details**

x and w are obtained from a tridiagonal eigenvalue problem.

**Value**

List with components x, the nodes or points in [a, b], and w, the weights applied at these nodes.

**Note**

Gauss quadrature is not suitable for functions with singularities.

**References**

Gautschi, W. (2004). Orthogonal Polynomials: Computation and Approximation. Oxford University Press.

Trefethen, L. N. (2000). Spectral Methods in Matlab. SIAM, Society for Industrial and Applied Mathematics.

**See Also**

[gaussHermite](#), [gaussLaguerre](#)

**Examples**

```
## Quadrature with Gauss-Legendre nodes and weights
f <- function(x) sin(x+cos(10*exp(x)))/3
#\dontrun{ezplot(f, -1, 1, fill = TRUE)}
cc <- gaussLegendre(51, -1, 1)
Q <- sum(cc$w * f(cc$x)) #=> 0.0325036515865218 , true error: < 1e-15

# If f is not vectorized, do an explicit summation:
```



```

Q <- 0; x <- cc$x; w <- cc$w
for (i in 1:51) Q <- Q + w[i] * f(x[i])

# If f is infinite at b = 1, set b <- b - eps (with, e.g., eps = 1e-15)

# Use Gauss-Kronrod approach for error estimation
cc <- gaussLegendre(103, -1, 1)
abs(Q - sum(cc$w * f(cc$x))) # rel.error < 1e-10

# Use Gauss-Hermite for vector-valued functions
f <- function(x) c(sin(pi*x), exp(x), log(1+x))
cc <- gaussLegendre(32, 0, 1)
drop(cc$w %% matrix(f(cc$x), ncol = 3)) # c(2/pi, exp(1) - 1, 2*log(2) - 1)
# absolute error < 1e-15

```

gaussNewton

*Gauss-Newton Function Minimization***Description**

Gauss-Newton method of minimizing a term  $f_1(x)^2 + \dots + f_m(x)^2$  or  $F'F$  where  $F = (f_1, \dots, f_m)$  is a multivariate function of  $n$  variables, not necessarily  $n = m$ .

**Usage**

```

gaussNewton(x0, Ffun, Jfun = NULL,
            maxiter = 100, tol = .Machine$double.eps^(1/2), ...)

```

**Arguments**

Ffun	m functions of n variables.
Jfun	function returning the Jacobian matrix of Ffun; if NULL, the default, the Jacobian will be computed numerically. The gradient of f will be computed internally from the Jacobian (i.e., cannot be supplied).
x0	Numeric vector of length n.
maxiter	Maximum number of iterations.
tol	Tolerance, relative accuracy.
...	Additional parameters to be passed to f.

**Details**

Solves the system of equations applying the Gauss-Newton's method. It is especially designed for minimizing a sum-of-squares of functions and can be used to find a common zero of several function.

This algorithm is described in detail in the textbook by Antoniou and Lu, incl. different ways to modify and remedy the Hessian if not being positive definite. Here, the approach by Goldfeld, Quandt and Trotter is used, and the hessian modified by the Matthews and Davies algorithm if still not invertible.

To accelerate the iteration, an inexact linesearch is applied.

**Value**

List with components:

xs the minimum or root found so far,  
 fs the square root of sum of squares of the values of f,  
 iter the number of iterations needed, and  
 relerr the absolute distance between the last two solutions.

**Note**

If  $n=m$  then directly applying the newtonsys function might be a better alternative.

**References**

Antoniou, A., and W.-S. Lu (2007). Practical Optimization: Algorithms and Engineering Applications. Springer Business+Science, New York.

**See Also**

[newtonsys](#), [softline](#)

**Examples**

```
f1 <- function(x) c(x[1]^2 + x[2]^2 - 1, x[1] + x[2] - 1)
gaussNewton(c(4, 4), f1)

f2 <- function(x) c(x[1] + 10*x[2], sqrt(5)*(x[1] - x[4]),
                  (x[2] - 2*x[3])^2, 10*(x[1] - x[4])^2)
gaussNewton(c(-2, -1, 1, 2), f2)

f3 <- function(x)
  c(2*x[1] - x[2] - exp(-x[1]), -x[1] + 2*x[2] - exp(-x[2]))
gaussNewton(c(0, 0), f3)
# $xs 0.5671433 0.5671433

f4 <- function(x) # Dennis Schnabel
  c(x[1]^2 + x[2]^2 - 2, exp(x[1] - 1) + x[2]^3 - 2)
gaussNewton(c(2.0, 0.5), f4)
# $xs 1 1

## Examples (from Matlab)
F1 <- function(x) c(2*x[1]-x[2]-exp(-x[1]), -x[1]+2*x[2]-exp(-x[2]))
gaussNewton(c(-5, -5), F1)

# Find a matrix X such that X %*% X %*% X = [1 2; 3 4]
F2 <- function(x) {
  X <- matrix(x, 2, 2)
  D <- X %*% X %*% X - matrix(c(1,3,2,4), 2, 2)
  return(c(D))
}
sol <- gaussNewton(ones(2,2), F2)
(X <- matrix(sol$xs, 2, 2))
```



gcd, lcm

*GCD and LCM Integer Functions***Description**

Greatest common divisor and least common multiple

**Usage**

```
gcd(a, b, extended = FALSE)
Lcm(a, b)
```

**Arguments**

`a, b`                vectors of integers.  
`extended`            logical; if TRUE the extended Euclidean algorithm will be applied.

**Details**

Computation based on the extended Euclidean algorithm.

If both `a` and `b` are vectors of the same length, the greatest common divisor/lowest common multiple will be computed elementwise. If one is a vector, the other a scalar, the scalar will be replicated to the same length.

**Value**

A numeric (integer) value or vector of integers. Or a list of three vectors named `c`, `d`, `g`, `g` containing the greatest common divisors, such that

$$g = c * a + d * b.$$
**Note**

The following relation is always true:

$$n * m = \text{gcd}(n, m) * \text{lcm}(n, m)$$
**See Also**

`numbers::extGCD`

**Examples**

```
gcd(12, 1:24)
gcd(46368, 75025) # Fibonacci numbers are relatively prime to each other
Lcm(12, 1:24)
Lcm(46368, 75025) # = 46368 * 75025
```

---

geomean, harmmean      *Geometric and Harmonic Mean (Matlab Style)*

---

### Description

Geometric and harmonic mean along a dimension of a vector, matrix, or array.  
trimmean is almost the same as mean in R.

### Usage

```
geomean(x, dim = 1)
harmmean(x, dim = 1)

trimmean(x, percent = 0)
```

### Arguments

x	numeric vector, matrix, or array.
dim	dimension along which to take the mean; dim=1 means along columns, dim=2 along rows, the result will still be a row vector, not a column vector as in Matlab.
percent	percentage, between 0 and 100, of trimmed values.

### Details

trimmean does not call mean with the trim option, but rather calculates  $k \leftarrow \text{round}(n \times \text{percent} / 100 / 2)$  and leaves out  $k$  values at the beginning and end of the sorted  $x$  vector (or row or column of a matrix).

### Value

Returns a scalar or vector (or array) of geometric or harmonic means: For dim=1 the mean of columns, dim=2 the mean of rows, etc.

### Note

To have an exact analogue of mean(x) in Matlab, apply trimmean(x).

### See Also

[mean](#)

**Examples**

```

A <- matrix(1:12, 3, 4)
geomean(A, dim = 1)
## [1] 1.817121 4.932424 7.958114 10.969613
harmmean(A, dim = 2)
## [1] 2.679426 4.367246 5.760000

x <- c(-0.98, -0.90, -0.68, -0.61, -0.61, -0.38, -0.37, -0.32, -0.20, -0.16,
       0.00, 0.05, 0.12, 0.30, 0.44, 0.77, 1.37, 1.64, 1.72, 2.80)
trimmean(x); trimmean(x, 20) # 0.2 0.085
mean(x); mean(x, 0.10)      # 0.2 0.085

```

---

 geo\_median

*Geometric Median*


---

**Description**

Compute the “geometric median” of points in n-dimensional space, that is the point with the least sum of (Euclidean) distances to all these points.

**Usage**

```
geo_median(P, tol = 1e-07, maxiter = 200)
```

**Arguments**

P	matrix of points, x <sub>i</sub> -coordinates in the ith column.
tol	relative tolerance.
maxiter	maximum number of iterations.

**Details**

The task is solved applying an iterative process, known as Weiszfeld’s algorithm. The solution is unique whenever the points are not collinear.

If the dimension is 1 (one column), the median will be returned.

**Value**

Returns a list with components p the coordinates of the solution point, d the sum of distances to all the sample points, reltol the relative tolerance of the iterative process, and niter the number of iterations.

**Note**

This is also known as the “1-median problem” and can be generalized to the “k-median problem” for k cluster centers; see kcca in the ‘flexclust’ package.

**References**

See Wikipedia's entry on "Geometric median".

**See Also**

[L1linreg](#)

**Examples**

```
# Generate 100 points on the unit sphere in the 10-dim. space
set.seed(1001)
P <- rands(n=100, N=9)
( sol <- geo_median(P) )
# $p
# [1] -0.009481361 -0.007643410 -0.001252910  0.006437703 -0.019982885 -0.045337987
# [7]  0.036249563  0.003232175  0.035040592  0.046713023
# $d
# [1] 99.6638
# $reltol
# [1] 3.069063e-08
# $niter
# [1] 10
```

---

givens

*Givens Rotation*

---

**Description**

Givens Rotations and QR decomposition

**Usage**

```
givens(A)
```

**Arguments**

A                    numeric square matrix.

**Details**

givens(A) returns a QR decomposition (or factorization) of the square matrix A by applying unitary 2-by-2 matrices U such that  $U * [x_k; x_1] = [x, 0]$  where  $x = \sqrt{x_k^2 + x_1^2}$

**Value**

List with two matrices Q and R, Q orthonormal and R upper triangular, such that  $A = Q \%*\% R$ .

**References**

Golub, G. H., and Ch. F. van Loan (1996). *Matrix Computations*. Third edition, John Hopkins University Press, Baltimore.

**See Also**

[householder](#)

**Examples**

```
## QR decomposition
A <- matrix(c(0,-4,2, 6,-3,-2, 8,1,-1), 3, 3, byrow=TRUE)
gv <- givens(A)
(Q <- gv$Q); (R <- gv$R)
zapsmall(Q %*% R)

givens(magic(5))
```

---

gmres

*Generalized Minimal Residual Method*


---

**Description**

gmres(A,b) attempts to solve the system of linear equations  $A*x=b$  for x.

**Usage**

```
gmres(A, b, x0 = rep(0, length(b)),
      errtol = 1e-6, kmax = length(b)+1, reorth = 1)
```

**Arguments**

A	square matrix.
b	numerical vector or column vector.
x0	initial iterate.
errtol	relative residual reduction factor.
kmax	maximum number of iterations
reorth	reorthogonalization method, see Details.

**Details**

Iterative method for the numerical solution of a system of linear equations. The method approximates the solution by the vector in a Krylov subspace with minimal residual. The Arnoldi iteration is used to find this vector.

Reorthogonalization method:

- 1 – Brown/Hindmarsh condition (default)
- 2 – Never reorthogonalize (not recommended)
- 3 – Always reorthogonalize (not cheap!)



**Value**

Returns a list with components `x` the solution, `error` the vector of residual norms, and `niter` the number of iterations.

**Author(s)**

Based on Matlab code from C. T. Kelley's book, see references.

**References**

C. T. Kelley (1995). *Iterative Methods for Linear and Nonlinear Equations*. SIAM, Society for Industrial and Applied Mathematics, Philadelphia, USA.

**See Also**

[solve](#)

**Examples**

```
A <- matrix(c(0.46, 0.60, 0.74, 0.61, 0.85,
             0.56, 0.31, 0.80, 0.94, 0.76,
             0.41, 0.19, 0.15, 0.33, 0.06,
             0.03, 0.92, 0.15, 0.56, 0.08,
             0.09, 0.06, 0.69, 0.42, 0.96), 5, 5)
x <- c(0.1, 0.3, 0.5, 0.7, 0.9)
b <- A %*% x
gmres(A, b)
# $x
#      [,1]
# [1,] 0.1
# [2,] 0.3
# [3,] 0.5
# [4,] 0.7
# [5,] 0.9
#
# $error
# [1] 2.37446e+00 1.49173e-01 1.22147e-01 1.39901e-02 1.37817e-02 2.81713e-31
#
# $niter
# [1] 5
```

---

golden\_ratio

*Golden Ratio Search*

---

**Description**

Golden Ratio search for a univariate function minimum in a bounded interval.

**Usage**

```
golden_ratio(f, a, b, ..., maxiter = 100, tol = .Machine$double.eps^0.5)
```

**Arguments**

f	Function or its name as a string.
a, b	endpoints of the interval.
maxiter	maximum number of iterations.
tol	absolute tolerance; default <code>sqrt(eps)</code> .
...	Additional arguments to be passed to f.

**Details**

'Golden ratio' search for a univariate function minimum in a bounded interval.

**Value**

Return a list with components `xmin`, `fmin`, the function value at the minimum, `niter`, the number of iterations done, and the estimated precision `estim.prec`

**See Also**

[uniroot](#)

**Examples**

```
f <- function(x) x * cos(0.1*exp(x)) * sin(0.1*pi*exp(x))
golden_ratio(f, 0, 4, tol=10^-10) # $xmin = 3.24848329206212
optimize(f, c(0,4), tol=10^-10) # $minimum = 3.24848328971188
```

---

grad

*Numerical Gradient*

---

**Description**

Numerical function gradient.

**Usage**

```
grad(f, x0, heps = .Machine$double.eps^(1/3), ...)
```

**Arguments**

f	function of several variables.
x0	point where the gradient is to build.
heps	step size.
...	more variables to be passed to function f.

**Details**

Computes the gradient

$$\left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right)$$

numerically using the “central difference formula”.

**Value**

Vector of the same length as  $x_0$ .

**References**

Mathews, J. H., and K. D. Fink (1999). Numerical Methods Using Matlab. Third Edition, Prentice Hall.

**See Also**

[fderiv](#)

**Examples**

```
f <- function(u) {
  x <- u[1]; y <- u[2]; z <- u[3]
  return(x^3 + y^2 + z^2 + 12*x*y + 2*z)
}
x0 <- c(1,1,1)
grad(f, x0)      # 15 14  4      # direction of steepest descent

sum(grad(f, x0) * c(1, -1, 0)) # 1 , directional derivative

f <- function(x) x[1]^2 + x[2]^2
grad(f, c(0,0))  # 0 0 , i.e. a local optimum
```

---

gradient

*Discrete Gradient (Matlab Style)*

---

**Description**

Discrete numerical gradient.

**Usage**

```
gradient(F, h1 = 1, h2 = 1)
```

**Arguments**

F	vector of function values, or a matrix of values of a function of two variables.
h1	x-coordinates of grid points, or one value for the difference between grid points in x-direction.
h2	y-coordinates of grid points, or one value for the difference between grid points in y-direction.

**Details**

Returns the numerical gradient of a vector or matrix as a vector or matrix of discrete slopes in x- (i.e., the differences in horizontal direction) and slopes in y-direction (the differences in vertical direction).

A single spacing value, h, specifies the spacing between points in every direction, where the points are assumed equally spaced.

**Value**

If F is a vector, one gradient vector will be returned.

If F is a matrix, a list with two components will be returned:

X	numerical gradient/slope in x-direction.
Y	numerical gradient/slope in x-direction.

where each matrix is of the same size as F.

**Note**

TODO: If h2 is missing, it will not automatically be adapted.

**See Also**

[fderiv](#)

**Examples**

```
x <- seq(0, 1, by=0.2)
y <- c(1, 2, 3)
(M <- meshgrid(x, y))
gradient(M$X^2 + M$Y^2)
gradient(M$X^2 + M$Y^2, x, y)

## Not run:
# One-dimensional example
x <- seq(0, 2*pi, length.out = 100)
y <- sin(x)
f <- gradient(y, x)
max(f - cos(x))      #=> 0.00067086
plot(x, y, type = "l", col = "blue")
lines(x, cos(x), col = "gray", lwd = 3)
```

```
lines(x, f, col = "red")
grid()

# Two-dimensional example
v <- seq(-2, 2, by=0.2)
X <- meshgrid(v, v)$X
Y <- meshgrid(v, v)$Y

Z <- X * exp(-X^2 - Y^2)
image(v, v, t(Z))
contour(v, v, t(Z), col="black", add = TRUE)
grid(col="white")

grX <- gradient(Z, v, v)$X
grY <- gradient(Z, v, v)$Y

quiver(X, Y, grX, grY, scale = 0.2, col="blue")

## End(Not run)
```

---

gramSchmidt

*Gram-Schmidt*

---

## Description

Modified Gram-Schmidt Process

## Usage

```
gramSchmidt(A, tol = .Machine$double.eps^0.5)
```

## Arguments

A                    numeric matrix with  $nrow(A) \geq ncol(A)$ .  
tol                  numerical tolerance for being equal to zero.

## Details

The modified Gram-Schmidt process uses the classical orthogonalization process to generate step by step an orthonormal basis of a vector space. The modified Gram-Schmidt iteration uses orthogonal projectors in order to make the process numerically more stable.

## Value

List with two matrices Q and R, Q orthonormal and R upper triangular, such that  $A=Q\%*\%R$ .

## References

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Society for Industrial and Applied Mathematics, Philadelphia.

**See Also**

[householder](#), [givens](#)

**Examples**

```
## QR decomposition
A <- matrix(c(0,-4,2, 6,-3,-2, 8,1,-1), 3, 3, byrow=TRUE)
gs <- gramSchmidt(A)
(Q <- gs$Q); (R <- gs$R)
Q %*% R # = A
```

---

hadamard

*Hadamard Matrix*

---

**Description**

Generate Hadamard matrix of a certain size.

**Usage**

```
hadamard(n)
```

**Arguments**

n                    An integer of the form  $2^e$ ,  $12 \cdot 2^e$ , or  $20 \cdot 2^e$

**Details**

An n-by-n Hadamard matrix with  $n > 2$  exists only if  $\text{rem}(n, 4) = 0$ . This function handles only the cases where n,  $n/12$ , or  $n/20$  is a power of 2.

**Value**

Matrix of size n-by-n of orthogonal columns consisting of 1 and -1 only.

**Note**

Hadamard matrices have applications in combinatorics, signal processing, and numerical analysis.

**See Also**

[hankel](#), [Toeplitz](#)

**Examples**

```
hadamard(4)
H <- hadamard(8)
t(H)
```

---

halley	<i>Halley's Root Finding Method</i>
--------	-------------------------------------

---

**Description**

Finding roots of univariate functions using the Halley method.

**Usage**

```
halley(fun, x0, maxiter = 500, tol = 1e-08, ...)
```

**Arguments**

fun	function whose root is to be found.
x0	starting value for the iteration.
maxiter	maximum number of iterations.
tol	absolute tolerance; default $\text{eps}^{(1/2)}$
...	additional arguments to be passed to the function.

**Details**

Well known root finding algorithms for real, univariate, continuous functions; the second derivative must be smooth, i.e. continuous. The first and second derivative are computed numerically.

**Value**

Return a list with components `root`, `f.root`, the function value at the found root, `iter`, the number of iterations done, and the estimated precision `estim.prec`

**References**

<https://mathworld.wolfram.com/HalleysMethod.html>

**See Also**

[newtonRaphson](#)

**Examples**

```
halley(sin, 3.0)          # 3.14159265358979 in 3 iterations
halley(function(x) x*exp(x) - 1, 1.0)
                        # 0.567143290409784 Gauss' omega constant

# Legendre polynomial of degree 5
lp5 <- c(63, 0, -70, 0, 15, 0)/8
f <- function(x) polyval(lp5, x)
halley(f, 1.0)          # 0.906179845938664
```

---

`hampel`*Hampel Filter*

---

**Description**

Median absolute deviation (MAD) outlier in Time Series

**Usage**

```
hampel(x, k, t0 = 3)
```

**Arguments**

<code>x</code>	numeric vector representing a time series
<code>k</code>	window length $2*k+1$ in indices
<code>t0</code>	threshold, default is 3 (Pearson's rule), see below.

**Details**

The 'median absolute deviation' computation is done in the  $[-k \dots k]$  vicinity of each point at least  $k$  steps away from the end points of the interval. At the lower and upper end the time series values are preserved.

A high threshold makes the filter more forgiving, a low one will declare more points to be outliers.  $t0 < -3$  (the default) corresponds to Ron Pearson's 3 sigma edit rule,  $t0 < -0$  to John Tukey's median filter.

**Value**

Returning a list `L` with `L$y` the corrected time series and `L$ind` the indices of outliers in the 'median absolute deviation' sense.

**Note**

Don't take the expression *outlier* too serious. It's just a hint to values in the time series that appear to be unusual in the vicinity of their neighbors under a normal distribution assumption.

**References**

Pearson, R. K. (1999). "Data cleaning for dynamic modeling and control". European Control Conference, ETH Zurich, Switzerland.

**See Also**

[findpeaks](#)



**Examples**

```
set.seed(8421)
x <- numeric(1024)
z <- rnorm(1024)
x[1] <- z[1]
for (i in 2:1024) {
  x[i] <- 0.4*x[i-1] + 0.8*x[i-1]*z[i-1] + z[i]
}
omad <- hampel(x, k=20)

## Not run:
plot(1:1024, x, type="l")
points(omad$ind, x[omad$ind], pch=21, col="darkred")
grid()
## End(Not run)
```

---

hankel

*Hankel Matrix*

---

**Description**

Generate Hankel matrix from column and row vector

**Usage**

```
hankel(a, b)
```

**Arguments**

a                    vector that will be the first column  
b                    vector that if present will form the last row.

**Details**

hankel(a) returns the square Hankel matrix whose first column is a and whose elements are zero below the secondary diagonal. (I.e.: b may be missing.)

hankel(a, b) returns a Hankel matrix whose first column is a and whose last row is b. If the first element of b differs from the last element of a it is overwritten by this one.

**Value**

matrix of size (length(a), length(b))

**See Also**

[Toeplitz](#), [hadamard](#)

**Examples**

```
hankel(1:5, 5:1)
```

---

hausdorff\_dist      *Hausdorff Distance*

---

**Description**

Hausdorff distance (aka Hausdorff dimension)

**Usage**

```
hausdorff_dist(P, Q)
```

**Arguments**

P, Q                    numerical matrices, representing points in an m-dim. space.

**Details**

Calculates the Hausdorff Distance between two sets of points, P and Q. Sets P and Q must be matrices with the same number of columns (dimensions).

The 'directional' Hausdorff distance (dhd) is defined as:

$$\text{dhd}(P,Q) = \max_{p \text{ in } P} [ \min_{q \text{ in } Q} [ \|p-q\| ] ]$$

Intuitively dhd finds the point p from the set P that is farthest from any point in Q and measures the distance from p to its nearest neighbor in Q. The Hausdorff Distance is defined as  $\max(\text{dhd}(P,Q), \text{dhd}(Q,P))$ .

**Value**

A single scalar, the Hausdorff distance (dimension).

**References**

Barnsley, M. (1993). Fractals Everywhere. Morgan Kaufmann, San Francisco.

**See Also**

[distmat](#)

**Examples**

```
P <- matrix(c(1,1,2,2, 5,4,5,4), 4, 2)
Q <- matrix(c(4,4,5,5, 2,1,2,1), 4, 2)
hausdorff_dist(P, Q)    # 4.242641 = sqrt(sum((c(4,2)-c(1,5))^2))
```

---

`haversine`*Haversine Formula*

---

**Description**

Haversine formula to calculate the arc distance between two points on earth (i.e., along a great circle).

**Usage**

```
haversine(loc1, loc2, R = 6371.0)
```

**Arguments**

<code>loc1, loc2</code>	Locations on earth; for format see Details.
<code>R</code>	Average earth radius $R = 6371$ km, can be changed on input.

**Details**

The Haversine formula is more robust for the calculating the distance as with the spherical cosine formula. The user may want to assume a slightly different earth radius, so this can be provided as input.

The location can be input in two different formats, as latitude and longitude in a character string, e.g. for Frankfurt airport as '50 02 00N, 08 34 14E', or as a numerical two-vector in degrees (not radians).

Here for latitude 'N' and 'S' stand for North and South, and for longitude 'E' or 'W' stand for East and West. For the degrees format, South and West must be negative.

These two formats can be mixed.

**Value**

Returns the distance in km.

**Author(s)**

Hans W. Borchers

**References**

Entry 'Great\_circle\_distance' in Wikipedia.

**See Also**

Implementations of the Haversine formula can also be found in other R packages, e.g. 'geoPlot' or 'geosphere'.

**Examples**

```

FRA = '50 02 00N, 08 34 14E' # Frankfurt Airport
ORD = '41 58 43N, 87 54 17W' # Chicago O'Hare Interntl. Airport
fra <- c(50+2/60, 8+34/60+14/3600)
ord <- c(41+58/60+43/3600, -(87+54/60+17/3600))

dis <- haversine(FRA, ORD) # 6971.059 km
fprintf('Flight distance Frankfurt-Chicago is %8.3f km.\n', dis)

dis <- haversine(fra, ord)
fprintf('Flight distance Frankfurt-Chicago is %8.3f km.\n', dis)

```

---

hessenberg	<i>Hessenberg Matrix</i>
------------	--------------------------

---

**Description**

Generates the Hessenberg matrix for A.

**Usage**

```
hessenberg(A)
```

**Arguments**

A                    square matrix

**Details**

An (upper) Hessenberg matrix has zero entries below the first subdiagonal.

The function generates a Hessenberg matrix H and a unitary matrix P (a similarity transformation) such that  $A = P * H * t(P)$ .

The Hessenberg matrix has the same eigenvalues. If A is symmetric, its Hessenberg form will be a tridiagonal matrix.

**Value**

Returns a list with two elements,

H                    the upper Hessenberg Form of matrix A.  
H                    a unitary matrix.

**References**

Press, Teukolsky, Vetterling, and Flannery (2007). Numerical Recipes: The Art of Scientific Computing. 3rd Edition, Cambridge University Press. (Section 11.6.2)

**See Also**[householder](#)**Examples**

```

A <- matrix(c(-149,  -50,  -154,
              537,  180,   546,
              -27,   -9,  -25), nrow = 3, byrow = TRUE)
hb <- hessenberg(A)
hb
## $H
##          [,1]      [,2]      [,3]
## [1,] -149.0000  42.20367124 -156.316506
## [2,] -537.6783 152.55114875 -554.927153
## [3,]   0.0000   0.07284727   2.448851
##
## $P
##          [,1]      [,2]      [,3]
## [1,]    1 0.0000000 0.0000000
## [2,]    0 -0.9987384 0.0502159
## [3,]    0 0.0502159 0.9987384

hb$P %*% hb$H %*% t(hb$P)
##          [,1] [,2] [,3]
## [1,] -149  -50 -154
## [2,]  537  180  546
## [3,]  -27   -9  -25

```

hessian

*Hessian Matrix***Description**

Numerically compute the Hessian matrix.

**Usage**

```
hessian(f, x0, h = .Machine$double.eps^(1/4), ...)
```

**Arguments**

f                    univariate function of several variables.  
x0                    point in  $R^n$ .  
h                     step size.  
...                    variables to be passed to f.

**Details**

Computes the hessian matrix based on the three-point central difference formula, expanded to two variables.

Assumes that the function has continuous partial derivatives.

**Value**

An n-by-n matrix with  $\frac{\partial^2 f}{\partial x_i \partial x_j}$  as (i, j) entry.

**References**

Fausett, L. V. (2007). Applied Numerical Analysis Using Matlab. Second edition, Prentice Hall.

**See Also**

[hessdiag](#), [hessvec](#), [laplacian](#)

**Examples**

```
f <- function(x) cos(x[1] + x[2])
x0 <- c(0, 0)
hessian(f, x0)

f <- function(u) {
  x <- u[1]; y <- u[2]; z <- u[3]
  return(x^3 + y^2 + z^2 + 12*x*y + 2*z)
}
x0 <- c(1,1,1)
hessian(f, x0)
```

---

Hessian utilities      *Hessian utilities*

---

**Description**

Fast multiplication of Hessian and vector where computation of the full Hessian is not needed. Or determine the diagonal of the Hessian when non-diagonal entries are not needed or are nearly zero.

**Usage**

```
hessvec(f, x, v, csd = FALSE, ...)

hessdiag(f, x, ...)
```

**Arguments**

f                    function whose hessian is to be computed.  
 x                    point in  $R^n$ .  
 v                    vector of length n.  
 csd                  logocal, shall complex-step be applied.  
 ...                  more arguments to be passed to the function.

**Details**

hessvec computes the product of a Hessian of a function times a vector without deriving the full Hessian by approximating the gradient (see the reference). If the function allows for the complex-step method, the gradient can be calculated much more accurate (see `grad_csd`).

hessdiag computes only the diagonal of the Hessian by applying the central difference formula of second order to approximate the partial derivatives.

**Value**

hessvec returns the product  $H(f, x) * v$  as a vector.

hessdiag returns the diagonal of the Hessian of f.

**References**

B.A. Pearlmutter, Fast Exact Multiplication by the Hessian, Neural Computation (1994), Vol. 6, Issue 1, pp. 147-160.

**See Also**

[hessian](#)

**Examples**

```
## Not run:
set.seed(1237); n <- 100
a <- runif(n); b <- rnorm(n)
fn <- function(x, a, b) sum(exp(-a*x)*sin(b*pi*x))
x0 <- rep(1, n)
v0 <- rexp(n, rate=0.1)

# compute with full hessian
h0 <- hessian(fn, x0, a = a, b = b)           # n=100 runtimes
v1 <- c(h0 %*% v0)                            # 0.167 sec

v2 <- hessvec(fn, x0, v0, a = a, b = b)       # 0.00209 sec
v3 <- hessvec(fn, x0, v0, csd=TRUE,a=a, b=b)  # 0.00145 sec
v4 <- hessdiag(fn, x0, a = a, b = b) * v0     # 0.00204 sec

# compare with exact analytical Hessian
hex <- diag((a^2-b^2*pi^2)*exp(-a*x0)*sin(b*pi*x0) -
            2*a*b*pi*exp(-a*x0)*cos(b*pi*x0))
```

```
vex <- c(hex %**% v0)

max(abs(vex - v1))      # 2.48e-05
max(abs(vex - v2))      # 7.15e-05
max(abs(vex - v3))      # 0.09e-05
max(abs(vex - v4))      # 2.46e-05
## End(Not run)
```

---

hilb

*Hilbert Matrix*

---

### Description

Generate Hilbert matrix of dimension n

### Usage

```
hilb(n)
```

### Arguments

n                    positive integer specifying the dimension of the Hilbert matrix

### Details

Generate the Hilbert matrix H of dimension n with elements  $H[i, j] = 1/(i+j-1)$ .  
(Note: This matrix is ill-conditioned, see e.g. `det(hilb(6))`.)

### Value

matrix of dimension n

### See Also

[vander](#)

### Examples

```
hilb(5)
```



---

histc *Histogram Count (Matlab style)*

---

### Description

Histogram-like counting.

### Usage

```
histc(x, edges)
```

### Arguments

`x` numeric vector or matrix.  
`edges` numeric vector of grid points, must be monotonically non-decreasing.

### Details

`n = histc(x, edges)` counts the number of values in vector `x` that fall between the elements in the `edges` vector (which must contain monotonically nondecreasing values). `n` is a `length(edges)` vector containing these counts.

If `x` is a matrix then `cnt` and `bin` are matrices too, and

```
for (j in (1:n)) cnt[k,j] <- sum(bin[, j] == k)
```

### Value

returns a list with components `cnt` and `bin`. `n(k)` counts the number of values in `x` that lie between `edges(k) <= x(i) < edges(k+1)`. The last counts any values of `x` that match `edges(n)`. Values outside the values in `edges` are not counted. Use `-Inf` and `Inf` in `edges` to include all values.

`bin[i]` returns `k` if `edges(k) <= x(i) < edges(k+1)`, and `0` if `x[i]` lies outside the grid.

### See Also

[hist](#), [histss](#), [findInterval](#)

### Examples

```
x <- seq(0.0, 1.0, by = 0.05)
e <- seq(0.1, 0.9, by = 0.10)
histc(x, e)
# $cnt
# [1] 2 2 2 2 2 2 2 2 1
# $bin
# [1] 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 0 0

## Not run:
# Compare
```

```

findInterval(x, e)
# [1] 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 9
findInterval(x, e, all.inside = TRUE)
# [1] 1 1 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 8 8 8
# cnt[i] <- sum(findInterval(x, e) == i)
## End(Not run)

x <- matrix( c(0.5029, 0.2375, 0.2243, 0.8495,
               0.0532, 0.1644, 0.4215, 0.4135,
               0.7854, 0.0879, 0.1221, 0.6170), 3, 4, byrow = TRUE)
e <- seq(0.0, 1.0, by = 0.2)
histc(x, e)
# $cnt
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    1    0
# [2,]    0    1    1    0
# [3,]    1    0    1    1
# [4,]    1    0    0    1
# [5,]    0    0    0    1
# [6,]    0    0    0    0
#
# $bin
#      [,1] [,2] [,3] [,4]
# [1,]    3    2    2    5
# [2,]    1    1    3    3
# [3,]    4    1    1    4

```

---

histss

*Histogram Bin-width Optimization*


---

### Description

Method for selecting the bin size of time histograms.

### Usage

```
histss(x, n = 100, plotting = FALSE)
```

### Arguments

x	numeric vector or matrix.
n	maximum number of bins.
plotting	logical; shall a histogram be plotted.

### Details

Bin sizes of histograms are optimized in a way to best displays the underlying spike rate, for example in neurophysiological studies.

**Value**

Returns the same list as the `hist` function; the list is invisible if the histogram is plotted.

**References**

Shimazaki H. and S. Shinomoto. A method for selecting the bin size of a time histogram. *Neural Computation* (2007) Vol. 19(6), 1503-1527

**See Also**

[hist](#), [histc](#)

**Examples**

```
x <- sin(seq(0, pi/2, length.out = 200))
H <- histss(x, n = 50, plotting = FALSE)
## Not run:
plot(H, col = "gainsboro") # Compare with hist(x), or
hist(x, breaks = H$breaks) # the same
## End(Not run)
```

---

 hooke\_jeeves

*Hooke-Jeeves Function Minimization Method*


---

**Description**

An implementation of the Hooke-Jeeves algorithm for derivative-free optimization.

**Usage**

```
hooke_jeeves(x0, fn, ..., lb = NULL, ub = NULL, tol = 1e-08,
             maxfeval = 10000, target = Inf, info = FALSE)
```

**Arguments**

<code>x0</code>	starting vector.
<code>fn</code>	nonlinear function to be minimized.
<code>...</code>	additional arguments to be passed to the function.
<code>lb, ub</code>	lower and upper bounds.
<code>tol</code>	relative tolerance, to be used as stopping rule.
<code>maxfeval</code>	maximum number of allowed function evaluations.
<code>target</code>	iteration stops when this value is reached.
<code>info</code>	logical, whether to print information during the main loop.

**Details**

This method computes a new point using the values of  $f$  at suitable points along the orthogonal coordinate directions around the last point.

**Value**

List with following components:

xmin	minimum solution found so far.
fmin	value of $f$ at minimum.
count	number of function evaluations.
convergence	NOT USED at the moment.
info	special info from the solver.

**Note**

Hooke-Jeeves is notorious for its number of function calls. Memoization is often suggested as a remedy.

For a similar implementation of Hooke-Jeeves see the 'dfoptim' package.

**References**

C.T. Kelley (1999), *Iterative Methods for Optimization*, SIAM.

Quarteroni, Sacco, and Saleri (2007), *Numerical Mathematics*, Springer-Verlag.

**See Also**

[nelder\\_mead](#)

**Examples**

```
## Rosenbrock function
rosenbrock <- function(x) {
  n <- length(x)
  x1 <- x[2:n]
  x2 <- x[1:(n-1)]
  sum(100*(x1-x2^2)^2 + (1-x2)^2)
}

hooke_jeves(c(0,0,0,0), rosenbrock)
## $xmin
## [1] 1.000002 1.000003 1.000007 1.000013
## $fmin
## [1] 5.849188e-11
## $count
## [1] 1691
## $convergence
## [1] 0
## $info
```

```

## $info$solver
## [1] "Hooke-Jeeves"
## $info$iterations
## [1] 26

hooke_jeeves(rep(0,4), lb=rep(-1,4), ub=0.5, rosenbrock)
## $xmin
## [1] 0.50000000 0.26221320 0.07797602 0.00608027
## $fmin
## [1] 1.667875
## $count
## [1] 536
## $convergence
## [1] 0
## $info
## $info$solver
## [1] "Hooke-Jeeves"
## $info$iterations
## [1] 26

```

---

horner

*Horner's Rule*


---

## Description

Compute the value of a polynomial via Horner's Rule.

## Usage

```

horner(p, x)
hornerdefl(p, x)

```

## Arguments

**p** Numeric vector representing a polynomial.  
**x** Numeric scalar, vector or matrix at which to evaluate the polynomial.

## Details

horner utilizes the Horner scheme to evaluate the polynomial and its first derivative at the same time.

The polynomial  $p = p_1x^n + p_2x^{n-1} + \dots + p_nx + p_{n+1}$  is hereby represented by the vector  $p_1, p_2, \dots, p_n, p_{n+1}$ , i.e. from highest to lowest coefficient.

hornerdefl uses a similar approach to return the value of  $p$  at  $x$  and a polynomial  $q$  that satisfies

$$p(t) = q(t) * (t - x) + r, \text{ } r \text{ constant}$$

which implies  $r=0$  if  $x$  is a root of  $p$ . This will allow for a repeated root finding of polynomials.

**Value**

horner returns a list with two elements, `list(y=..., dy=...)` where the first list elements returns the values of the polynomial, the second the values of its derivative at the point(s) `x`.

hornerdefl returns a list `list(y=..., dy=...)` where `q` represents a polynomial, see above.

**Note**

For fast evaluation, there is no error checking for `p` and `x`, which both must be numerical vectors (`x` can be a matrix in horner).

**References**

Quarteroni, A., and Saleri, F. (2006) Scientific Computing with Matlab and Octave. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[polyval](#)

**Examples**

```
x <- c(-2, -1, 0, 1, 2)
p <- c(1, 0, 1) # polynomial x^2 + x, derivative 2*x
horner(p, x)$y  #=> 5 2 1 2 5
horner(p, x)$dy #=> -4 -2 0 2 4

p <- Poly(c(1, 2, 3)) # roots 1, 2, 3
hornerdefl(p, 3)      # q = x^2- 3 x + 2 with roots 1, 2
```

---

householder

*Householder Reflections*

---

**Description**

Householder reflections and QR decomposition

**Usage**

householder(A)

**Arguments**

A                    numeric matrix with `nrow(A) >= ncol(A)`.

**Details**

The Householder method applies a succession of elementary unitary matrices to the left of matrix `A`. These matrices are the so-called Householder reflections.

**Value**

List with two matrices Q and R, Q orthonormal and R upper triangular, such that  $A=Q \times R$ .

**References**

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Society for Industrial and Applied Mathematics, Philadelphia.

**See Also**

[givens](#)

**Examples**

```
## QR decomposition
A <- matrix(c(0,-4,2, 6,-3,-2, 8,1,-1), 3, 3, byrow=TRUE)
S <- householder(A)
(Q <- S$Q); (R <- S$R)
Q %% R # = A

## Solve an overdetermined linear system of equations
A <- matrix(c(1:8,7,4,2,3,4,2,2), ncol=3, byrow=TRUE)
S <- householder(A); Q <- S$Q; R <- S$R
m <- nrow(A); n <- ncol(A)
b <- rep(6, 5)

x <- numeric(n)
b <- t(Q) %% b
x[n] <- b[n] / R[n, n]
for (k in (n-1):1)
  x[k] <- (b[k] - R[k, (k+1):n] %% x[(k+1):n]) / R[k, k]
qr.solve(A, rep(6, 5)); x
```

---

humps

*Matlab Test Functions*

---

**Description**

Matlab test functions.

**Usage**

```
humps(x)
sinc(x)
psinc(x, n)
```

**Arguments**

x                    numeric scalar or vector.  
n                    positive integer.

**Details**

humps is a test function for finding zeros, for optimization and integration. Its root is at  $x = 1.2995$ , a (local) minimum at  $x = 0.6370$ , and the integral from  $0.5$  to  $1.0$  is  $8.0715$ .

sinc is defined as  $\text{sinc}(t) = \frac{\sin(\pi t)}{\pi t}$ . It is the continuous inverse Fourier transform of the rectangular pulse of width  $2\pi$  and height 1.

psinc is the 'periodic sinc function' and is defined as  $\text{psinc}(x, n) = \frac{\sin(xn/2)}{n \sin(x/2)}$ .

**Value**

Numeric scalar or vector.

**Examples**

```
## Not run:
plot(humps(), type="l"); grid()

x <- seq(0, 10, length=101)
plot(x, sinc(x), type="l"); grid()

## End(Not run)
```

---

hurstexp

*Hurst Exponent*


---

**Description**

Calculates the Hurst exponent using R/S analysis.

**Usage**

```
hurstexp(x, d = 50, display = TRUE)
```

**Arguments**

x	a time series.
d	smallest box size; default 50.
display	logical; shall the results be printed to the console?

**Details**

hurstexp(x) calculates the Hurst exponent of a time series x using R/S analysis, after Hurst, with slightly different approaches, or corrects it with small sample bias, see for example Weron.

These approaches are a corrected R/S method, an empirical and corrected empirical method, and a try at a theoretical Hurst exponent. It should be mentioned that the results are sometimes very different, so providing error estimates will be highly questionable.

Optimal sample sizes are automatically computed with a length that possesses the most divisors among series shorter than x by no more than 1 percent.



**Value**

hurstexp(x) returns a list with the following components:

- Hs - simplified R over S approach
- Hrs - corrected R over S Hurst exponent
- He - empirical Hurst exponent
- Hal - corrected empirical Hurst exponent
- Ht - theoretical Hurst exponent

**Note**

Derived from Matlab code of R. Weron, published on Matlab Central.

**References**

H.E. Hurst (1951) Long-term storage capacity of reservoirs, Transactions of the American Society of Civil Engineers 116, 770-808.

R. Weron (2002) Estimating long range dependence: finite sample properties and confidence intervals, Physica A 312, 285-299.

**See Also**

fractal::hurstSpec, RoverS, hurstBlock and fArma::LrdModelling

**Examples**

```
## Computing the Hurst exponent
data(brown72)
x72 <- brown72                # H = 0.72
xgn <- rnorm(1024)            # H = 0.50
xlm <- numeric(1024); xlm[1] <- 0.1 # H = 0.43
for (i in 2:1024) xlm[i] <- 4 * xlm[i-1] * (1 - xlm[i-1])

hurstexp(brown72, d = 128)    # 0.72
# Simple R/S Hurst estimation: 0.6590931
# Corrected R over S Hurst exponent: 0.7384611
# Empirical Hurst exponent: 0.7068613
# Corrected empirical Hurst exponent: 0.6838251
# Theoretical Hurst exponent: 0.5294909

hurstexp(xgn)                # 0.50
# Simple R/S Hurst estimation: 0.5518143
# Corrected R over S Hurst exponent: 0.5982146
# Empirical Hurst exponent: 0.6104621
# Corrected empirical Hurst exponent: 0.5690305
# Theoretical Hurst exponent: 0.5368124

hurstexp(xlm)                # 0.43
# Simple R/S Hurst estimation: 0.4825898
# Corrected R over S Hurst exponent: 0.5067766
```

```

# Empirical Hurst exponent:          0.4869625
# Corrected empirical Hurst exponent: 0.4485892
# Theoretical Hurst exponent:        0.5368124

## Compare with other implementations
## Not run:
library(fractal)

x <- x72
hurstSpec(x)           # 0.776   # 0.720
RoverS(x)              # 0.717
hurstBlock(x, method="aggAbs") # 0.648
hurstBlock(x, method="aggVar") # 0.613
hurstBlock(x, method="diffvar") # 0.714
hurstBlock(x, method="higuchi") # 1.001

x <- xgn
hurstSpec(x)           # 0.538   # 0.500
RoverS(x)              # 0.663
hurstBlock(x, method="aggAbs") # 0.463
hurstBlock(x, method="aggVar") # 0.430
hurstBlock(x, method="diffvar") # 0.471
hurstBlock(x, method="higuchi") # 0.574

x <- xlm
hurstSpec(x)           # 0.478   # 0.430
RoverS(x)              # 0.622
hurstBlock(x, method="aggAbs") # 0.316
hurstBlock(x, method="aggVar") # 0.279
hurstBlock(x, method="diffvar") # 0.547
hurstBlock(x, method="higuchi") # 0.998

## End(Not run)

```

---

hypot

*Hypotenuse Function*


---

### Description

Square root of sum of squares

### Usage

hypot(x, y)

### Arguments

x, y                    Vectors of real or complex numbers of the same size

**Details**

Element-by-element computation of the square root of the sum of squares of vectors resp. matrices  $x$  and  $y$ .

**Value**

Returns a vector or matrix of the same size.

**Note**

Returns  $c()$  if  $x$  or  $y$  is empty and the other one has length 1. If one input is scalar, the other a vector, the scalar will be extended to a vector of appropriate length. In all other cases,  $x$  and  $y$  have to be of the same size.

**Examples**

```
hypot(3,4)
hypot(1, c(3, 4, 5))
hypot(c(0, 0), c(3, 4))
```

---

iff

*Inverse Fast Fourier Transformation*

---

**Description**

Performs the inverse Fast Fourier Transform.

**Usage**

```
iff(x)
iffshift(x)
fftshift(x)
```

**Arguments**

$x$  a real or complex vector

**Details**

`iff` returns the value of the normalized discrete, univariate, inverse Fast Fourier Transform of the values in  $x$ .

`iffshift` and `fftshift` shift the zero-component to the center of the spectrum, that is swap the left and right half of  $x$ .

**Value**

Real or complex vector of the same length.

**Note**

Almost an alias for R's `fft(x, inverse=TRUE)`, but dividing by `length(x)`.

**See Also**

[fft](#)

**Examples**

```
x <- c(1, 2, 3, 4)
(y <- fft(x))
ifft(x)
ifft(y)

## Compute the derivative:  $F(df/dt) = (1i*k) * F(f)$ 
# hyperbolic secans f <- sech
df <- function(x) -sech(x) * tanh(x)
d2f <- function(x) sech(x) - 2*sech(x)^3
L <- 20 # domain [-L/2, L/2]
N <- 128 # number of Fourier nodes
x <- linspace(-L/2, L/2, N+1) # domain discretization
x <- x[1:N] # because of periodicity
dx <- x[2] - x[1] # finite difference
u <- sech(x) # hyperbolic secans
u1d <- df(x); u2d <- d2f(x) # first and second derivative
ut <- fft(u) # discrete Fourier transform
k <- (2*pi/L)*fftshift((-N/2):(N/2-1)) # shifted frequencies
u1 <- Re(ifft((1i*k) * ut)) # inverse transform
u2 <- Re(ifft(-k^2 * ut)) # first and second derivative
## Not run:
plot(x, u1d, type = "l", col = "blue")
points(x, u1)
grid()
figure()
plot(x, u2d, type = "l", col = "darkred")
points(x, u2)
grid()
## End(Not run)
```

---

inpolygon

*Polygon Region*

---

**Description**

Points inside polygon region.

**Usage**

```
inpolygon(x, y, xp, yp, boundary = FALSE)
```

**Arguments**

<code>x, y</code>	x-, y-coordinates of points to be tested for being inside the polygon region.
<code>xp, yp</code>	coordinates of the vertices specifying the polygon.
<code>boundary</code>	Logical; does the boundary belong to the interior.

**Details**

For a polygon defined by points (`xp, yp`), determine if the points (`x, y`) are inside or outside the polygon. The boundary can be included or excluded (default) for the interior.

**Value**

Logical vector, the same length as `x`.

**Note**

Special care taken for points on the boundary.

**References**

Hormann, K., and A. Agathos (2001). The Point in Polygon Problem for Arbitrary Polygons. *Computational Geometry*, Vol. 20, No. 3, pp. 131–144.

**See Also**

[polygon](#)

**Examples**

```
xp <- c(0.5, 0.75, 0.75, 0.5, 0.5)
yp <- c(0.5, 0.5, 0.75, 0.75, 0.5)
x <- c(0.6, 0.75, 0.6, 0.5)
y <- c(0.5, 0.6, 0.75, 0.6)
inpolygon(x, y, xp, yp, boundary = FALSE) # FALSE
inpolygon(x, y, xp, yp, boundary = TRUE)  # TRUE

## Not run:
pg <- matrix(c(0.15, 0.75, 0.25, 0.45, 0.70,
              0.80, 0.35, 0.55, 0.20, 0.90), 5, 2)
plot(c(0, 1), c(0, 1), type="n")
polygon(pg[,1], pg[,2])
P <- matrix(runif(20000), 10000, 2)
R <- inpolygon(P[, 1], P[, 2], pg[, 1], pg[,2])
clrs <- ifelse(R, "red", "blue")
points(P[, 1], P[, 2], pch = ".", col = clrs)
## End(Not run)
```

---

 integral

*Adaptive Numerical Integration*


---

**Description**

Combines several approaches to adaptive numerical integration of functions of one variable.

**Usage**

```
integral(fun, xmin, xmax,
         method = c("Kronrod", "Clenshaw", "Simpson"),
         no_intervals = 8, random = FALSE,
         reltol = 1e-8, abstol = 0, ...)
```

**Arguments**

fun	integrand, univariate (vectorized) function.
xmin, xmax	endpoints of the integration interval.
method	integration procedure, see below.
no_intervals	number of subdivisions at at start.
random	logical; shall the length of subdivisions be random.
reltol	relative tolerance.
abstol	absolute tolerance; not used.
...	additional parameters to be passed to the function.

**Details**

integral combines the following methods for adaptive numerical integration (also available as separate functions):

- Kronrod (Gauss-Kronrod)
- Clenshaw (Clenshaw-Curtis; not yet made adaptive)
- Simpson (adaptive Simpson)

Recommended default method is Gauss-Kronrod. Also try Clenshaw-Curtis that may be faster at times.

Most methods require that function *f* is vectorized. This will be checked and the function vectorized if necessary.

By default, the integration domain is subdivided into `no_intervals` subdomains to avoid 0 results if the support of the integrand function is small compared to the whole domain. If `random` is true, nodes will be picked randomly, otherwise forming a regular division.

If the interval is infinite, `quadinf` will be called that accepts the same methods as well. [If the function is array-valued, `quadv` is called that applies an adaptive Simpson procedure, other methods are ignored – not true anymore.]

**Value**

Returns the integral, no error terms given.

**Note**

integral does not provide ‘new’ functionality, everything is already contained in the functions called here. Other interesting alternatives are Gauss-Richardson (quadgr) and Romberg (romberg) integration.

**References**

Davis, Ph. J., and Ph. Rabinowitz (1984). *Methods of Numerical Integration*. Dover Publications, New York.

**See Also**

[quadgk](#), [quadgr](#), [quadcc](#), [simpadpt](#), [romberg](#), [quadv](#), [quadinf](#)

**Examples**

```
## Very smooth function
fun <- function(x) 1/(x^4+x^2+0.9)
val <- 1.582232963729353
for (m in c("Kron", "Clen", "Simp")) {
  Q <- integral(fun, -1, 1, reltol = 1e-12, method = m)
  cat(m, Q, abs(Q-val), "\n")}
# Kron 1.582233 3.197442e-13
# Rich 1.582233 3.197442e-13 # use quadgr()
# Clen 1.582233 3.199663e-13
# Simp 1.582233 3.241851e-13
# Romb 1.582233 2.555733e-13 # use romberg()

## Highly oscillating function
fun <- function(x) sin(100*pi*x)/(pi*x)
val <- 0.4989868086930458
for (m in c("Kron", "Clen", "Simp")) {
  Q <- integral(fun, 0, 1, reltol = 1e-12, method = m)
  cat(m, Q, abs(Q-val), "\n")}
# Kron 0.4989868 2.775558e-16
# Rich 0.4989868 4.440892e-16 # use quadgr()
# Clen 0.4989868 2.231548e-14
# Simp 0.4989868 6.328271e-15
# Romb 0.4989868 1.508793e-13 # use romberg()

## Evaluate improper integral
fun <- function(x) log(x)^2 * exp(-x^2)
val <- 1.9475221803007815976
Q <- integral(fun, 0, Inf, reltol = 1e-12)
# For infinite domains Gauss integration is applied!
cat(m, Q, abs(Q-val), "\n")
# Kron 1.94752218028062 2.01587635473288e-11
```

```
## Example with small function support
fun <- function(x)
  ifelse (x <= 0 | x >= pi, 0, sin(x))
integral(fun, -100, 100, no_intervals = 1)      # 0
integral(fun, -100, 100, no_intervals = 10)    # 1.99999999723
integral(fun, -100, 100, random=FALSE)        # 2
integral(fun, -100, 100, random=TRUE)         # 2 (sometimes 0 !)
integral(fun, -1000, 10000, random=FALSE)     # 0
integral(fun, -1000, 10000, random=TRUE)     # 0 (sometimes 2 !)
```

---

integral2

*Numerically Evaluate Double and Triple Integrals*


---

### Description

Numerically evaluate a double integral, resp. a triple integral by reducing it to a double integral.

### Usage

```
integral2(fun, xmin, xmax, ymin, ymax, sector = FALSE,
          reltol = 1e-6, abstol = 0, maxlist = 5000,
          singular = FALSE, vectorized = TRUE, ...)
```

```
integral3(fun, xmin, xmax, ymin, ymax, zmin, zmax,
          reltol = 1e-6, ...)
```

### Arguments

fun	function
xmin, xmax	lower and upper limits of x.
ymin, ymax	lower and upper limits of y.
zmin, zmax	lower and upper limits of z.
sector	logical.
reltol	relative tolerance.
abstol	absolute tolerance.
maxlist	maximum length of the list of rectangles.
singular	logical; are there singularities at vertices.
vectorized	logical; is the function fully vectorized.
...	additional parameters to be passed to the function.



**Details**

integral2 implements the ‘TwoD’ algorithm, that is Gauss-Kronrod with (3, 7)-nodes on 2D rectangles.

The borders of the domain of integration must be finite. The limits of y, that is y<sub>min</sub> and y<sub>max</sub>, can be constants or scalar functions of x that describe the lower and upper boundaries. These functions must be vectorized.

integral2 attempts to satisfy  $ERRBND \leq \max(AbsTol, RelTol * |Q|)$ . This is absolute error control when |Q| is sufficiently small and relative error control when |Q| is larger.

The function fun itself must be fully vectorized: It must accept arrays X and Y and return an array Z = f(X, Y) of corresponding values. If option vectorized is set to FALSE the procedure will enforce this vectorized behavior.

With sector=TRUE the region is a generalized sector that is described in polar coordinates (r,theta) by

$0 \leq a \leq \theta \leq b - a$  and b must be constants

$c \leq r \leq d - c$  and d can be constants or ...

... functions of theta that describe the lower and upper boundaries. Functions must be vectorized.

NOTE Polar coordinates are used only to describe the region – the integrand is f(x, y) for both kinds of regions.

integral2 can be applied to functions that are singular on a boundary. With value singular=TRUE, this option causes integral2 to use transformations to weaken singularities for better performance.

integral3 also accepts functions for the inner interval limits. y<sub>min</sub>, y<sub>max</sub> must be constants or functions of one variable (x), z<sub>min</sub>, z<sub>max</sub> constants or functions of two variables (x, y), all functions vectorized.

The triple integral will be first integrated over the second and third variable with integral2, and then integrated over a single variable with integral.

**Value**

Returns a list with Q the integral and error the error term.

**Note**

To avoid recursion, a possibly large matrix will be used and passed between subprograms. A more efficient implementation may be possible.

**Author(s)**

Copyright (c) 2008 Lawrence F. Shampine for Matlab code and description of the program; adapted and converted to R by Hans W Borchers.

**References**

Shampine, L. F. (2008). MATLAB Program for Quadrature in 2D. Proceedings of Applied Mathematics and Computation, 2008, pp. 266–274.

**See Also**

[integral](#), [cubature](#):[adaptIntegrate](#)

**Examples**

```

fun <- function(x, y) cos(x) * cos(y)
integral2(fun, 0, 1, 0, 1, reltol = 1e-10)
# $Q:      0.708073418273571 # 0.70807341827357119350 = sin(1)^2
# $error:  8.618277e-19      # 1.110223e-16

## Compute the volume of a sphere
f <- function(x, y) sqrt(1 - x^2 - y^2)
xmin <- 0; xmax <- 1
ymin <- 0; ymax <- function(x) sqrt(1 - x^2)
I <- integral2(f, xmin, xmax, ymin, ymax)
I$Q # 0.5236076 - pi/6 => 8.800354e-06

## Compute the volume over a sector
I <- integral2(f, 0, pi/2, 0, 1, sector = TRUE)
I$Q # 0.5236308 - pi/6 => 3.203768e-05

## Integrate 1/( sqrt(x + y)*(1 + x + y)^2 ) over the triangle
## 0 <= x <= 1, 0 <= y <= 1 - x. The integrand is infinite at (0,0).
f <- function(x,y) 1/( sqrt(x + y) * (1 + x + y)^2 )
ymax <- function(x) 1 - x
I <- integral2(f, 0, 1, 0, ymax)
I$Q + 1/2 - pi/4 # -3.247091e-08

## Compute this integral as a sector
rmax <- function(theta) 1/(sin(theta) + cos(theta))
I <- integral2(f, 0, pi/2, 0, rmax, sector = TRUE, singular = TRUE)
I$Q + 1/2 - pi/4 # -4.998646e-11

## Examples of computing triple integrals
f0 <- function(x, y, z) y*sin(x) + z*cos(x)
integral3(f0, 0, pi, 0, 1, -1, 1) # - 2.0 => 0.0

f1 <- function(x, y, z) exp(x+y+z)
integral3(f1, 0, 1, 1, 2, 0, 0.5)
## [1] 5.206447 # 5.20644655

f2 <- function(x, y, z) x^2 + y^2 + z
a <- 2; b <- 4
ymin <- function(x) x - 1
ymax <- function(x) x + 6
zmin <- -2
zmax <- function(x, y) 4 + y^2
integral3(f2, a, b, ymin, ymax, zmin, zmax)
## [1] 47416.75556 # 47416.755556

f3 <- function(x, y, z) sqrt(x^2 + y^2)
a <- -2; b <- 2

```

```

ymin <- function(x) -sqrt(4-x^2)
ymax <- function(x) sqrt(4-x^2)
zmin <- function(x, y) sqrt(x^2 + y^2)
zmax <- 2
integral3(f3, a, b, ymin, ymax, zmin, zmax)
## [1] 8.37758 # 8.377579076269617

```

---

interpl *One-dimensional Interpolation*

---

### Description

One-dimensional interpolation of points.

### Usage

```

interpl(x, y, xi = x,
        method = c("linear", "constant", "nearest", "spline", "cubic"))

```

### Arguments

x	Numeric vector; points on the x-axis; at least two points require; will be sorted if necessary.
y	Numeric vector; values of the assumed underlying function; x and y must be of the same length.
xi	Numeric vector; points at which to compute the interpolation; all points must lie between $\min(x)$ and $\max(x)$ .
method	One of "constant", "linear", "nearest", "spline", or "cubic"; default is "linear"

### Details

Interpolation to find  $y_i$ , the values of the underlying function at the points in the vector  $x_i$ .

Methods can be:

linear	linear interpolation (default)
constant	constant between points
nearest	nearest neighbor interpolation
spline	cubic spline interpolation
cubic	cubic Hermite interpolation

### Value

Numeric vector representing values at points  $x_i$ .

**Note**

Method ‘spline’ uses the spline approach by Moler et al., and is identical with the Matlab option of the same name, but slightly different from R’s spline function.

The Matlab option “cubic” seems to have no direct correspondence in R. Therefore, we simply use pchip here.

**See Also**

[approx](#), [spline](#)

**Examples**

```
x <- c(0.8, 0.3, 0.1, 0.6, 0.9, 0.5, 0.2, 0.0, 0.7, 1.0, 0.4)
y <- x^2
xi <- seq(0, 1, len = 81)
yl <- interp1(x, y, xi, method = "linear")
yn <- interp1(x, y, xi, method = "nearest")
ys <- interp1(x, y, xi, method = "spline")

## Not run:
plot(x, y); grid()
lines(xi, yl, col="blue", lwd = 2)
lines(xi, yn, col="black", lty = 2)
lines(xi, ys, col="red")

## End(Not run)

## Difference between spline (Matlab) and spline (R).
x <- 1:6
y <- c(16, 18, 21, 17, 15, 12)
xs <- linspace(1, 6, 51)
ys <- interp1(x, y, xs, method = "spline")
sp <- spline(x, y, n = 51, method = "fmm")

## Not run:
plot(x, y, main = "Matlab and R splines")
grid()
lines(xs, ys, col = "red")
lines(sp$x, sp$y, col = "blue")
legend(4, 20, c("Matlab spline", "R spline"),
      col = c("red", "blue"), lty = 1)

## End(Not run)
```

---

 interp2

*Two-dimensional Data Interpolation*


---

**Description**

Two-dimensional data interpolation similar to a table look-up.

**Usage**

```
interp2(x, y, Z, xp, yp, method = c("linear", "nearest", "constant"))
```

**Arguments**

x, y	vectors with monotonically increasing elements, representing x- and y-coordinates of the data values in Z.
Z	numeric length(y)-by-length(x) matrix.
xp, yp	x-, y-coordinates of points at which interpolated values will be computed.
method	interpolation method, "linear" the most useful.

**Details**

Computes a vector containing elements corresponding to the elements of xp and yp, determining by interpolation within the two-dimensional function specified by vectors x and y, and matrix Z.

x and y must be monotonically increasing. They specify the points at which the data Z is given. Therefore, length(x) = nrow(Z) and length(y) = ncol(Z) must be satisfied.

xp and yp must be of the same length.

The functions appears vectorized as xp, yp can be vectors, but internally they are treated in a for loop.

**Value**

Vector the length of xp of interpolated values.

For methods "constant" and "nearest" the intervals are considered closed from left and below. Out of range values are returned as NAs.

**Note**

The corresponding Matlab function has also the methods "cubic" and "spline". If in need of a nonlinear interpolation, take a look at barylag2d in this package and the example therein.

**See Also**

[interp1](#), [barylag2d](#)

**Examples**

```
## Not run:
x <- linspace(-1, 1, 11)
y <- linspace(-1, 1, 11)
mgrid <- meshgrid(x, y)
Z <- mgrid$X^2 + mgrid$Y^2
xp <- yp <- linspace(-1, 1, 101)

method <- "linear"
zp <- interp2(x, y, Z, xp, yp, method)
plot(xp, zp, type = "l", col = "blue")
```

```
method = "nearest"
zp <- interp2(x, y, Z, xp, yp, method)
lines(xp, zp, col = "red")
grid()
## End(Not run)
```

---

inv

*Matrix Inverse (Matlab Style)*

---

### Description

Invert a numeric or complex matrix.

### Usage

```
inv(a)
```

### Arguments

a                    real or complex square matrix

### Details

Computes the matrix inverse by calling `solve(a)` and catching the error if the matrix is nearly singular.

### Value

square matrix that is the inverse of a.

### Note

`inv()` is the function name used in Matlab/Octave.

### See Also

[solve](#)

### Examples

```
A <- hilb(6)
B <- inv(A)
B
# Compute the inverse matrix through Cramer's rule:
n <- nrow(A)
detA <- det(A)
b <- matrix(NA, nrow = n, ncol = n)
for (i in 1:n) {
```

```

    for (j in 1:n) {
      b[i, j] <- (-1)^(i+j) * det(A[-j, -i]) / detA
    }
  }
  b

```

---

 invlap

*Inverse Laplacian*


---

### Description

Numerical inversion of Laplace transforms.

### Usage

```
invlap(Fs, t1, t2, nnt, a = 6, ns = 20, nd = 19)
```

### Arguments

Fs	function representing the function to be inverse-transformed.
t1, t2	end points of the interval.
nnt	number of grid points between t1 and t2.
a	shift parameter; it is recommended to preserve value 6.
ns, nd	further parameters, increasing them leads to lower error.

### Details

The transform Fs may be any reasonable function of a variable  $s^a$ , where a is a real exponent. Thus, the function `invlap` can solve fractional problems and invert functions Fs containing (ir)rational or transcendental expressions.

### Value

Returns a list with components `x` the x-coordinates and `y` the y-coordinates representing the original function in the interval  $[t1, t2]$ .

### Note

Based on a presentation in the first reference. The function `invlap` on MatlabCentral (by ) served as guide. The Talbot procedure from the second reference could be an interesting alternative.

### References

J. Valsa and L. Brancik (1998). Approximate Formulae for Numerical Inversion of Laplace Transforms. Intern. Journal of Numerical Modelling: Electronic Networks, Devices and Fields, Vol. 11, (1998), pp. 153-166.

L.N.Trefethen, J.A.C.Weideman, and T.Schmelzer (2006). Talbot quadratures and rational approximations. BIT. Numerical Mathematics, 46(3):653–670.

**Examples**

```

Fs <- function(s) 1/(s^2 + 1)          # sine function
Li <- invlap(Fs, 0, 2*pi, 100)

## Not run:
plot(Li[[1]], Li[[2]], type = "l", col = "blue"); grid()

Fs <- function(s) tanh(s)/s           # step function
L1 <- invlap(Fs, 0.01, 20, 1000)
plot(L1[[1]], L1[[2]], type = "l", col = "blue")
L2 <- invlap(Fs, 0.01, 20, 2000, 6, 280, 59)
lines(L2[[1]], L2[[2]], col="darkred"); grid()

Fs <- function(s) 1/(sqrt(s)*s)
L1 <- invlap(Fs, 0.01, 5, 200, 6, 40, 20)
plot(L1[[1]], L1[[2]], type = "l", col = "blue"); grid()

Fs <- function(s) 1/(s^2 - 1)         # hyperbolic sine function
Li <- invlap(Fs, 0, 2*pi, 100)
plot(Li[[1]], Li[[2]], type = "l", col = "blue"); grid()

Fs <- function(s) 1/s/(s + 1)        # exponential approach
Li <- invlap(Fs, 0, 2*pi, 100)
plot(Li[[1]], Li[[2]], type = "l", col = "blue"); grid()

gamma <- 0.577215664901532           # Euler-Mascheroni constant
Fs <- function(s) -1/s * (log(s)+gamma) # natural logarithm
Li <- invlap(Fs, 0, 2*pi, 100)
plot(Li[[1]], Li[[2]], type = "l", col = "blue"); grid()

Fs <- function(s) (20.5+3.7343*s^1.15)/(21.5+3.7343*s^1.15+0.8*s^2.2+0.5*s^0.9)/s
L1 <- invlap(Fs, 0.01, 5, 200, 6, 40, 20)
plot(L1[[1]], L1[[2]], type = "l", col = "blue")
grid()
## End(Not run)

```

---

 isempty

*isempty Property*


---

**Description**

Determine if an object is empty.

**Usage**

```
isempty(x)
```

**Arguments**

x                    an R object



**Details**

An empty object has length zero.

**Value**

TRUE if x has length 0; otherwise, FALSE.

**Examples**

```
isempty(c(0))          # FALSE
isempty(matrix(0, 1, 0)) # TRUE
```

---

isposdef

*Positive Definiteness*

---

**Description**

Test for positive definiteness.

**Usage**

```
isposdef(A, psd = FALSE, tol = 1e-10)
```

**Arguments**

A	symmetric matrix
psd	logical, shall semi-positive definiteness be tested?
tol	tolerance to check symmetry and Cholesky decomposition.

**Details**

Whether matrix A is positive definite will be determined by applying the Cholesky decomposition. The matrix must be symmetric.

With psd=TRUE the matrix will be tested for being semi-positive definite. If not positive definite, still a warning will be generated.

**Value**

Returns TRUE or FALSE.

**Examples**

```
A <- magic(5)
# isposdef(A)
## [1] FALSE
## Warning message:
## In isposdef(A) : Matrix 'A' is not symmetric.
## FALSE

A <- t(A) %% A
isposdef(A)
## [1] TRUE

A[5, 5] <- 0
isposdef(A)
## [1] FALSE
```

---

isprime

*isprime Property*

---

**Description**

Vectorized version, returning for a vector or matrix of positive integers a vector of the same size containing 1 for the elements that are prime and 0 otherwise.

**Usage**

```
isprime(x)
```

**Arguments**

x                    vector or matrix of nonnegative integers

**Details**

Given an array of positive integers returns an array of the same size of 0 and 1, where the i indicates a prime number in the same position.

**Value**

array of elements 0, 1 with 1 indicating prime numbers

**See Also**

[factors](#), [primes](#)

**Examples**

```
x <- matrix(1:10, nrow=10, ncol=10, byrow=TRUE)
x * isprime(x)

# Find first prime number octett:
octett <- c(0, 2, 6, 8, 30, 32, 36, 38) - 19
while (TRUE) {
  octett <- octett + 210
  if (all(as.logical(isprime(octett)))) {
    cat(octett, "\n", sep=" ")
    break
  }
}
```

---

itersolve

*Iterative Methods*


---

**Description**

Iterative solutions of systems of linear equations.

**Usage**

```
itersolve(A, b, x0 = NULL, nmax = 1000, tol = .Machine$double.eps^(0.5),
          method = c("Gauss-Seidel", "Jacobi", "Richardson"))
```

**Arguments**

A	numerical matrix, square and non-singular.
b	numerical vector or column vector.
x0	starting solution for iteration; defaults to null vector.
nmax	maximum number of iterations.
tol	relative tolerance.
method	iterative method, Gauss-Seidel, Jacobi, or Richardson.

**Details**

Iterative methods are based on splitting the matrix  $A=(P-A)-A$  with a so-called ‘preconditioner’ matrix P. The methods differ in how to choose this preconditioner.

**Value**

Returns a list with components x the solution, iter the number of iterations, and method the name of the method applied.

**Note**

Richardson’s method allows to specify a ‘preconditioner’; this has not been implemented yet.

**References**

Quarteroni, A., and F. Saleri (2006). Scientific Computing with MATLAB and Octave. Springer-Verlag, Berlin Heidelberg.

**See Also**

[qrSolve](#)

**Examples**

```
N <- 10
A <- Diag(rep(3,N)) + Diag(rep(-2, N-1), k=-1) + Diag(rep(-1, N-1), k=1)
b <- A %*% rep(1, N)
x0 <- rep(0, N)

itersolve(A, b, tol = 1e-8, method = "Gauss-Seidel")
# [1] 1 1 1 1 1 1 1 1 1 1
# [1] 87
itersolve(A, b, x0 = 1:10, tol = 1e-8, method = "Jacobi")
# [1] 1 1 1 1 1 1 1 1 1 1
# [1] 177
```

---

jacobian

*Jacobian Matrix*

---

**Description**

Jacobian matrix of a function  $R^n \rightarrow R^m$ .

**Usage**

```
jacobian(f, x0, heps = .Machine$double.eps^(1/3), ...)
```

**Arguments**

f	m functions of n variables.
x0	Numeric vector of length n.
heps	This is h in the derivative formula.
...	parameters to be passed to f.

**Details**

Computes the derivative of each function  $f_j$  by variable  $x_i$  separately, taking the discrete step  $h$ .

**Value**

Numeric m-by-n matrix J where the entry  $J[j, i]$  is  $\frac{\partial f_j}{\partial x_i}$ , i.e. the derivatives of function  $f_j$  line up in row  $i$  for  $x_1, \dots, x_n$ .

**Note**

Obviously, this function is *not* vectorized.

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

gradient

**Examples**

```
## Example function from Quarteroni & Saleri
f <- function(x) c(x[1]^2 + x[2]^2 - 1, sin(pi*x[1]/2) + x[2]^3)
jf <- function(x)
  matrix( c(2*x[1], pi/2 * cos(pi*x[1]/2), 2*x[2], 3*x[2]^2), 2, 2)
all.equal(jf(c(1,1)), jacobian(f, c(1,1)))
# TRUE
```

---

kriging

*Interpolation by Kriging*


---

**Description**

Simple and ordinary Kriging interpolation and interpolating function.

**Usage**

```
kriging(u, v, u0, type = c("ordinary", "simple"))
```

**Arguments**

u	an $n \times m$ -matrix of $n$ points in the $m$ -dimensional space.
v	an $n$ -dim. (column) vector of interpolation values.
u0	a $k \times m$ -matrix of $k$ points in $R^m$ to be interpolated.
type	character; values 'simple' or 'ordinary'; no partial matching.

**Details**

Kriging is a geo-spatial estimation procedure that estimates points based on the variations of known points in a non-regular grid. It is especially suited for surfaces.

**Value**

kriging returns a  $k$ -dim. vektor of interpolation values.

**Note**

In the literature, different versions and extensions are discussed.

**References**

Press, W. H., A. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (2007). Numerical recipes: The Art of Scientific Computing (3rd Ed.). Cambridge University Press, New York, Sect. 3.7.4, pp. 144-147.

**See Also**

[akimaInterp](#), [barylag2d](#), package `kriging`

**Examples**

```
## Interpolate the Saddle Point function
f <- function(x) x[1]^2 - x[2]^2      # saddle point function

set.seed(8237)
n <- 36
x <- c(1, 1, -1, -1, runif(n-4, -1, 1)) # add four vertices
y <- c(1, -1, 1, -1, runif(n-4, -1, 1))
u <- cbind(x, y)
v <- numeric(n)
for (i in 1:n) v[i] <- f(c(x[i], y[i]))

kriging(u, v, c(0, 0))                #=> 0.006177183
kriging(u, v, c(0, 0), type = "simple") #=> 0.006229557

## Not run:
xs <- linspace(-1, 1, 101)           # interpolation on a diagonal
u0 <- cbind(xs, xs)

yo <- kriging(u, v, u0, type = "ordinary") # ordinary kriging
ys <- kriging(u, v, u0, type = "simple")   # simple kriging
plot(xs, ys, type = "l", col = "blue", ylim = c(-0.1, 0.1),
      main = "Kriging interpolation along the diagonal")
lines(xs, yo, col = "red")
legend(-1.0, 0.10, c("simple kriging", "ordinary kriging", "function"),
      lty = c(1, 1, 1), lwd = c(1, 1, 2), col=c("blue", "red", "black"))
grid()
lines(c(-1, 1), c(0, 0), lwd = 2)
## End(Not run)

## Find minimum of the sphere function
f <- function(x, y) x^2 + y^2 + 100
v <- bsxfun(f, x, y)

ff <- function(w) kriging(u, v, w)
ff(c(0, 0))                #=> 100.0317
## Not run:
optim(c(0.0, 0.0), ff)
```

```
# $par: [1] 0.04490075 0.01970690
# $value: [1] 100.0291
ezcontour(ff, c(-1, 1), c(-1, 1))
points(0.04490075, 0.01970690, col = "red")
## End(Not run)
```

---

kron	<i>Kronecker product (Matlab Style)</i>
------	---

---

## Description

Kronecker tensor product of two matrices.

## Usage

```
kron(a, b)
```

## Arguments

a	real or complex matrix
b	real or complex matrix

## Details

The Kronecker product is a large matrix formed by all products between the elements of a and those of b. The first left block is  $a_{11} * b$ , etc.

## Value

an  $(n * p \times m * q)$ -matrix, if a is  $(n \times m)$  and b is  $(p \times q)$ .

## Note

`kron()` is an alias for the R function `kronecker()`, which can also be executed with the binary operator `'%x%'`.

## Examples

```
a <- diag(1, 2, 2)
b <- matrix(1:4, 2, 2)
kron(a, b)
kron(b, a)
```

---

`L1linreg`*L1 Linear Regression*

---

**Description**

Solve the linear system  $Ax = b$  in an  $L_p$  sense, that is minimize the term  $\sum |b - Ax|^p$ . The case  $p=1$  is also called “least absolute deviation” (LAD) regression.

**Usage**

```
L1linreg(A, b, p = 1, tol = 1e-07, maxiter = 200)
```

**Arguments**

<code>A</code>	matrix of independent variables.
<code>b</code>	independent variables.
<code>p</code>	the $p$ in $L^p$ norm, $p \leq 1$ .
<code>tol</code>	relative tolerance.
<code>maxiter</code>	maximum number of iterations.

**Details**

$L1/L_p$  regression is here solved applying the “iteratively reweighted least square” (IRLS) method in which each step involves a weighted least squares problem.

If an intercept term is required, add a unit column to  $A$ .

**Value**

Returns a list with components `x` the linear coefficients describing the solution, `reltol` the relative tolerance reached, and `niter` the number of iterations.

**Note**

In this case of  $p=1$ , the problem would be better approached by use of linear programming methods.

**References**

Dasgupta, M., and S.K. Mishra (2004). Least absolute deviation estimation of linear econometric models: A literature review. MPRA Paper No. 1781.

**See Also**

[lm](#), [lsqnonlin](#), [quantreg::rq](#)



**Examples**

```

m <- 101; n <- 10      # no. of data points, degree of polynomial
x <- seq(-1, 1, len=m)
y <- runge(x)         # Runge's function
A <- outer(x, n:0, '^') # Vandermonde matrix
b <- y

( sol <- L1linreg(A, b) )
# $x
# [1] -21.93242  0.00000  62.91092  0.00000 -67.84854  0.00000
# [7]  34.14400  0.00000  -8.11899  0.00000  0.84533
#
# $reltol
# [1] 6.712355e-10
#
# $niter
# [1] 81

# minimum value of polynomial L1 regression
sum(abs(polyval(sol$x, x) - y))
# [1] 3.061811

```

---

laguerre

*Laguerre's Method*


---

**Description**

Laguerre's method for finding roots of complex polynomials.

**Usage**

```
laguerre(p, x0, nmax = 25, tol = .Machine$double.eps^(1/2))
```

**Arguments**

p	real or complex vector representing a polynomial.
x0	real or complex point near the root.
nmax	maximum number of iterations.
tol	absolute tolerance.

**Details**

Uses values of the polynomial and its first and second derivative.

**Value**

The root found, or a warning about the number of iterations.

**Note**

Computations are carried out in complex arithmetic, and it is possible to obtain a complex root even if the starting estimate is real.

**References**

Fausett, L. V. (2007). Applied Numerical Analysis Using Matlab. Second edition, Prentice Hall.

**See Also**

[roots](#)

**Examples**

```
# 1 x^5 - 5.4 x^4 + 14.45 x^3 - 32.292 x^2 + 47.25 x - 26.46
p <- c(1.0, -5.4, 14.45, -32.292, 47.25, -26.46)
laguerre(p, 1)  #=> 1.2
laguerre(p, 2)  #=> 2.099987      (should be 2.1)
laguerre(p, 2i) #=> 0+2.236068i  (+- 2.2361i, i.e sqrt(-5))
```

---

lambertWp

*Lambert's W Function*


---

**Description**

Principal real branch of the Lambert W function.

**Usage**

```
lambertWp(x)
lambertWn(x)
```

**Arguments**

x                    Numeric vector of real numbers  $\geq -1/e$ .

**Details**

The Lambert W function is the inverse of  $x \rightarrow x e^x$ , with two real branches,  $W_0$  for  $x \geq -1/e$  and  $W_{-1}$  for  $-1/e \leq x < 0$ . Here the principal branch is called `lambertWp`, the other one `lambertWn`, computed for real  $x$ .

The value is calculated using an iteration that stems from applying Halley's method. This iteration is quite fast and accurate.

The functions is not really vectorized, but at least returns a vector of values when presented with a numeric vector of length  $\geq 2$ .

**Value**

Returns the solution  $w$  of  $w \cdot \exp(w) = x$  for real  $x$  with NaN if  $x < 1/\exp(1)$  (resp.  $x \geq 0$  for the second branch).

**Note**

See the examples how values for the second branch or the complex Lambert W function could be calculated by Newton's method.

**References**

Corless, R. M., G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, and D. E. Knuth (1996). On the Lambert W Function. *Advances in Computational Mathematics*, Vol. 5, pp. 329-359.

**See Also**

[halley](#)

**Examples**

```
## Examples
lambertWp(0)           #=> 0
lambertWp(1)           #=> 0.5671432904097838... Omega constant
lambertWp(exp(1))      #=> 1
lambertWp(-log(2)/2)  #=> -log(2)

# The solution of  $x * a^x = z$  is  $W(\log(a)*z)/\log(a)$ 
#  $x * 123^{(x-1)} = 3$ 
lambertWp(3*123*log(123))/log(123) #=> 1.19183018...

x <- seq(-0.35, 0.0, by=0.05)
w <- lambertWn(x)
w * exp(w)           # max. error < 3e-16
# [1] -0.35 -0.30 -0.25 -0.20 -0.15 -0.10 -0.05 NaN

## Not run:
xs <- c(-1/exp(1), seq(-0.35, 6, by=0.05))
ys <- lambertWp(xs)
plot(xs, ys, type="l", col="darkred", lwd=2, ylim=c(-2,2),
     main="Lambert W0 Function", xlab="", ylab="")
grid()
points(c(-1/exp(1), 0, 1, exp(1)), c(-1, 0, lambertWp(1), 1))
text(1.8, 0.5, "Omega constant")

## End(Not run)

## Analytic derivative of lambertWp (similar for lambertWn)
D_lambertWp <- function(x) {
  xw <- lambertWp(x)
  1 / (1+xw) / exp(xw)
}
```

```

D_lambertWp(c(-1/exp(1), 0, 1, exp(1)))
# [1] Inf 1.0000000 0.3618963 0.1839397

## Second branch resp. the complex function lambertWm()
F <- function(xy, z0) {
  z <- xy[1] + xy[2]*1i
  fz <- z * exp(z) - z0
  return(c(Re(fz), Im(fz)))
}
newtonsys(F, c(-1, -1), z0 = -0.1) #=> -3.5771520639573
newtonsys(F, c(-1, -1), z0 = -pi/2) #=> -1.5707963267949i = -pi/2 * 1i

```

---

laplacian

*Laplacian Operator*


---

### Description

Numerically compute the Laplacian of a function.

### Usage

```
laplacian(f, x0, h = .Machine$double.eps^(1/4), ...)
```

### Arguments

f	univariate function of several variables.
x0	point in $R^n$ .
h	step size.
...	variables to be passed to f.

### Details

Computes the Laplacian operator  $f_{x_1x_1} + \dots + f_{x_nx_n}$  based on the three-point central difference formula, expanded to this special case.

Assumes that the function has continuous partial derivatives.

### Value

Real number.

### References

Fausett, L. V. (2007). Applied Numerical Analysis Using Matlab. Second edition, Prentice Hall.

### See Also

[hessian](#)

**Examples**

```
f <- function(x) x[1]^2 + 2*x[1]*x[2] + x[2]^2
laplacian(f, c(1,1))
```

---

lebesgue	<i>Lebesgue Constant</i>
----------	--------------------------

---

**Description**

Estimates the Lebesgue constant.

**Usage**

```
lebesgue(x, refine = 4, plotting = FALSE)
```

**Arguments**

x	numeric vector of grid points
refine	refine the grid with $2^{\text{refine}}$ grid points; can only be an integer between 2 and 10, default 4.
plotting	shall the Lebesgue function be plotted.

**Details**

The Lebesgue constant gives an estimation  $\|P_n f\| \leq L \|f\|$  (in minimax norm) where  $P_n f$  is the interpolating polynomial of order  $n$  for  $f$  on an interval  $[a, b]$ .

**Value**

Lebesgue constant for the given grid points.

**Note**

The Lebesgue constant plays an important role when estimating the distance of interpolating polynomials from the minimax solution (see the Remez algorithm).

**References**

Berrut, J.-P., and L. Nick Trefethen (2004). “Barycentric Lagrange Interpolation”. *SIAM Review*, Vol. 46(3), pp.501–517.

**See Also**

[barylag](#)

**Examples**

```
lebesgue(seq(0, 1, length.out = 6)) #=> 3.100425
```

---

 legendre

*Legendre Functions (Matlab Style)*


---

**Description**

Calculate the values of (associated) Legendre functions.

**Usage**

```
legendre(n, x)
```

**Arguments**

n	degree of the Legendre polynomial involved.
x	real points to evaluate Legendre's functions at.

**Details**

legendre(n, x) computes the associated Legendre functions of degree n and order  $m=0, 1, \dots, n$ , evaluated for each element of x where x must contain real values in  $[-1, 1]$ .

If x is a vector, then  $L=legendre(n, x)$  is an  $(n+1)$ -by- $N$  matrix, where  $N=length(x)$ . Each element  $L[m+1, i]$  corresponds to the associated Legendre function of degree legendre(n, x) and order m evaluated at  $x[i]$ .

Note that the first row of L is the Legendre polynomial evaluated at x.

**Value**

Returns a matrix of size  $(n+1)$ -by- $N$  where  $N=length(x)$ .

**Note**

Legendre functions are solutions to Legendre's differential equation (it occurs when solving Laplace's equation in spherical coordinates).

**See Also**

[chebPoly](#)

**Examples**

```
x <- c(0.0, 0.1, 0.2)
legendre(2, x)
#      [,1]      [,2]      [,3]
# [1,] -0.5 -0.4850000 -0.4400000
# [2,]  0.0 -0.2984962 -0.5878775
# [3,]  3.0  2.9700000  2.8800000

## Not run:
```

```

x <- seq(0, 1, len = 50)
L <- legendre(2, x)
plot(x, L[1, ], type = "l", col = 1, ylim = c(-2, 3), ylab = "y",
      main = "Legendre Functions of degree 2")
lines(x, L[2, ], col = 2)
lines(x, L[3, ], col = 3)
grid()
## End(Not run)
## Generate Legendre's Polynomial as function
# legendre_P <- function(n, x) {
#   L <- legendre(n, x)
#   return(L[1, ])
# }

```

---

linearproj, affineproj

*Linear Projection onto a Subspace*

---

### Description

Computes the projection of points in the columns of B onto the linear subspace spanned by the columns of A, resp. the projection of a point onto an affine subspace and its distance.

### Usage

```
linearproj(A, B)
```

```
affineproj(x0, C, b, unbound = TRUE, maxniter = 100)
```

### Arguments

A	Matrix whose columns span a subspace of some $\mathbb{R}^n$ .
B	Matrix whose columns are to be projected.
x0	Point in $\mathbb{R}^n$ to be projected onto $Cx = b$ .
C, b	Matrix and vector, defining an affine subspace as $Cx = b$
unbound	Logical; require all $x \geq 0$ if unbound is false.
maxniter	Maximum number of iterations (if is unbound is false).

### Details

linearproj projects points onto a *linear* subspace in  $\mathbb{R}^n$ . The columns of A are assumed be the basis of a linear subspace, esp. they are required to be linearly independent. The columns of matrix B define points in  $\mathbb{R}^n$  that will be projected onto A, and their resp. coefficients in terms of the basis in A are computed.

The columns of A need to be linearly independent; if not, generate an orthonormal basis of this subspace with `orth(A)`. If you want to project points onto a subspace that is defined by  $Ax = 0$ , then generate an orthonormal basis of the nullspace of A with `null(A)`.

Technically, the orthogonal projection can be determined by a finite 'Fourier expansion' with coefficients calculated as scalar products, see the examples.

affineproj projects (single) points onto an affine subspace defined by  $Ax = b$  and calculates the distance of  $x_0$  from this subspace. The calculation is based on the following formula:

$$p = (I - A'(AA')^{-1})x_0 + A'(AA')^{-1}b$$

Technically, if  $a$  is one solution of  $Cx = b$ , then the projection onto  $C$  can be derived from the projection onto  $S = \{Cx = 0\}$  with  $\text{proj}_C(x) = a + \text{proj}_S(x - a)$ , see the examples.

In case the user requests the coordinates of the projected point to be positive, an iteration procedure is started where negative coordinates are set to zero in each iteration.

### Value

The functions linearproj returns a list with components P and Q. The columns of P contain the coefficients – in the basis of A – of the corresponding projected points in B, and the columns of Q are the the coordinates of these points in the natural coordinate system of  $R^n$ .

affineproj returns a list with components proj, dist, and niter. proj is the projected point, dist the distance from the subspace (and niter the number of iterations if positivity of the coordinates was requested.).

### Note

Some timings show that these implementations are to a certain extent competitive with direct applications of quadprog.

### Author(s)

Hans W. Borchers, partly based on code snippets by Ravi Varadhan.

### References

G. Strang (2006). Linear Algebra and Its Applications. Fourth Edition, Cengage Learning, Boston, MA.

### See Also

[nullspace](#), [orth](#)

### Examples

```
#-- Linear projection -----
# Projection onto the line (1,1,1) in R^3
A <- matrix(c(1,1,1), 3, 1)
B <- matrix(c(1,0,0, 1,2,3, -1,0,1), 3, 3)
S <- linearproj(A, B)
## S$Q
##      [,1] [,2] [,3]
## [1,] 0.3333333 2 0
```



```

## [2,] 0.3333333 2 0
## [3,] 0.3333333 2 0

# Fourier expansion': sum(<x0, a_i> a_i /<a_i, a_i>), a_i = A[,i]
dot(c(1,2,3), A) * A / dot(A, A) # A has only one column

#-- Affine projection -----

# Projection onto the (hyper-)surface x+y+z = 1 in R^3
A <- t(A); b <- 1
x0 <- c(1,2,3)
affineproj(x0, A, b) # (-2/3, 1/3, 4/3)

# Linear translation: Let S be the linear subspace and A the parallel
# affine subspace of A x = b, a the solution of the linear system, then
# proj_A(x) = a + proj_S(x-a)
a <- qr.solve(A, b)
A0 <- nullspace(A)
xp <- c(a + linearproj(A0, x0 - a)$Q)
## [1] -0.6666667 0.3333333 1.3333333

#-- Projection with positivity ----- 24 ms -- 1.3 s --
s <- affineproj(x0, A, b, unbound = FALSE)
zapsmall(s$proj) # [1] 0 0 1
## $x : 0.000000e+00 3.833092e-17 1.000000e+00
## $niter : 35

#-- Extended Example ----- 80 ms --
## Not run:
set.seed(65537)
n = 1000; m = 100 # dimension, codimension
x0 <- rep(0, n) # project (0, ..., 0)
A <- matrix(runif(m*n), nrow = m) # 100 x 1000
b <- rep(1, m) # A x = b, linear system
a <- qr.solve(A, b) # A a = b, LS solution
A0 <- nullspace(A) # 1000 x 900, base of <A>
xp <- a+drop(A0 %*% dot(x0-a, A0)) # projection
Norm(xp - x0) # [1] 0.06597077

## End(Not run)

#-- Solution with quadprog ----- 40 ms --
# D <- diag(1, n) # quadratic form
# A1 <- rbind(A, diag(1, n)) # A x = b and
# b1 <- c(b, rep(0, n)) # x >= 0
# n <- nrow(A)
# sol = quadprog::solve.QP(D, x0, t(A1), b1, meq = n)
# xp <- sol$solution

#-- Solution with CVXR ----- 50 ms --
# library(CVXR)
# x = Variable(n) # n decision variables
# objective = Minimize(p_norm(x0 - x)) # min! || p0 - x ||

```

```
# constraint = list(A %% x == b, x >= 0) # A x = b, x >= 0
# problem = Problem(objective, constraint)
# solution = solve(problem) # Solver: ECOS
# solution$value #
# xp <- solution$getValue(x) #
```

---

line_integral	<i>Line integral (in the complex plane)</i>
---------------	---

---

### Description

Provides complex line integrals.

### Usage

```
line_integral(fun, waypoints, method = NULL, reltol = 1e-8, ...)
```

### Arguments

fun	integrand, complex (vectorized) function.
method	integration procedure, see below.
waypoints	complex integration: points on the integration curve.
reltol	relative tolerance.
...	additional parameters to be passed to the function.

### Details

line\_integral realizes complex line integration, in this case straight lines between the waypoints. By passing discrete points densely along the curve, arbitrary line integrals can be approximated.

line\_integral will accept the same methods as integral; default is integrate from Base R.

### Value

Returns the integral, no error terms given.

### See Also

[integral](#)

**Examples**

```
## Complex integration examples
points <- c(0, 1+1i, 1-1i, 0)           # direction mathematically negative
f <- function(z) 1 / (2*z -1)
I <- line_integral(f, points)
abs(I - (0-pi*1i))                     # 0 ; residuum 2 pi 1i * 1/2

f <- function(z) 1/z
points <- c(-1i, 1, 1i, -1, -1i)
I <- line_integral(f, points)           # along a rectangle around 0+0i
abs(I - 2*pi*1i)                       #=> 0 ; residuum: 2 pi i * 1

N <- 100
x <- linspace(0, 2*pi, N)
y <- cos(x) + sin(x)*1i
J <- line_integral(f, waypoints = y)    # along a circle around 0+0i
abs(I - J)                              #=> 5.015201e-17; same residuum
```

linprog

*Linear Programming Solver***Description**

Solves simple linear programming problems, allowing for inequality and equality constraints as well as lower and upper bounds.

**Usage**

```
linprog(cc, A = NULL, b = NULL, Aeq = NULL, beq = NULL,
        lb = NULL, ub = NULL, x0 = NULL, I0 = NULL,
        bigM = 100, maxiter = 20, maximize = FALSE)
```

**Arguments**

cc	defines the linear objective function.
A	matrix representing the inequality constraints $A x \leq b$ .
b	vector, right hand side of the inequalities.
Aeq	matrix representing the equality constraints $Aeq x \leq beq$ .
beq	vector, right hand side of the inequalities.
lb	lower bounds, if not NULL must all be greater or equal 0.
ub	upper bounds, if not NULL must all be greater or equal lb.
x0	feasible base vector, will not be used at the moment.
I0	index set of x0, will not be used at the moment.
bigM	big-M constant, will be used for finding a base vector.
maxiter	maximum number of iterations.
maximize	logical; shall the objective be minimized or maximized?

**Details**

Solves linear programming problems of the form  $\min c' * x$  such that

$$A * x \leq b$$

$$A_{eq} * x = b_{eq}$$

$$lb \leq x \leq ub$$

**Value**

List with

- x the solution vector.
- fval the value at the optimal solution.
- errno, message the error number and message.

**Note**

This is a first version that will be unstable at times. For real linear programming problems use package lpSolve.

**Author(s)**

HwB <hwborchers@googlemail.com>

**References**

Vanderbei, R. J. (2001). Linear Programming: Foundations and Extensions. Princeton University Press.

Eiselt, H. A., and C.-L. Sandblom (2012). Operations Research: A Model-based Approach. Springer-Verlag, Berlin Heidelberg.

**See Also**

linprog::solveLP, lpSolve::lp

**Examples**

```
## Examples from the book "Operations research - A Model-based Approach"
##-- production planning
cc <- c(5, 3.5, 4.5)
Ain <- matrix(c(3, 5, 4,
               6, 1, 3), 2, 3, byrow=TRUE)
bin <- c(540, 480)
linprog(cc, A = Ain, b = bin, maximize = TRUE)
# $x      20    0 120
# $fval   640

##-- diet problem
cc <- c(1.59, 2.19, 2.99)
```

```

Ain <- matrix(c(-250, -380, -257,
               250,  380,  257,
               13,  31,  28), 3, 3, byrow = TRUE)
bin <- c(-1800, 2200, 100)
linprog(cc, A = Ain, b = bin)

#-- employee scheduling
cc <- c(1, 1, 1, 1, 1, 1)
A <- (-1)*matrix(c(1, 0, 0, 0, 0, 1,
                  1, 1, 0, 0, 0, 0,
                  0, 1, 1, 0, 0, 0,
                  0, 0, 1, 1, 0, 0,
                  0, 0, 0, 1, 1, 0,
                  0, 0, 0, 0, 1, 1), 6, 6, byrow = TRUE)
b <- -c(17, 9, 19, 12, 5, 8)
linprog(cc, A, b)

#-- inventory models
cc <- c(1, 1.1, 1.2, 1.25, 0.05, 0.15, 0.15)
Aeq <- matrix(c(1, 0, 0, 0, -1, 0, 0,
                0, 1, 0, 0, 1, -1, 0,
                0, 0, 1, 0, 0, 1, -1,
                0, 0, 0, 1, 0, 0, 1), 4, 7, byrow = TRUE)
beq <- c(60, 70, 130, 150)
ub <- c(120, 140, 150, 140, Inf, Inf, Inf)
linprog(cc, Aeq = Aeq, beq = beq, ub = ub)

#-- allocation problem
cc <- c(1, 1, 1, 1, 1)
A <- matrix(c(-5,  0,  0,  0,  0,
              0, -4.5,  0,  0,  0,
              0,  0, -5.5,  0,  0,
              0,  0,  0, -3.5,  0,
              0,  0,  0,  0, -5.5,
              5,  0,  0,  0,  0,
              0,  4.5,  0,  0,  0,
              0,  0,  5.5,  0,  0,
              0,  0,  0,  3.5,  0,
              0,  0,  0,  0,  5.5,
              -5, -4.5, -5.5, -3.5, -5.5,
              10, 10.0, 10.0, 10.0, 10.0,
              0.2, 0.2, 0.2, -1.0, 0.2), 13, 5, byrow = TRUE)
b <- c(-50, -55, -60, -50, -50, rep(100, 5), -5*64, 700, 0)
# linprog(cc, A = A, b = b)
lb <- b[1:5] / diag(A[1:5, ])
ub <- b[6:10] / diag(A[6:10, ])
A1 <- A[11:13, ]
b1 <- b[11:13]
linprog(cc, A1, b1, lb = lb, ub = ub)

#-- transportation problem
cc <- c(1, 7, 4, 2, 3, 5)
Aeq <- matrix(c(1, 1, 1, 0, 0, 0,

```

```
0, 0, 0, 1, 1, 1,  
1, 0, 0, 1, 0, 0,  
0, 1, 0, 0, 1, 0,  
0, 0, 1, 0, 0, 1), 5, 6, byrow = TRUE)  
beq <- c(30, 20, 15, 25, 10)  
linprog(cc, Aeq = Aeq, beq = beq)
```

---

linspace

*Linearly Spaced Sequences*

---

### Description

Generate linearly spaced sequences.

### Usage

```
linspace(x1, x2, n = 100)
```

### Arguments

x1	numeric scalar specifying starting point
x2	numeric scalar specifying ending point
n	numeric scalar specifying number of points to be generated

### Details

These functions will generate  $n$  linearly spaced points between  $x1$  and  $x2$ .

If  $n < 2$ , the result will be the ending point  $x2$ .

### Value

vector containing  $n$  points between  $x1$  and  $x2$  inclusive.

### See Also

[logspace](#), [seq](#)

### Examples

```
linspace(1, 10, 9)
```

---

logspace	<i>Log-linearly Spaced Sequences</i>
----------	--------------------------------------

---

**Description**

Generate log-linearly spaced sequences.

**Usage**

```
logspace(x1, x2, n = 50)
logseq(x1, x2, n = 100)
```

**Arguments**

x1	numeric scalar specifying starting point
x2	numeric scalar specifying ending point
n	numeric scalar specifying number of points to be generated

**Details**

These functions will generate logarithmically resp. exponentially spaced points between x1 and x2 resp.  $10^{x1}$  and  $10^{x2}$ .

If  $n < 2$ , the result will be the ending point x2. For `logspace()`, if  $x2 = \pi$ , the endpoint will be  $\pi$  and not  $10^{\pi}$ !

**Value**

vector containing n points between x1 and x2 inclusive.

**See Also**

[logspace](#), [seq](#)

**Examples**

```
logspace(1, pi, 36)
logseq(0.05, 1, 20)
```

---

`lsqlin`*Linear Least-Squares Fitting*

---

**Description**

Solves linearly constrained linear least-squares problems.

**Usage**

```
lsqlin(A, b, C, d, tol = 1e-13)
```

**Arguments**

A	nxm-matrix defining the least-squares problem.
b	vector or column matrix with n rows; when it has more than one column it describes several least-squares problems.
C	pxm-matrix for the constraint system.
d	vector or px1-matrix, right hand side for the constraints.
tol	tolerance to be passed to pinv.

**Details**

`lsqlin(A, b, C, d)` minimizes  $\|A*x - b\|$  (i.e., in the least-squares sense) subject to  $C*x = d$ .

**Value**

Returns a least-squares solution as column vector, or a matrix of solutions in the columns if b is a matrix with several columns.

**Note**

The Matlab function `lsqlin` solves a more general problem, allowing additional linear inequalities and bound constraints. In `pracma` this task is solved applying function `lsqlincon`.

**Author(s)**

HwB email: <hwborchers@googlemail.com>

**References**

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Society for Industrial and Applied Mathematics, Philadelphia.

**See Also**

[nullspace](#), [pinv](#), [lsqlincon](#)



**Examples**

```

A <- matrix(c(
  0.8147, 0.1576, 0.6557,
  0.9058, 0.9706, 0.0357,
  0.1270, 0.9572, 0.8491,
  0.9134, 0.4854, 0.9340,
  0.6324, 0.8003, 0.6787,
  0.0975, 0.1419, 0.7577,
  0.2785, 0.4218, 0.7431,
  0.5469, 0.9157, 0.3922,
  0.9575, 0.7922, 0.6555,
  0.9649, 0.9595, 0.1712), 10, 3, byrow = TRUE)
b <- matrix(c(
  0.7060, 0.4387,
  0.0318, 0.3816,
  0.2769, 0.7655,
  0.0462, 0.7952,
  0.0971, 0.1869,
  0.8235, 0.4898,
  0.6948, 0.4456,
  0.3171, 0.6463,
  0.9502, 0.7094,
  0.0344, 0.7547), 10, 2, byrow = TRUE)
C <- matrix(c(
  1.0000, 1.0000, 1.0000,
  1.0000, -1.0000, 0.5000), 2, 3, byrow = TRUE)
d <- as.matrix(c(1, 0.5))

# With a full rank constraint system
(L <- lsqlin(A, b, C, d))
# 0.10326838 0.3740381
# 0.03442279 0.1246794
# 0.86230882 0.5012825
C %*% L
# 1.0 1.0
# 0.5 0.5

## Not run:
# With a rank deficient constraint system
C <- str2num('[1 1 1;1 1 1]')
d <- str2num('[1;1]')
(L <- lsqlin(A, b[, 1], C, d))
# 0.2583340
# -0.1464215
# 0.8880875
C %*% L      # 1 1 as column vector

# Where both A and C are rank deficient
A2 <- repmat(A[, 1:2], 1, 2)
C <- ones(2, 4) # d as above
(L <- lsqlin(A2, b[, 2], C, d))
# 0.2244121

```

```
# 0.2755879
# 0.2244121
# 0.2755879
C %% L      # 1 1 as column vector
## End(Not run)
```

---

lsqincon

*Linear Least-Squares Fitting with linear constraints*


---

### Description

Solves linearly constrained linear least-squares problems.

### Usage

```
lsqincon(C, d, A = NULL, b = NULL,
         Aeq = NULL, beq = NULL, lb = NULL, ub = NULL)
```

### Arguments

C	mxn-matrix defining the least-squares problem.
d	vector or a one column matrix with m rows
A	pxn-matrix for the linear inequality constraints.
b	vector or px1-matrix, right hand side for the constraints.
Aeq	qxn-matrix for the linear equality constraints.
beq	vector or qx1-matrix, right hand side for the constraints.
lb	lower bounds, a scalar will be extended to length n.
ub	upper bounds, a scalar will be extended to length n.

### Details

lsqincon(C, d, A, b, Aeq, beq, lb, ub) minimizes  $\|C*x - d\|$  (i.e., in the least-squares sense) subject to the following constraints:  $A*x \leq b$ ,  $Aeq*x = beq$ , and  $lb \leq x \leq ub$ .

It applies the quadratic solver in quadprog with an active-set method for solving quadratic programming problems.

If some constraints are NULL (the default), they will not be taken into account. In case no constraints are given at all, it simply uses `qr.solve`.

### Value

Returns the least-squares solution as a vector.

### Note

Function `lsqin` in `pracma` solves this for equality constraints only, by computing a base for the nullspace of `Aeq`. But for linear inequality constraints there is no simple linear algebra ‘trick’, thus a real optimization solver is needed.

**Author(s)**

HwB email: <hwborchers@googlemail.com>

**References**

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Society for Industrial and Applied Mathematics, Philadelphia.

**See Also**

[lsqlin](#), `quadprog::solve.QP`

**Examples**

```
## MATLABs lsqlin example
C <- matrix(c(
  0.9501,  0.7620,  0.6153,  0.4057,
  0.2311,  0.4564,  0.7919,  0.9354,
  0.6068,  0.0185,  0.9218,  0.9169,
  0.4859,  0.8214,  0.7382,  0.4102,
  0.8912,  0.4447,  0.1762,  0.8936), 5, 4, byrow=TRUE)
d <- c(0.0578, 0.3528, 0.8131, 0.0098, 0.1388)
A <- matrix(c(
  0.2027,  0.2721,  0.7467,  0.4659,
  0.1987,  0.1988,  0.4450,  0.4186,
  0.6037,  0.0152,  0.9318,  0.8462), 3, 4, byrow=TRUE)
b <- c(0.5251, 0.2026, 0.6721)
Aeq <- matrix(c(3, 5, 7, 9), 1)
beq <- 4
lb <- rep(-0.1, 4) # lower and upper bounds
ub <- rep( 2.0, 4)

x <- lsqlincon(C, d, A, b, Aeq, beq, lb, ub)
# -0.10000000 -0.10000000 0.1599088 0.4089598
# check A %*% x - b >= 0
# check Aeq %*% x - beq == 0
# check sum((C %*% x - d)^2) # 0.1695104
```

---

 lsqnonlin

*Nonlinear Least-Squares Fitting*


---

**Description**

`lsqnonlin` solves nonlinear least-squares problems, including nonlinear data-fitting problems, through the Levenberg-Marquardt approach.

`lsqnonneg` solve nonnegative least-squares constraints problem.

**Usage**

```
lsqnonlin(fun, x0, options = list(), ...)
lsqnonneg(C, d)

lsqsep(flist, p0, xdata, ydata, const = TRUE)
lsqcurvefit(fun, p0, xdata, ydata)
```

**Arguments**

fun	User-defined, vector-valued function.
x0	starting point.
...	additional parameters passed to the function.
options	list of options, for details see below.
C, d	matrix and vector such that $Cx - d$ will be minimized with $x \geq 0$ .
flist	list of (nonlinear) functions, depending on one extra parameter.
p0	starting parameters.
xdata, ydata	data points to be fitted.
const	logical; shall a constant term be included.

**Details**

lsqnonlin computes the sum-of-squares of the vector-valued function fun, that is if  $f(x) = (f_1(x), \dots, f_n(x))$  then

$$\min \|f(x)\|_2^2 = \min (f_1(x)^2 + \dots + f_n(x)^2)$$

will be minimized.

$x = \text{lsqnonlin}(\text{fun}, x0)$  starts at point  $x0$  and finds a minimum of the sum of squares of the functions described in fun. fun shall return a vector of values and not the sum of squares of the values. (The algorithm implicitly sums and squares fun(x).)

options is a list with the following components and defaults:

- tau: used as starting value for Marquardt parameter.
- tol<sub>x</sub>: stopping parameter for step length.
- tol<sub>g</sub>: stopping parameter for gradient.
- maxeval the maximum number of function evaluations.

Typical values for tau are from  $1e-6 \dots 1e-3 \dots 1$  with small values for good starting points and larger values for not so good or known bad starting points.

lsqnonneg solves the linear least-squares problem  $Cx - d$ ,  $x$  nonnegative, treating it through an active-set approach..

lsqsep solves the separable least-squares fitting problem

$$y = a_0 + a_1 f_1(b_1, x) + \dots + a_n f_n(b_n, x)$$

where  $f_i$  are nonlinear functions each depending on a single extra parameter  $b_i$ , and  $a_i$  are additional linear parameters that can be separated out to solve a nonlinear problem in the  $b_i$  alone.

lsqcurvefit is simply an application of lsqnonlin to fitting data points.  $\text{fun}(p, x)$  must be a function of two groups of variables such that  $p$  will be varied to minimize the least squares sum, see the example below.

### Value

lsqnonlin returns a list with the following elements:

- $x$ : the point with least sum of squares value.
- $ssq$ : the sum of squares.
- $ng$ : norm of last gradient.
- $nh$ : norm of last step used.
- $\mu$ : damping parameter of Levenberg-Marquardt.
- $neval$ : number of function evaluations.
- $errno$ : error number, corresponds to error message.
- $errmsg$ : error message, i.e. reason for stopping.

lsqnonneg returns a list of  $x$  the non-negative solution, and  $\text{resid.norm}$  the norm of the residual.

lsqsep will return the coefficients separately,  $a_0$  for the constant term (being 0 if  $\text{const}=\text{FALSE}$ ) and the vectors  $a$  and  $b$  for the linear and nonlinear terms, respectively.

### Note

The refined approach, Fletcher's version of the Levenberg-Marquardt algorithm, may be added at a later time; see the references.

### References

Madsen, K., and H. B.Nielsen (2010). Introduction to Optimization and Data Fitting. Technical University of Denmark, Institute of Computer Science and Mathematical Modelling.

Lawson, C.L., and R.J. Hanson (1974). Solving Least-Squares Problems. Prentice-Hall, Chapter 23, p. 161.

Fletcher, R., (1971). A Modified Marquardt Subroutine for Nonlinear Least Squares. Report AERE-R 6799, Harwell.

### See Also

[nlm](#), [nls](#)

### Examples

```
## Rosenber function as least-squares problem
x0 <- c(0, 0)
fun <- function(x) c(10*(x[2]-x[1]^2), 1-x[1])
lsqnonlin(fun, x0)

## Example from R-help
y <- c(5.5199668, 1.5234525, 3.3557000, 6.7211704, 7.4237955, 1.9703127,
```

```

    4.3939336, -1.4380091, 3.2650180, 3.5760906, 0.2947972, 1.0569417)
x <- c(1, 0, 0, 4, 3, 5, 12, 10, 12, 100, 100, 100)
# Define target function as difference
f <- function(b)
  b[1] * (exp((b[2] - x)/b[3]) * (1/b[3]))/(1 + exp((b[2] - x)/b[3]))^2 - y
x0 <- c(21.16322, 8.83669, 2.957765)
lsqnonlin(f, x0)      # ssq 50.50144 at c(36.133144, 2.572373, 1.079811)

# nls() will break down
# nls(Y ~ a*(exp((b-X)/c)*(1/c))/(1 + exp((b-X)/c))^2,
#   start=list(a=21.16322, b=8.83669, c=2.957765), algorithm = "plinear")
# Error: step factor 0.000488281 reduced below 'minFactor' of 0.000976563

## Example: Hougon function
x1 <- c(470, 285, 470, 470, 470, 100, 100, 470, 100, 100, 100, 285, 285)
x2 <- c(300, 80, 300, 80, 80, 190, 80, 190, 300, 300, 80, 300, 190)
x3 <- c(10, 10, 120, 120, 10, 10, 65, 65, 54, 120, 120, 10, 120)
rate <- c(8.55, 3.79, 4.82, 0.02, 2.75, 14.39, 2.54,
  4.35, 13.00, 8.50, 0.05, 11.32, 3.13)
fun <- function(b)
  (b[1]*x2 - x3/b[5])/(1 + b[2]*x1 + b[3]*x2 + b[4]*x3) - rate
lsqnonlin(fun, rep(1, 5))
# $x [1.25258502 0.06277577 0.04004772 0.11241472 1.19137819]
# $ssq 0.298901

## Example for lsqnonneg()
C1 <- matrix( c(0.1210, 0.2319, 0.4398, 0.9342, 0.1370,
  0.4508, 0.2393, 0.3400, 0.2644, 0.8188,
  0.7159, 0.0498, 0.3142, 0.1603, 0.4302,
  0.8928, 0.0784, 0.3651, 0.8729, 0.8903,
  0.2731, 0.6408, 0.3932, 0.2379, 0.7349,
  0.2548, 0.1909, 0.5915, 0.6458, 0.6873,
  0.8656, 0.8439, 0.1197, 0.9669, 0.3461,
  0.2324, 0.1739, 0.0381, 0.6649, 0.1660,
  0.8049, 0.1708, 0.4586, 0.8704, 0.1556,
  0.9084, 0.9943, 0.8699, 0.0099, 0.1911), ncol = 5, byrow = TRUE)
C2 <- C1 - 0.5
d <- c(0.4225, 0.8560, 0.4902, 0.8159, 0.4608,
  0.4574, 0.4507, 0.4122, 0.9016, 0.0056)
( sol <- lsqnonneg(C1, d) )      #-> resid.norm 0.3694372
( sol <- lsqnonneg(C2, d) )      #-> $resid.norm 2.863979

## Example for lsqcurvefit()
# Lanczos1 data (artificial data)
# f(x) = 0.0951*exp(-x) + 0.8607*exp(-3*x) + 1.5576*exp(-5*x)
x <- linspace(0, 1.15, 24)
y <- c(2.51340000, 2.04433337, 1.66840444, 1.36641802, 1.12323249, 0.92688972,
  0.76793386, 0.63887755, 0.53378353, 0.44793636, 0.37758479, 0.31973932,
  0.27201308, 0.23249655, 0.19965895, 0.17227041, 0.14934057, 0.13007002,
  0.11381193, 0.10004156, 0.08833209, 0.07833544, 0.06976694, 0.06239313)

p0 <- c(1.2, 0.3, 5.6, 5.5, 6.5, 7.6)
fp <- function(p, x) p[1]*exp(-p[2]*x) + p[3]*exp(-p[4]*x) + p[5]*exp(-p[6]*x)

```

```

lsqcurvefit(fp, p0, x, y)

## Example for lsqsep()
f <- function(x) 0.5 + x^-0.5 + exp(-0.5*x)
set.seed(8237); n <- 15
x <- sort(0.5 + 9*runif(n))
y <- f(x)                #y <- f(x) + 0.01*rnorm(n)

m <- 2
f1 <- function(b, x) x^b
f2 <- function(b, x) exp(b*x)
flist <- list(f1, f2)
start <- c(-0.25, -0.75)

sol <- lsqsep(flist, start, x, y, const = TRUE)
a0 <- sol$a0; a <- sol$a; b <- sol$b
fsol <- function(x) a0 + a[1]*f1(b[1], x) + a[2]*f2(b[2], x)

## Not run:
ezplot(f, 0.5, 9.5, col = "gray")
points(x, y, col = "blue")
xs <- linspace(0.5, 9.5, 51)
ys <- fsol(xs)
lines(xs, ys, col = "red")

## End(Not run)

```

---

lu

*LU Matrix Factorization*


---

### Description

LU decomposition of a positive definite matrix as Gaussian factorization.

### Usage

```

lu(A, scheme = c("kji", "jki", "ijk"))
lu_cROUT(A)

lufact(A)
lusys(A, b)

```

### Arguments

A	square positive definite numeric matrix (will not be checked).
scheme	order of row and column operations.
b	right hand side of a linear system of equations.

## Details

For a given matrix  $A$ , the LU decomposition exists and is unique iff its principal submatrices of order  $i=1, \dots, n-1$  are nonsingular. The procedure here is a simple Gauss elimination with or without pivoting.

The scheme abbreviations refer to the order in which the cycles of row- and column-oriented operations are processed. The “ijk” scheme is one of the two compact forms, here the Doolite factorization (the Crout factorization would be similar).

`lu_crout` implements the Crout algorithm. For the Doolite algorithm, the  $L$  matrix has ones on its diagonal, for the Crout algorithm, the diagonal of the  $U$  matrix only has ones.

`lufact` applies partial pivoting (along the rows). `lusys` uses LU factorization to solve the linear system  $A*x=b$ .

These function are not meant to process huge matrices or linear systems of equations. Without pivoting they may also be harmed by considerable inaccuracies.

## Value

`lu` and `lu_crout` return a list with components  $L$  and  $U$ , the lower and upper triangular matrices such that  $A=L\%*U$ .

`lufact` returns a list with  $L$  and  $U$  combined into one matrix  $LU$ , the rows used in partial pivoting, and `det` representing the determinant of  $A$ . See the examples how to extract matrices  $L$  and  $U$  from  $LU$ .

`lusys` returns the solution of the system as a column vector.

## Note

To get the Crout decomposition of a matrix  $A$  do `Z <- lu(t(A)); L <- t(Z$U); U <- t(Z$L)`.

## References

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second edition, Springer-Verlag, Berlin Heidelberg.

J.H. Mathews and K.D. Fink (2003). Numerical Methods Using MATLAB. Fourth Edition, Pearson (Prentice-Hall), updated 2006.

## See Also

[qr](#)

## Examples

```
A <- magic(5)
D <- lu(A, scheme = "ijk")      # Doolittle scheme
D$L %% D$U
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  17  24   1   8  15
## [2,]  23   5   7  14  16
## [3,]   4   6  13  20  22
```



```
## [4,] 10 12 19 21 3
## [5,] 11 18 25 2 9

H4 <- hilb(4)
lufact(H4)$det
## [1] 0.0000001653439

x0 <- c(1.0, 4/3, 5/3, 2.0)
b <- H4 %*% x0
lusys(H4, b)
##           [,1]
## [1,] 1.000000
## [2,] 1.333333
## [3,] 1.666667
## [4,] 2.000000
```

---

magic

*Magic Square*

---

### Description

Create a magic square.

### Usage

```
magic(n)
```

### Arguments

`n` numeric scalar specifying dimensions for the result; `n` must be a scalar greater than or equal to 3.

### Details

A magic square is a square matrix where all row and column sums and also the diagonal sums all have the same value.

This value or the characteristic sum for a magic square of order  $n$  is  $sum(1 : n^2)/n$ .

### Value

Returns an  $n$ -by- $n$  matrix constructed from the integers 1 through  $N^2$  with equal row and column sums.

### Note

A magic square, scaled by its magic sum, is doubly stochastic.

**Author(s)**

P. Roebuck <roebuck@mdanderson.org> for the first R version in the package 'matlab'. The version here is more R-like.

**Examples**

```
magic(3)
```

---

matlab	<i>Matlab Compatibility</i>
--------	-----------------------------

---

**Description**

Matlab compatibility.

**Usage**

```
matlab()
```

**Details**

Lists all the functions and function names that emulate Matlab functions.

**Value**

Invisible NULL value.

---

meshgrid	<i>Generate a Mesh Grid</i>
----------	-----------------------------

---

**Description**

Generate two matrices for use in three-dimensional plots.

**Usage**

```
meshgrid(x, y = x)
```

**Arguments**

x	numerical vector, represents points along the x-axis.
y	numerical vector, represents points along the y-axis.

**Details**

The rows of the output array X are copies of the vector x; columns of the output array Y are copies of the vector y.

**Value**

Returns two matrices as a list with X and Y components.

**Note**

The three-dimensional variant `meshgrid(x, y, z)` is not yet implemented.

**See Also**

[outer](#)

**Examples**

```
meshgrid(1:5)$X
meshgrid(c(1, 2, 3), c(11, 12))
```

---

mexpfit

*Multi-exponential Fitting*

---

**Description**

Multi-exponential fitting means fitting of data points by a sum of (decaying) exponential functions, with or without a constant term.

**Usage**

```
mexpfit(x, y, p0, w = NULL, const = TRUE, options = list())
```

**Arguments**

x, y	x-, y-coordinates of data points to be fitted.
p0	starting values for the exponentials alone; can be positive or negative, but not zero.
w	weight vector; not used in this version.
const	logical; shall an absolute term be included.
options	list of options for <code>lsqnonlin</code> , see there.

**Details**

The multi-exponential fitting problem is solved here with with a separable nonlinear least-squares approach. If the following function is to be fitted,

$$y = a_0 + a_1 e^{b_1 x} + \dots + a_n e^{b_n x}$$

it will be looked at as a nonlinear optimization problem of the coefficients  $b_i$  alone. Given the  $b_i$ , coefficients  $a_i$  are uniquely determined as solution of an (overdetermined) system of linear equations.

This approach reduces the dimension of the search space by half and improves numerical stability and accuracy. As a convex problem, the solution is unique and global.

To solve the nonlinear part, the function `lsqnonlin` that uses the Levenberg-Marquard algorithm will be applied.

### Value

`mexpfit` returns a list with the following elements:

- `a0`: the absolute term, 0 if `const` is false.
- `a`: linear coefficients.
- `b`: coefficient in the exponential functions.
- `ssq`: the sum of squares for the final fitting.
- `iter`: number of iterations resp. function calls.
- `errmess`: an error or info message.

### Note

As the Jacobian for this expression is known, a more specialized approach would be possible, without using `lsqnonlin`; see the `immoptibox` of H. B. Nielsen, Techn. University of Denmark.

### Author(s)

HwB email: <hwborchers@googlemail.com>

### References

Madsen, K., and H. B. Nielsen (2010). Introduction to Optimization and Data Fitting. Technical University of Denmark, Institute of Computer Science and Mathematical Modelling.

Nielsen, H. B. (2000). Separable Nonlinear Least Squares. IMM, DTU, Report IMM-REP-2000-01.

### See Also

[lsqsep](#), [lsqnonlin](#)

### Examples

```
# Lanczos1 data (artificial data)
# f(x) = 0.0951*exp(-x) + 0.8607*exp(-3*x) + 1.5576*exp(-5*x)
x <- linspace(0, 1.15, 24)
y <- c(2.51340000, 2.04433337, 1.66840444, 1.36641802, 1.12323249, 0.92688972,
      0.76793386, 0.63887755, 0.53378353, 0.44793636, 0.37758479, 0.31973932,
      0.27201308, 0.23249655, 0.19965895, 0.17227041, 0.14934057, 0.13007002,
      0.11381193, 0.10004156, 0.08833209, 0.07833544, 0.06976694, 0.06239313)
p0 <- c(-0.3, -5.5, -7.6)
mexpfit(x, y, p0, const = FALSE)
## $a0
## [1] 0
```

```
## $a
## [1] 0.09510431 0.86071171 1.55758398
## $b
## [1] -1.000022 -3.000028 -5.000009
## $ssq
## [1] 1.936163e-16
## $iter
## [1] 26
## $errmess
## [1] "Stopped by small gradient."
```

---

mldivide

*Matlab backslash operator*


---

### Description

Emulate the Matlab backslash operator “\” through QR decomposition.

### Usage

```
mldivide(A, B, pinv = TRUE)
mrdivide(A, B, pinv = TRUE)
```

### Arguments

A, B	Numerical or complex matrices; A and B must have the same number of rows (for mldivide) or the same number of columns (for mrdivide)
pinv	logical; shall SVD decomposition be used; default true.

### Details

mldivide performs matrix left division (and mrdivide matrix right division). If A is scalar it performs element-wise division.

If A is square, mldivide is roughly the same as `inv(A) %% B` except it is computed in a different way — using QR decomposition.

If `pinv = TRUE`, the default, the SVD will be used as `pinv(t(A)%%A)%%t(A)%%B` to generate results similar to Matlab. Otherwise, `qr.solve` will be used.

If A is not square, `x <- mldivide(A, b)` returns a least-squares solution that minimizes the length of the vector `A %% x - b` (which is equivalent to `norm(A %% x - b, "F")`).

### Value

If A is an n-by-p matrix and B n-by-q, then the result of `mldivide(A, B)` is a p-by-q matrix (mldivide).

### Note

`mldivide(A, B)` corresponds to `A\B` in Matlab notation.

**Examples**

```
# Solve a system of linear equations
A <- matrix(c(8,1,6, 3,5,7, 4,9,2), nrow = 3, ncol = 3, byrow = TRUE)
b <- c(1, 1, 1)
mldivide(A, b) # 0.06666667 0.06666667 0.06666667

A <- rbind(1:3, 4:6)
mldivide(A, c(1,1)) # -0.5 0 0.5 ,i.e. Matlab/Octave result
mldivide(A, c(1,1), pinv = FALSE) # -1 1 0 R qr.solve result
```

---

 mod, rem

*Integer Division*


---

**Description**

Integer division functions and remainders

**Usage**

```
mod(n, m)
rem(n, m)

idivide(n, m, rounding = c("fix", "floor", "ceil", "round"))
```

**Arguments**

`n` numeric vector (preferably of integers)  
`m` must be a scalar integer (positive, zero, or negative)  
`rounding` rounding mode.

**Details**

`mod(n, m)` is the modulo operator and returns  $n \bmod m$ . `mod(n, 0)` is `n`, and the result always has the same sign as `m`.

`rem(n, m)` is the same modulo operator and returns  $n \bmod m$ . `rem(n, 0)` is `NaN`, and the result always has the same sign as `n`.

`idivide(n, m)` is integer division, with the same effect as `n %/% m` or using an optional rounding mode.

**Value**

a numeric (integer) value or vector/matrix.

**Note**

The following relation is fulfilled (for  $m \neq 0$ ):

$$\text{mod}(n, m) = n - m * \text{floor}(n/m)$$

**See Also**

Binary R operators `%%` and `%%`.

**Examples**

```
mod(c(-5:5), 5)
rem(c(-5:5), 5)

idivide(c(-2, 2), 3, "fix")    # 0 0
idivide(c(-2, 2), 3, "floor") # -1 0
idivide(c(-2, 2), 3, "ceil")  # 0 1
idivide(c(-2, 2), 3, "round") # -1 1
```

---

Mode

*Mode function (Matlab style)*

---

**Description**

Most frequent value in vector or matrix

**Usage**

Mode(x)

**Arguments**

x                      Real or complex vector or of factor levels.

**Details**

Computes the ‘sample mode’, i.e. the most frequently occurring value in x.

Among values occurring equally frequently, Mode() chooses the smallest one (for a numeric vector), one with a smallest absolute value (for complex ones) or the first occurring value (for factor levels).

A matrix will be changed to a vector.

**Value**

One element from x and of the same type. The number of occurrences will not be returned.

**Note**

In Matlab/Octave an array dimension can be selected along which to find the mode value; this has not been realized here.

Shadows the R function mode that returns essentially the type of an object.

**See Also**

[median](#)

**Examples**

```
x <- round(rnorm(1000), 2)
Mode(x)
```

---

moler

*Moler Matrix*

---

**Description**

Generate the Moler matrix of size  $n \times n$ . The Moler matrix is for testing eigenvalue computations.

**Usage**

```
moler(n)
```

**Arguments**

n                    integer

**Details**

The Moler matrix for testing eigenvalue computations is a symmetric matrix with exactly one small eigenvalue.

**Value**

matrix of size  $n \times n$

**See Also**

[wilkinson](#)

**Examples**

```
(a <- moler(10))
min(eig(a))
```



---

 movavg *Moving Average Filters*


---

**Description**

Different types of moving average of a time series.

**Usage**

```
movavg(x, n, type=c("s", "t", "w", "m", "e", "r"))
```

**Arguments**

x	time series as numeric vector.
n	backward window length.
type	one of 's', 't', 'w', 'm', 'e', or 'r'; default is 's'.

**Details**

Types of available moving averages are:

- s for “simple”, it computes the simple moving average. n indicates the number of previous data points used with the current data point when calculating the moving average.
- t for “triangular”, it computes the triangular moving average by calculating the first simple moving average with window width of  $\text{ceil}(n+1)/2$ ; then it calculates a second simple moving average on the first moving average with the same window size.
- w for “weighted”, it calculates the weighted moving average by supplying weights for each element in the moving window. Here the reduction of weights follows a linear trend.
- m for “modified”, it calculates the modified moving average. The first modified moving average is calculated like a simple moving average. Subsequent values are calculated by adding the new value and subtracting the last average from the resulting sum.
- e for “exponential”, it computes the exponentially weighted moving average. The exponential moving average is a weighted moving average that reduces influences by applying more weight to recent data points ( ) reduction factor  $2/(n+1)$ ; or
- r for “running”, this is an exponential moving average with a reduction factor of  $1/n$  [same as the modified average?].

**Value**

Vector the same length as time series x.

**References**

Matlab Techdoc

**See Also**

filter

**Examples**

```
## Not run:
abbshares <- scan(file="")
25.69 25.89 25.86 26.08 26.41 26.90 26.27 26.45 26.49 26.08 26.11 25.57 26.02
25.53 25.27 25.95 25.19 24.78 24.96 24.63 25.68 25.24 24.87 24.71 25.01 25.06
25.62 25.95 26.08 26.25 25.91 26.61 26.34 25.55 25.36 26.10 25.63 25.52 24.74
25.00 25.38 25.01 24.57 24.95 24.89 24.13 23.83 23.94 23.74 23.12 23.13 21.05
21.59 19.59 21.88 20.59 21.59 21.86 22.04 21.48 21.37 19.94 19.49 19.46 20.34
20.59 19.96 20.18 20.74 20.83 21.27 21.19 20.27 18.83 19.46 18.90 18.09 17.99
18.03 18.50 19.11 18.94 18.21 18.06 17.66 16.77 16.77 17.10 17.62 17.22 17.95
17.08 16.42 16.71 17.06 17.75 17.65 18.90 18.80 19.54 19.23 19.48 18.98 19.28
18.49 18.49 19.08 19.63 19.40 19.59 20.37 19.95 18.81 18.10 18.32 19.02 18.78
18.68 19.12 17.79 18.10 18.64 18.28 18.61 18.20 17.82 17.76 17.26 17.08 16.70
16.68 17.68 17.70 18.97 18.68 18.63 18.80 18.81 19.03 18.26 18.78 18.33 17.97
17.60 17.72 17.79 17.74 18.37 18.24 18.47 18.75 18.66 18.51 18.71 18.83 19.82
19.71 19.64 19.24 19.60 19.77 19.86 20.23 19.93 20.33 20.98 21.40 21.14 21.38
20.89 21.08 21.30 21.24 20.55 20.83 21.57 21.67 21.91 21.66 21.53 21.63 21.83
21.48 21.71 21.44 21.67 21.10 21.03 20.83 20.76 20.90 20.92 20.80 20.89 20.49
20.70 20.60 20.39 19.45 19.82 20.28 20.24 20.30 20.66 20.66 21.00 20.88 20.99
20.61 20.45 20.09 20.34 20.61 20.29 20.20 20.00 20.41 20.70 20.43 19.98 19.92
19.77 19.23 19.55 19.93 19.35 19.66 20.27 20.10 20.09 20.48 19.86 20.22 19.35
19.08 18.81 18.87 18.26 18.27 17.91 17.68 17.73 17.56 17.20 17.14 16.84 16.47
16.45 16.25 16.07

plot(abbshares, type = "l", col = 1, ylim = c(15, 30),
     main = "Types of moving averages", sub = "Mid 2011--Mid 2012",
     xlab = "Days", ylab = "ABB Shares Price (in USD)")
y <- movavg(abbshares, 50, "s"); lines(y, col = 2)
y <- movavg(abbshares, 50, "t"); lines(y, col = 3)
y <- movavg(abbshares, 50, "w"); lines(y, col = 4)
y <- movavg(abbshares, 50, "m"); lines(y, col = 5)
y <- movavg(abbshares, 50, "e"); lines(y, col = 6)
y <- movavg(abbshares, 50, "r"); lines(y, col = 7)
grid()
legend(120, 29, c("original data", "simple", "triangular", "weighted",
                 "modified", "exponential", "running"),
      col = 1:7, lty = 1, lwd = 1, box.col = "gray", bg = "white")

## End(Not run)
```

muller

*Muller's Method***Description**

Muller's root finding method, similar to the secant method, using a parabola through three points for approximating the curve.

**Usage**

```
muller(f, p0, p1, p2 = NULL, maxiter = 100, tol = 1e-10)
```

**Arguments**

f	function whose root is to be found; function needs to be defined on the complex plain.
p0, p1, p2	three starting points, should enclose the assumed root.
tol	relative tolerance, change in successive iterates.
maxiter	maximum number of iterations.

**Details**

Generalizes the secant method by using parabolic interpolation between three points. This technique can be used for any root-finding problem, but is particularly useful for approximating the roots of polynomials, and for finding zeros of analytic functions in the complex plane.

**Value**

List of root, fval, niter, and reltol.

**Note**

Muller's method is considered to be (a bit) more robust than Newton's.

**References**

Pseudo- and C code available from the 'Numerical Recipes'; pseudocode in the book 'Numerical Analysis' by Burden and Faires (2011).

**See Also**

[secant](#), [newtonRaphson](#), [newtonsys](#)

**Examples**

```
muller(function(x) x^10 - 0.5, 0, 1) # root: 0.9330329915368074

f <- function(x) x^4 - 3*x^3 + x^2 + x + 1
p0 <- 0.5; p1 <- -0.5; p2 <- 0.0
muller(f, p0, p1, p2)
## $root
## [1] -0.3390928-0.4466301i
## ...

## Roots of complex functions:
fz <- function(z) sin(z)^2 + sqrt(z) - log(z)
muller(fz, 1, 1i, 1+1i)
## $root
## [1] 0.2555197+0.8948303i
```

```
## $fval
## [1] -4.440892e-16+0i
## $niter
## [1] 8
## $reltol
## [1] 3.656219e-13
```

---

nchoosek

*Binomial Coefficients*


---

### Description

Compute the Binomial coefficients.

### Usage

```
nchoosek(n, k)
```

### Arguments

n, k                    integers with k between 0 and n

### Details

Alias for the corresponding R function choose.

### Value

integer, the Binomial coefficient  $\binom{n}{k}$ .

### Note

In Matlab/Octave, if n is a vector all combinations of k elements from vector n will be generated. Here, use the function combs instead.

### See Also

[choose](#)

### Examples

```
S <- sapply(0:6, function(k) nchoosek(6, k)) # 1 6 15 20 15 6 1

# Catalan numbers
catalan <- function(n) choose(2*n, n)/(n+1)
catalan(0:10)
# 1 1 2 5 14 42 132 429 1430 4862 16796

# Relations
n <- 10
sum((-1)^c(0:n) * sapply(0:n, function(k) nchoosek(n, k))) # 0
```

---

ndims	<i>Number of Dimensions</i>
-------	-----------------------------

---

**Description**

Number of matrix or array dimensions.

**Usage**

```
ndims(x)
```

**Arguments**

x                    a vector, matrix, array, or list

**Details**

Returns the number of dimensions as `length(x)`.

For an empty object its dimension is 0, for vectors it is 1 (deviating from MATLAB), for matrices it is 2, and for arrays it is the number of dimensions, as usual. Lists are considered to be (one-dimensional) vectors.

**Value**

the number of dimensions in a vector, matrix, or array x.

**Note**

The result will differ from Matlab when x is a vector.

**See Also**

[size](#)

**Examples**

```
ndims(c())           # 0
ndims(as.numeric(1:8)) # 1
ndims(list(a=1, b=2, c=3)) # 1
ndims(matrix(1:12, 3, 4)) # 2
ndims(array(1:8, c(2,2,2))) # 3
```

---

`nearest_spd`*Nearest Symmetric Positive-definite Matrix*

---

**Description**

Find nearest (in Frobenius norm) symmetric positive-definite matrix to A.

**Usage**

```
nearest_spd(A)
```

**Arguments**

A                    square numeric matrix.

**Details**

"The nearest symmetric positive semidefinite matrix in the Frobenius norm to an arbitrary real matrix A is shown to be  $(B + H)/2$ , where H is the symmetric polar factor of  $B=(A + A')/2$ ."  
N. J. Highham

**Value**

Returns a matrix of the same size.

**References**

Nicholas J. Higham (1988). Computing a nearest symmetric positive semidefinite matrix. *Linear Algebra and its Applications*. Vol. 103, pp.103-118.

**See Also**

[randortho](#), [procrustes](#)

**Examples**

```
A <- matrix(1:9, 3, 3)
B <- nearest_spd(A); B
#           [,1]      [,2]      [,3]
# [1,] 2.034900 3.202344 4.369788
# [2,] 3.202344 5.039562 6.876781
# [3,] 4.369788 6.876781 9.383774
norm(B - A, type = 'F')
# [1] 3.758517
```

---

`nelder_mead`*Nelder-Mead Function Minimization Method*

---

## Description

An implementation of the Nelder-Mead algorithm for derivative-free optimization / function minimization.

## Usage

```
nelder_mead(fn, x0, ..., adapt = TRUE,
            tol = 1e-08, maxfeval = 5000,
            step = rep(1.0, length(x0)))
```

## Arguments

<code>fn</code>	nonlinear function to be minimized.
<code>x0</code>	starting point for the iteration.
<code>...</code>	additional arguments to be passed to the function.
<code>adapt</code>	logical; adapt to parameter dimension.
<code>tol</code>	terminating limit for the variance of function values; can be made <i>*very*</i> small, like <code>tol=1e-50</code> .
<code>maxfeval</code>	maximum number of function evaluations.
<code>step</code>	size and shape of initial simplex; relative magnitudes of its elements should reflect the units of the variables.

## Details

Also called a ‘simplex’ method for finding the local minimum of a function of several variables. The method is a pattern search that compares function values at the vertices of the simplex. The process generates a sequence of simplices with ever reducing sizes.

The simplex function minimisation procedure due to Nelder and Mead (1965), as implemented by O’Neill (1971), with subsequent comments by Chambers and Ertel 1974, Benyon 1976, and Hill 1978. For another elaborate implementation of Nelder-Mead in R based on Matlab code by Kelley see package ‘dfoptim’.

`nelder_mead` can be used up to 20 dimensions (then ‘tol’ and ‘maxfeval’ need to be increased). With `adapt=TRUE` it applies adaptive coefficients for the simplicial search, depending on the problem dimension – see Fuchang and Lixing (2012). This approach especially reduces the number of function calls.

**Value**

List with following components:

xmin	minimum solution found.
fmin	value of f at minimum.
count	number of iterations performed.
info	list with solver name and no. of restarts.

**Note**

Original FORTRAN77 version by R O'Neill; MATLAB version by John Burkardt under LGPL license. Re-implemented in R by Hans W. Borchers.

**References**

Nelder, J., and R. Mead (1965). A simplex method for function minimization. *Computer Journal*, Volume 7, pp. 308-313.

O'Neill, R. (1971). Algorithm AS 47: Function Minimization Using a Simplex Procedure. *Applied Statistics*, Volume 20(3), pp. 338-345.

J. C. Lagarias et al. (1998). Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal for Optimization*, Vol. 9, No. 1, pp 112-147.

Fuchang Gao and Lixing Han (2012). Implementing the Nelder-Mead simplex algorithm with adaptive parameters. *Computational Optimization and Applications*, Vol. 51, No. 1, pp. 259-277.

**See Also**

[hooke\\_jeeves](#)

**Examples**

```
## Classical tests as in the article by Nelder and Mead
# Rosenbrock's parabolic valley
rpv <- function(x) 100*(x[2] - x[1]^2)^2 + (1 - x[1])^2
x0 <- c(-2, 1)
nelder_mead(rpv, x0)                # 1 1

# Fletcher and Powell's helic valley
fphv <- function(x)
  100*(x[3] - 10*atan2(x[2], x[1])/(2*pi))^2 +
  (sqrt(x[1]^2 + x[2]^2) - 1)^2 + x[3]^2
x0 <- c(-1, 0, 0)
nelder_mead(fphv, x0)              # 1 0 0

# Powell's Singular Function (PSF)
psf <- function(x) (x[1] + 10*x[2])^2 + 5*(x[3] - x[4])^2 +
  (x[2] - 2*x[3])^4 + 10*(x[1] - x[4])^4
x0 <- c(3, -1, 0, 1)
# needs maximum number of function calls
nelder_mead(psf, x0, maxfeval=30000) # 0 0 0 0
```



```
## Not run:
# Can run Rosenbrock's function in 30 dimensions in one and a half minutes:
nelder_mead(fnRosenbrock, rep(0, 30), tol=1e-20, maxfeval=10^7)
# $xmin
# [1] 0.9999998 1.0000004 1.0000000 1.0000001 1.0000000 1.0000001
# [7] 1.0000002 1.0000001 0.9999997 0.9999999 0.9999997 1.0000000
# [13] 0.9999999 0.9999994 0.9999998 0.9999999 0.9999999 0.9999999
# [19] 0.9999999 1.0000001 0.9999998 1.0000000 1.0000003 0.9999999
# [25] 1.0000000 0.9999996 0.9999995 0.9999990 0.9999973 0.9999947
# $fmin
# [1] 5.617352e-10
# $fcount
# [1] 1426085
# elapsed time is 96.008000 seconds
## End(Not run)
```

---

neville

*Neville's Method*


---

### Description

Neville's's method of polynomial interpolation.

### Usage

```
neville(x, y, xs)
```

### Arguments

x, y	x-, y-coordinates of data points defining the polynomial.
xs	single point to be interpolated.

### Details

Straightforward implementation of Neville's method; not yet vectorized.

### Value

Interpolated value at xs of the polynomial defined by x, y.

### References

Each textbook on numerical analysis.

### See Also

[newtonInterp](#), [barylag](#)

**Examples**

```

p <- Poly(c(1, 2, 3))
fp <- function(x) polyval(p, x)

x <- 0:4; y <- fp(x)
xx <- linspace(0, 4, 51)
yy <- numeric(51)
for (i in 1:51) yy[i] <- neville(x, y, xx[i])

## Not run:
ezplot(fp, 0, 4)
points(xx, yy)
## End(Not run)

```

---

newmark

*Newmark Method*


---

**Description**

Newmark's is a method to solve higher-order differential equations without passing through the equivalent first-order system. It generalizes the so-called 'leap-frog' method. Here it is restricted to second-order equations.

**Usage**

```
newmark(f, t0, t1, y0, ..., N = 100, zeta = 0.25, theta = 0.5)
```

**Arguments**

f	function in the differential equation $y'' = f(x, y, y')$ ; defined as a function $R \times R^2 \rightarrow R$ .
t0, t1	start and end points of the interval.
y0	starting values as row or column vector; y0 needs to be a vector of length 2, the first component representing $y(t_0)$ , the second $dy/dt(t_0)$ .
N	number of steps.
zeta, theta	two non-negative real numbers.
...	Additional parameters to be passed to the function.

**Details**

Solves second order differential equations using the Newmark method on an equispaced grid of N steps.

Function f must return a vector, whose elements hold the evaluation of  $f(t, y)$ , of the same dimension as y0. Each row in the solution array Y corresponds to a time returned in t.

The method is 'implicit' unless zeta=theta=0, second order if theta=1/2 and first order accurate if theta!=1/2. theta>=1/2 ensures stability. The condition set theta=1/2; zeta=1/4 (the

defaults) is a popular approach that is unconditionally stable, but introduces oscillatory spurious solutions on long time intervals. (For these simulations it is preferable to use  $\theta > 1/2$  and  $\zeta > (\theta + 1/2)^{1/2}$ .)

No attempt is made to catch any errors in the root finding functions.

### Value

List with components `t` for grid (or 'time') points between `t0` and `t1`, and `y` an `n`-by-2 matrix with solution variables in columns, i.e. each row contains one time stamp.

### Note

This is for demonstration purposes only; for real problems or applications please use `ode23` or `rk4sys`.

### References

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

### See Also

[ode23](#), [cranknic](#)

### Examples

```
# Mathematical pendulum  $m l y'' + m g \sin(y) = 0$ 
pendel <- function(t, y) -sin(y[1])
sol <- newmark(pendel, 0, 4*pi, c(pi/4, 0))

## Not run:
plot(sol$t, sol$y[, 1], type="l", col="blue",
      xlab="Time", ylab="Elongation/Speed", main="Mathematical Pendulum")
lines(sol$t, sol$y[, 2], col="darkgreen")
grid()
## End(Not run)
```

---

newtonHorner

*Newton's Root Finding Method for Polynomials.*

---

### Description

Finding roots of univariate polynomials.

### Usage

```
newtonHorner(p, x0, maxiter = 50, tol = .Machine$double.eps^0.5)
```

**Arguments**

<code>p</code>	Numeric vector representing a polynomial.
<code>x0</code>	starting value for <code>newtonHorner()</code> .
<code>maxiter</code>	maximum number of iterations; default 100.
<code>tol</code>	absolute tolerance; default $\text{eps}^{(1/2)}$

**Details**

Similar to `newtonRahson`, except that the computation of the derivative is done through the Horner scheme in parallel with computing the value of the polynomial. This makes the algorithm significantly faster.

**Value**

Return a list with components `root`, `f.root`, the function value at the found root, `iter`, the number of iterations done, and `root`, and the estimated precision `estim.prec`

The estimated precision is given as the difference to the last solution before stop.

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[newtonRaphson](#)

**Examples**

```
## Example: x^3 - 6 x^2 + 11 x - 6 with roots 1, 2, 3
p <- c(1, -6, 11, -6)
x0 <- 0
while (length(p) > 1) {
  N <- newtonHorner(p, x0)
  if (!is.null(N$root)) {
    cat("x0 =", N$root, "\n")
    p <- N$deflate
  } else {
    break
  }
}
## Try: p <- Poly(c(1:20))
```

---

newtonInterp                      *Lagrange and Newtons Interpolation*

---

### Description

Lagrange's and Newton's method of polynomial interpolation.

### Usage

```
newtonInterp(x, y, xs = c())
```

```
lagrangeInterp(x, y, xs)
```

### Arguments

x, y	x-, y-coordinates of data points defining the polynomial.
xs	either empty, or a vector of points to be interpolated.

### Details

Straightforward implementation of Lagrange's Newton's method (vectorized in xs).

### Value

A vector of values at xs of the polynomial defined by x, y.

### References

Each textbook on numerical analysis.

### See Also

[neville](#), [barylag](#)

### Examples

```
p <- Poly(c(1, 2, 3))
fp <- function(x) polyval(p, x)

x <- 0:4; y <- fp(x)
xx <- linspace(0, 4, 51)
yy <- lagrangeInterp(x, y, xx)
yy <- newtonInterp(x, y, xx)
## Not run:
ezplot(fp, 0, 4)
points(xx, yy)
## End(Not run)
```

---

`newtonRaphson`*Rootfinding through Newton-Raphson or Secant.*

---

**Description**

Finding roots of univariate functions. (Newton never invented or used this method; it should be called more appropriately Simpson's method!)

**Usage**

```
newtonRaphson(fun, x0, dfun = NULL, maxiter = 500, tol = 1e-08, ...)  
newton(fun, x0, dfun = NULL, maxiter = 500, tol = 1e-08, ...)
```

**Arguments**

<code>fun</code>	Function or its name as a string.
<code>x0</code>	starting value for <code>newtonRaphson()</code> .
<code>dfun</code>	A function to compute the derivative of <code>f</code> . If <code>NULL</code> , a numeric derivative will be computed.
<code>maxiter</code>	maximum number of iterations; default 100.
<code>tol</code>	absolute tolerance; default $\text{eps}^{(1/2)}$
<code>...</code>	Additional arguments to be passed to <code>f</code> .

**Details**

Well known root finding algorithms for real, univariate, continuous functions.

**Value**

Return a list with components `root`, `f.root`, the function value at the found root, `iter`, the number of iterations done, and `estim.prec`, and the estimated precision `estim.prec`

The estimated precision is given as the difference to the last solution before stop; this may be misleading.

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[newtonHorner](#)

**Examples**

```
# Legendre polynomial of degree 5
lp5 <- c(63, 0, -70, 0, 15, 0)/8
f <- function(x) polyval(lp5, x)
newton(f, 1.0)      # 0.9061798459 correct to 10 decimals in 5 iterations
```

newtonsys

*Newton Method for Nonlinear Systems***Description**

Newton's method applied to multivariate nonlinear functions.

**Usage**

```
newtonsys(Ffun, x0, Jfun = NULL, ...,
          maxiter = 100, tol = .Machine$double.eps^(1/2))
```

**Arguments**

Ffun	m functions of n variables.
Jfun	Function returning a square n-by-n matrix (of partial derivatives) or NULL, the default.
x0	Numeric vector of length n.
maxiter	Maximum number of iterations.
tol	Tolerance, relative accuracy.
...	Additional parameters to be passed to f.

**Details**

Solves the system of equations applying Newton's method with the univariate derivative replaced by the Jacobian.

**Value**

List with components: zero the root found so far, fnorm the square root of sum of squares of the values of f, and iter the number of iterations needed.

**Note**

TODO: better error checking, e.g. when the Jacobian is not invertible.

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[newtonRaphson](#), [broyden](#)

**Examples**

```
## Example from Quarteroni & Saleri
F1 <- function(x) c(x[1]^2 + x[2]^2 - 1, sin(pi*x[1]/2) + x[2]^3)
newtonsys(F1, x0 = c(1, 1)) # zero: 0.4760958 -0.8793934

## Find the roots of the complex function sin(z)^2 + sqrt(z) - log(z)
F2 <- function(x) {
  z <- x[1] + x[2]*1i
  fz <- sin(z)^2 + sqrt(z) - log(z)
  c(Re(fz), Im(fz))
}
newtonsys(F2, c(1, 1))
# $zero 0.2555197 0.8948303 , i.e. z0 = 0.2555 + 0.8948i
# $fnorm 2.220446e-16
# $niter 8

## Two more problematic examples
F3 <- function(x)
  c(2*x[1] - x[2] - exp(-x[1]), -x[1] + 2*x[2] - exp(-x[2]))
newtonsys(F3, c(0, 0))
# $zero 0.5671433 0.5671433
# $fnorm 0
# $niter 4

## Not run:
F4 <- function(x) # Dennis Schnabel
  c(x[1]^2 + x[2]^2 - 2, exp(x[1] - 1) + x[2]^3 - 2)
newtonsys(F4, c(2.0, 0.5))
# will result in an error ``missing value in ... err<tol && niter<maxiter''
## End(Not run)
```

---

nextpow2

*Next Power of 2*

---

**Description**

Smallest power of 2 greater than the argument.

**Usage**

```
nextpow2(x)
```

**Arguments**

x numeric scalar, vector, or matrix



**Details**

Computes the smallest integer  $n$  such that  $\text{abs}(x) \leq 2^n$ . If  $x$  is a vector or matrix, returns the result component-wise. For negative or complex values, the absolute value will be taken.

**Value**

an integer  $n$  such that  $x \leq 2^n$ .

**See Also**

[pow2](#)

**Examples**

```
nextpow2(10)           #=> 4
nextpow2(1:10)        #=> 0 1 2 2 3 3 3 3 4 4
nextpow2(-2^10)       #=> 10
nextpow2(.Machine$double.eps) #=> -52
```

---

 nnz

*Nonzero Elements*


---

**Description**

Number of non-zero elements.

**Usage**

```
nnz(x)
```

**Arguments**

$x$  a numeric or complex vector or matrix.

**Value**

the number of non-zero elements of  $x$ .

**See Also**

[find](#)

**Examples**

```
nnz(diag(10))
```

---

Norm	<i>Vector Norm</i>
------	--------------------

---

**Description**

The Norm function calculates several different types of vector norms for  $x$ , depending on the argument  $p$ .

**Usage**

```
Norm(x, p = 2)
```

**Arguments**

$x$	Numeric vector; matrices not allowed.
$p$	Numeric scalar or Inf, -Inf; default is 2

**Details**

Norm returns a scalar that gives some measure of the magnitude of the elements of  $x$ . It is called the  $p$ -norm for values  $-\text{Inf} \leq p \leq \text{Inf}$ , defining Hilbert spaces on  $R^n$ .

Norm( $x$ ) is the Euclidean length of a vector  $x$ ; same as Norm( $x$ , 2).

Norm( $x$ ,  $p$ ) for finite  $p$  is defined as  $\text{sum}(\text{abs}(x)^p)^{1/p}$ .

Norm( $x$ , Inf) returns  $\text{max}(\text{abs}(x))$ , while Norm( $x$ , -Inf) returns  $\text{min}(\text{abs}(x))$ .

**Value**

Numeric scalar (or Inf), or NA if an element of  $x$  is NA.

**Note**

In Matlab/Octave this is called norm; R's norm function norm( $x$ , "F") ('Frobenius Norm') is the same as Norm( $x$ ).

**See Also**

[norm](#) of a matrix

**Examples**

```
Norm(c(3, 4))           #=> 5 Pythagoras triple
Norm(c(1, 1, 1), p=2)  #  sqrt(3)
Norm(1:10, p = 1)     #  sum(1:10)
Norm(1:10, p = 0)    #  Inf
Norm(1:10, p = Inf)  #  max(1:10)
Norm(1:10, p = -Inf) #  min(1:10)
```

---

normest	<i>Estimated Matrix Norm</i>
---------	------------------------------

---

**Description**

Estimate the 2-norm of a real (or complex-valued) matrix. 2-norm is also the maximum absolute eigenvalue of  $M$ , computed here using the power method.

**Usage**

```
normest(M, maxiter = 100, tol = .Machine$double.eps^(1/2))
```

**Arguments**

<code>M</code>	Numeric matrix; vectors will be considered as column vectors.
<code>maxiter</code>	Maximum number of iterations allowed; default: 100.
<code>tol</code>	Tolerance used for stopping the iteration.

**Details**

Estimate the 2-norm of the matrix  $M$ , typically used for large or sparse matrices, where the cost of calculating the norm (A) is prohibitive and an approximation to the 2-norm is acceptable.

Theoretically, the 2-norm of a matrix  $M$  is defined as

$$\|M\|_2 = \max \frac{\|M*x\|_2}{\|x\|_2} \text{ for all } x \neq 0$$

where  $\|\cdot\|_2$  is the Euclidean/Frobenius norm.

**Value**

2-norm of the matrix as a positive real number.

**Note**

If feasible, an accurate value of the 2-norm would simply be calculated as the maximum of the singular values (which are all positive):

```
max(svd(M)$d)
```

**References**

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Philadelphia.

**See Also**

[cond](#), [svd](#)

**Examples**

```
normest(magic(5)) == max(svd(magic(5))$d) # TRUE
normest(magic(100))                      # 500050
```

---

nthroot	<i>Real nth Root</i>
---------	----------------------

---

**Description**

Compute the real n-th root of real numbers.

**Usage**

```
nthroot(x, n)
```

**Arguments**

x	numeric vector or matrix
n	positive integer specifying the exponent $1/n$ .

**Details**

Computes the n-th root real numbers of a numeric vector  $x$ , while  $x^{(1/n)}$  will return NaN for negative numbers, even in case  $n$  is odd. If some numbers in  $x$  are negative,  $n$  must be odd. (This is different in *Octave*)

**Value**

Returns a numeric vector of solutions to  $x^{1/n}$ .

**See Also**

[sqrt](#)

**Examples**

```
nthroot(c(1, -2, 3), 3) #=> 1.000000 -1.259921 1.442250
(-2)^(1/3)             #=> NaN
```

---

nullspace

*Kernel or Nullspace*

---

### Description

Kernel of the linear map defined by matrix M.

### Usage

nullspace(M)  
null(M)

### Arguments

M                      Numeric matrix; vectors will be considered as column vectors.

### Details

The kernel (aka null space/nullspace) of a matrix M is the set of all vectors x for which  $Ax=0$ . It is computed from the QR-decomposition of the matrix.

null is simply an alias for nullspace – and the Matlab name.

### Value

If M is an n-by-m (operating from left on m-dimensional column vectors), then  $N=nullspace(M)$  is a m-by-k matrix whose columns define a (linearly independent) basis of the k-dimensional kernel in  $R^m$ .

If the kernel is only the null vector  $(0\ 0\ \dots\ 0)$ , then NULL will be returned.

As the rank of a matrix is also the dimension of its image, the following relation is true:

$$m = \dim(\text{nullspace}(M)) + \text{rank}(M)$$

### Note

The image of M can be retrieved from `orth()`.

### References

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Philadelphia.

### See Also

[Rank](#), [orth](#), `MASS::Null`

**Examples**

```

M <- matrix(1:12, 3, 4)
Rank(M)           #=> 2
N <- nullspace(M)
#           [,1]      [,2]      [,3]
# [1,] 0.4082483 -0.8164966 0.4082483
M

M1 <- matrix(1:6, 2, 3) # of rank 2
M2 <- t(M1)
nullspace(M1)        # corresponds to 1 -2 1
nullspace(M2)        # NULL, i.e. 0 0

M <- magic(5)
Rank(M)             #=> 5
nullspace(M)        #=> NULL, i.e. 0 0 0 0 0

```

---

numderiv

*Richardson's Numerical Derivative*


---

**Description**

Richardson's method applied to the computation of the numerical derivative.

**Usage**

```

numderiv(f, x0, maxiter = 16, h = 1/2, ..., tol = .Machine$double.eps)

numdiff(f, x, maxiter = 16, h = 1/2, ..., tol = .Machine$double.eps)

```

**Arguments**

f	function to be differentiated.
x0, x	point(s) at which the derivative is to be computed.
maxiter	maximum number of iterations.
h	starting step size, should be the default h=0.5.
tol	relative tolerance.
...	variables to be passed to function f.

**Details**

numderiv returns the derivative of f at x0, where x0 must be a single scalar in the domain of the function.

numdiff is a vectorized form of numderiv such that the derivatives will be returned at all points of the vector x.

**Value**

Numeric scalar or vector of approximated derivatives.

**Note**

See `grad` in the ‘`numDeriv`’ package for another implementation of Richardson’s method in the context of numerical differentiation.

**References**

Mathews, J. H., and K. D. Fink (1999). *Numerical Methods Using Matlab*. Third Edition, Prentice Hall.

**See Also**

[fderiv](#), [complexstep](#)

**Examples**

```
# Differentiate an anti-derivative function
f <- function(x) sin(x)*sqrt(1+sin(x))
F <- function(x)
  integrate(f, 0, x, rel.tol = 1e-12)$value
x0 <- 1
dF0 <- numderiv(F, x0, tol = 6.5e-15)   #=> 1.141882942715462
f(x0)                                  # 1.141882942715464 true value
# fderiv(F, x0)                         # 1.141882942704476
# numDeriv::grad(F, x0)                 # 1.141882942705797

# Compare over a whole period
x <- seq(0, 2*pi, length.out = 11)
max(abs(numdiff(sin, x) - cos(x)))      #=> 3.44e-15
# max(abs(numDeriv::grad(sin, x) - cos(x))) # 7.70e-12

# Example from complex step
f <- function(x) exp(x) / sqrt(sin(x)^3 + cos(x)^3)
x0 <- 1.5
numderiv(f, x0)                         # 4.05342789389876, error 0.5e-12
# 4.053427893898621... true value
```

---

numel

*Number of Elements*

---

**Description**

Number of elements in a vector, matrix, or array.

**Usage**

`numel(x)`

**Arguments**

`x` a vector, matrix, array or list

**Value**

the number of elements of `a`.

**See Also**

[size](#)

**Examples**

```
numel(c(1:12))
numel(matrix(1:12, 3, 4))
```

---

ode23

*Non-stiff (and stiff) ODE solvers*

---

**Description**

Runge-Kutta (2, 3)-method with variable step size, resp. (4,5)-method with Dormand-Price coefficients, or (7,8)-pairs with Fehlberg coefficients. The function  $f(t, y)$  has to return the derivative as a column vector.

**Usage**

```
ode23(f, t0, tfinal, y0, ..., rtol = 1e-3, atol = 1e-6)
```

```
ode23s(f, t0, tfinal, y0, jac = NULL, ...,
        rtol = 1e-03, atol = 1e-06, hmax = 0.0)
```

```
ode45(f, t0, tfinal, y0, ..., atol = 1e-6, hmax = 0.0)
```

```
ode78(f, t0, tfinal, y0, ..., atol = 1e-6, hmax = 0.0)
```

**Arguments**

`f` function in the differential equation  $y' = f(x, y)$ ; defined as a function  $R \times R^m \rightarrow R^m$ , where  $m$  is the number of equations.

`t0, tfinal` start and end points of the interval.

`y0` starting values as column vector; for  $m$  equations `y0` needs to be a vector of length  $m$ .

`jac` jacobian of `f` as a function of `x` alone; if not specified, a finite difference approximation will be used.

`rtol, atol` relative and absolute tolerance.

`hmax` maximal step size, default is  $(tfinal - t0)/10$ .

`...` Additional parameters to be passed to the function.



**Details**

ode23 is an integration method for systems of ordinary differential equations using second and third order Runge-Kutta-Fehlberg formulas with automatic step-size.

ode23s can be used to solve a stiff system of ordinary differential equations, based on a modified Rosenbrock triple method of order (2,3); See section 4.1 in [Shampine and Reichelt].

ode45 implements Dormand-Prince (4,5) pair that minimizes the local truncation error in the 5th-order estimate which is what is used to step forward (local extrapolation). Generally it produces more accurate results and costs roughly the same computationally.

ode78 implements Fehlberg's (7,8) pair and is a 7th-order accurate integrator therefore the local error normally expected is  $O(h^8)$ . However, because this particular implementation uses the 8th-order estimate for xout (i.e. local extrapolation) moving forward with the 8th-order estimate will yield errors on the order of  $O(h^9)$ . It requires 13 function evaluations per integration step.

**Value**

List with components t for grid (or 'time') points between t0 and tfinal, and y an n-by-m matrix with solution variables in columns, i.e. each row contains one time stamp.

**Note**

Copyright (c) 2004 C. Moler for the Matlab textbook version ode23tx.

**References**

Ascher, U. M., and L. R. Petzold (1998). Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. SIAM.

L.F. Shampine and M.W. Reichelt (1997). The MATLAB ODE Suite. SIAM Journal on Scientific Computing, Vol. 18, pp. 1-22.

Moler, C. (2004). Numerical Computing with Matlab. Revised Reprint, SIAM. <https://www.mathworks.com/moler/chapters.html>.

**See Also**

[rk4sys](#), [deval](#)

**Examples**

```
## Example1: Three-body problem
f <- function(t, y)
as.matrix(c(y[2]*y[3], -y[1]*y[3], 0.51*y[1]*y[2]))
y0 <- as.matrix(c(0, 1, 1))
t0 <- 0; tf <- 20
sol <- ode23(f, t0, tf, y0, rtol=1e-5, atol=1e-10)
## Not run:
matplot(sol$t, sol$y, type = "l", lty = 1, lwd = c(2, 1, 1),
        col = c("darkred", "darkblue", "darkgreen"),
        xlab = "Time [min]", ylab = "",
        main = "Three-body Problem")
```

```

grid()
## End(Not run)

## Example2: Van der Pol Equation
#  $x'' + (x^2 - 1)x' + x = 0$ 
f <- function(t, x)
  as.matrix(c(x[1] * (1 - x[2]^2) - x[2], x[1]))
t0 <- 0; tf <- 20
x0 <- as.matrix(c(0, 0.25))
sol <- ode23(f, t0, tf, x0)
## Not run:
plot(c(0, 20), c(-3, 3), type = "n",
      xlab = "Time", ylab = "", main = "Van der Pol Equation")
lines(sol$t, sol$y[, 1], col = "blue")
lines(sol$t, sol$y[, 2], col = "darkgreen")
grid()
## End(Not run)

## Example3: Van der Pol as stiff equation
vdP <- function(t,y) as.matrix(c(y[2], 10*(1-y[1]^2)*y[2]-y[1]))
ajax <- function(t, y)
  matrix(c(0, 1, -20*y[1]*y[2]-1, 10*(1-y[1]^2)), 2,2, byrow = TRUE)
sol <- ode23s(vdP, t0, tf, c(2, 0), jac = ajax, hmax = 1.0)
## Not run:
plot(sol$t, sol$y[, 1], col = "blue")
lines(sol$t, sol$y[, 1], col = "blue")
lines(sol$t, sol$y[, 2]/8, col = "red", lwd = 2)
grid()
## End(Not run)

## Example4: pendulum
m = 1.0; l = 1.0 # [kg] resp. [m]
g = 9.81; b = 0.7 # [m/s^2] resp. [N s/m]
fp = function(t, x)
  c( x[2] , 1/(1/3*m*l^2)*(-b*x[2]-m*g*l/2*sin(x[1])) )
t0 <- 0.0; tf <- 5.0; hmax = 0.1
y0 = c(30*pi/180, 0.0)
sol = ode45(fp, t0, tf, y0, hmax = 0.1)
## Not run:
matplot(sol$t, sol$y, type = "l", lty = 1)
grid()
## End(Not run)

## Example: enforced pendulum
g <- 9.81
L <- 1.0; Y <- 0.25; w <- 2.5
f <- function(t, y) {
  as.matrix(c(y[2], -g/L * sin(y[1]) + w^2/L * Y * cos(y[1]) * sin(w*t)))
}
y0 <- as.matrix(c(0, 0))
sol <- ode78(f, 0.0, 60.0, y0, hmax = 0.05)
## Not run:
plot(sol$t, sol$y[, 1], type="l", col="blue")

```

```
grid()  
## End(Not run)
```

---

odregress	<i>Orthogonal Distance Regression</i>
-----------	---------------------------------------

---

### Description

Orthogonal Distance Regression (ODR, a.k.a. total least squares) is a regression technique in which observational errors on both dependent and independent variables are taken into account.

### Usage

```
odregress(x, y)
```

### Arguments

x	matrix of independent variables.
y	vector representing dependent variable.

### Details

The implementation used here is applying PCA resp. the singular value decomposition on the matrix of independent and dependent variables.

### Value

Returns list with components `coeff` linear coefficients and intercept term, `ssq` sum of squares of orthogonal distances to the linear line or hyperplane, `err` the orthogonal distances, `fitted` the fitted values, `resid` the residuals, and `normal` the normal vector to the hyperplane.

### Note

The "geometric mean" regression not implemented because questionable.

### References

Golub, G.H., and C.F. Van Loan (1980). An analysis of the total least squares problem. Numerical Analysis, Vol. 17, pp. 883-893.

See ODRPACK or ODRPACK95 (TOMS Algorithm 676).  
URL: [https://docs.scipy.org/doc/external/odr\\_ams.pdf](https://docs.scipy.org/doc/external/odr_ams.pdf)

### See Also

[lm](#)

**Examples**

```

# Example in one dimension
x <- c(1.0, 0.6, 1.2, 1.4, 0.2)
y <- c(0.5, 0.3, 0.7, 1.0, 0.2)
odr <- odregress(x, y)
( cc <- odr$coeff )
# [1] 0.65145762 -0.03328271
lm(y ~ x)
# Coefficients:
# (Intercept)          x
#   -0.01379         0.62931

# Prediction
xnew <- seq(0, 1.5, by = 0.25)
( ynew <- cbind(xnew, 1) %*% cc )

## Not run:
plot(x, y, xlim=c(0, 1.5), ylim=c(0, 1.2), main="Orthogonal Regression")
abline(lm(y ~ x), col="blue")
lines(c(0, 1.5), cc[1]*c(0, 1.5) + cc[2], col="red")
points(xnew, ynew, col = "red")
grid()
## End(Not run)

# Example in two dimensions
x <- cbind(c(0.92, 0.89, 0.85, 0.05, 0.62, 0.55, 0.02, 0.73, 0.77, 0.57),
          c(0.66, 0.47, 0.40, 0.23, 0.17, 0.09, 0.92, 0.06, 0.09, 0.60))
y <- x %*% c(0.5, 1.5) + 1
odr <- odregress(x, y); odr
# $coeff
# [1] 0.5 1.5 1.0
# $ssq
# [1] 1.473336e-31

y <- y + rep(c(0.1, -0.1), 5)
odr <- odregress(x, y); odr
# $coeff
# [1] 0.5921823 1.6750269 0.8803822
# $ssq
# [1] 0.02168174

lm(y ~ x)
# Coefficients:
# (Intercept)          x1          x2
#   0.9153         0.5671         1.6209

```

**Description**

Range space or image of a matrix.

**Usage**

```
orth(M)
```

**Arguments**

M                      Numeric matrix; vectors will be considered as column vectors.

**Details**

$B = \text{orth}(A)$  returns an orthonormal basis for the range of A. The columns of B span the same space as the columns of A, and the columns of B are orthogonal to each other.

The number of columns of B is the rank of A.

**Value**

Matrix of orthogonal columns, spanning the image of M.

**References**

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Philadelphia.

**See Also**

[nullspace](#)

**Examples**

```
M <- matrix(1:12, 3, 4)
Rank(M)                      #=> 2
orth(M)
```

---

pade

*Pade Approximation*

---

**Description**

A Pade approximation is a rational function (of a specified order) whose power series expansion agrees with a given function and its derivatives to the highest possible order.

**Usage**

```
pade(p1, p2 = c(1), d1 = 5, d2 = 5)
```

**Arguments**

p1	polynomial representing or approximating the function, preferably the Taylor series of the function around some point.
p2	if present, the function is given as p1/p2.
d1	the degree of the numerator of the rational function.
d2	the degree of the denominator of the rational function.

**Details**

The relationship between the coefficients of p1 (and p2) and r1 and r2 is determined by a system of linear equations. The system is then solved by applying the pseudo-inverse `pinv` for for the left-hand matrix.

**Value**

List with components r1 and r2 for the numerator and denominator polynomials, i.e. r1/r2 is the rational approximation sought.

**Note**

In general, errors for Pade approximations are smallest when the degrees of numerator and denominator are the same or when the degree of the numerator is one larger than that of the denominator.

**References**

Press, W. H., S. A. Teukolsky, W. T Vetterling, and B. P. Flannery (2007). Numerical Recipes: The Art of Numerical Computing. Third Edition, Cambridge University Press, New York.

**See Also**

[taylor](#), [ratInterp](#)

**Examples**

```
## Exponential function
p1 <- c(1/24, 1/6, 1/2, 1.0, 1.0) # Taylor series of exp(x) at x=0
R <- pade(p1); r1 <- R$r1; r2 <- R$r2
f1 <- function(x) polyval(r1, x) / polyval(r2, x)
## Not run:
xs <- seq(-1, 1, length.out=51); ys1 <- exp(xs); ys2 <- f1(xs)
plot(xs, ys1, type = "l", col="blue")
lines(xs, ys2, col = "red")
grid()
## End(Not run)
```

---

pascal	<i>Pascal Triangle</i>
--------	------------------------

---

**Description**

Pascal triangle in matrix format

**Usage**

```
pascal(n, k = 0)
```

**Arguments**

n	natural number
k	natural number, $k \leq n$

**Details**

Pascal triangle with k variations.

**Value**

matrix representing the Pascal triangle

**See Also**

nchoosek

**Examples**

```
pascal(5)
pascal(5, 1)
pascal(5, 2)
```

---

pchip	<i>Hermitean Interpolation Polynomials</i>
-------	--

---

**Description**

Piecewise Cubic Hermitean Interpolation Polynomials.

**Usage**

```
pchip(xi, yi, x)
```

```
pchipfun(xi, yi)
```

**Arguments**

<code>xi, yi</code>	x- and y-coordinates of supporting nodes.
<code>x</code>	x-coordinates of interpolation points.

**Details**

pchip is a 'shape-preserving' piecewise cubic Hermite polynomial approach that attempts to determine slopes such that function values do not overshoot data values. pchipfun is a wrapper around pchip and returns a function. Both pchip and the function returned by pchipfun are vectorized.

`xi` and `yi` must be vectors of the same length greater or equal 3 (for cubic interpolation to be possible), and `xi` must be sorted. pchip can be applied to points outside  $[\min(xi), \max(xi)]$ , but the result does not make much sense outside this interval.

**Value**

Values of interpolated data at points `x`.

**Author(s)**

Copyright of the Matlab version from Cleve Moler in his book "Numerical Computing with Matlab", Chapter 3 on Interpolation. R Version by Hans W. Borchers, 2011.

**References**

Moler, C. (2004). Numerical Computing with Matlab. Revised Reprint, SIAM.

**See Also**

[interp1](#)

**Examples**

```
x <- c(1, 2, 3, 4, 5, 6)
y <- c(16, 18, 21, 17, 15, 12)
pchip(x, y, seq(1, 6, by = 0.5))
fp <- pchipfun(x, y)
fp(seq(1, 6, by = 0.5))

## Not run:
plot(x, y, col="red", xlim=c(0,7), ylim=c(10,22),
     main = "Spline and 'pchip' Interpolation")
grid()

xs <- seq(1, 6, len=51)
ys <- interp1(x, y, xs, "spline")
lines(xs, ys, col="cyan")
yp <- pchip(x, y, xs)
lines(xs, yp, col = "magenta")
## End(Not run)
```



---

peaks *Peaks Function (Matlab Style)*

---

**Description**

An example functions in two variables, with peaks.

**Usage**

```
peaks(v = 49, w)
```

**Arguments**

v                      vector, whose length will be used, or a natural number.  
w                      another vector, will be used in meshgrid(x,y).

**Details**

peaks is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating three-dimensional plots.

**Value**

Returns three matrices as a list with X, Y, and Z components, the first two being the result of the meshgrid function, and Z the application of the following function at the points of X and Y:

$$z <- 3 * (1-x)^2 * \exp(-(x^2) - (y+1)^2) - 10 * (x/5 - x^3 - y^5) * \exp(-x^2 - y^2) - 1/3 * \exp(-(x+1)^2 - y^2)$$
**Note**

The variant that peaks() will display the 3-dim. graph as in Matlab is not yet implemented.

**See Also**

[meshgrid](#)

**Examples**

```
peaks(3)
## Not run:
P <- peaks()
x <- P$X[1,]; y <- P$Y[, 1]
persp(x, y, P$Z)

## End(Not run)
```

---

perms

*Generate Permutations*

---

### Description

Generates all permutations of a vector a.

### Usage

perms(a)

### Arguments

a                    numeric vector of some length n

### Details

If a is a vector of length n, generate all permutations of the elements in a as a matrix of size  $n! \times n$  where each row represents one permutation.

A matrix will be expanded as vector.

### Value

matrix of permutations of the elements of a

### Note

Not feasible for  $\text{length}(a) > 10$ .

### See Also

[randperm](#)

### Examples

```
perms(6)
perms(1:6)
perms(c(1, exp(1), pi))
```

---

piecewise

*Piecewise Linear Function*

---

### Description

Compute zeros and area of a piecewise linear function.

### Usage

```
piecewise(x, y, abs = FALSE)
```

### Arguments

x, y	x- and y-coordinates of points defining the piecewise linear function
abs	logical; shall the integral or the total area between the x-axis and the function be calculated

### Details

Compute zeros and integral resp. area of a piecewise linear function given by points with x and y as coordinates.

### Value

Returns a list with the integral or area as first element and the vector as all zeroes as second.

### See Also

[trapz](#)

### Examples

```
x <- c(0, 2, 3, 4, 5)
y <- c(2, -2, 0, -2, 0)
piecewise(x, y)
piecewise(x, y, abs=TRUE)
```

pinv

*Pseudoinverse or Generalized Inverse***Description**

Computes the Moore-Penrose generalized inverse of a matrix.

**Usage**

```
pinv(A, tol=.Machine$double.eps^(2/3))
```

**Arguments**

A                    real or complex matrix  
tol                   tolerance used for assuming an eigenvalue is zero.

**Details**

Compute the generalized inverse  $B$  of a matrix  $A$  using the singular value decomposition `svd()`. This generalized inverse is characterized by this equation:  $A \%*\% B \%*\% A == A$

The pseudoinverse  $B$  solves the problem to minimize  $|Ax - b|$  by setting  $x = Bb$

```
s <- svd(A)
D <- diag(s\[d])
Dinv <- diag(1/s\[d])
U <- s\[u]; V <- s\[v]
X = V Dinv t(U)
```

Thus  $B$  is computed as  $s\$v \%*\% \text{diag}(1/s\$d) \%*\% t(s\$u)$ .

**Value**

The pseudoinverse of matrix  $A$ .

**Note**

The pseudoinverse or ‘generalized inverse’ is also provided by the function `ginv()` in package ‘MASS’. It is included in a somewhat simplified way to be independent of that package.

**References**

Ben-Israel, A., and Th. N. E. Greville (2003). *Generalized Inverses - Theory and Applications*. Springer-Verlag, New York.

**See Also**

MASS::ginv

**Examples**

```
A <- matrix(c(7,6,4,8,10,11,12,9,3,5,1,2), 3, 4)
b <- apply(A, 1, sum) # 32 16 20 row sum
x <- pinv(A) %*% b
A %*% x             #=> 32 16 20 as column vector
```

plotyy

*Plotting Two y-Axes***Description**

Line plot with y-axes on both left and right side.

**Usage**

```
plotyy(x1, y1, x2, y2, gridp = TRUE, box.col = "grey",
        type = "l", lwd = 1, lty = 1,
        xlab = "x", ylab = "y", main = "",
        col.y1 = "navy", col.y2 = "maroon", ...)
```

**Arguments**

x1, x2	x-coordinates for the curves
y1, y2	the y-values, with ordinates y1 left, y2 right.
gridp	logical; shall a grid be plotted.
box.col	color of surrounding box.
type	type of the curves, line or points (for both data).
lwd	line width (for both data).
lty	line type (for both data).
xlab, ylab	text below and on the left.
main	main title of the plot.
col.y1, col.y2	colors to be used for the lines or points.
...	additional plotting parameters.

**Details**

Plots y1 versus x1 with y-axis labeling on the left and plots y2 versus x2 with y-axis labeling on the right.

The x-values should not be too far apart. To exclude certain points, use NA values. Both curves will be line or point plots, and have the same line type and width.

**Value**

Generates a graph, no return values.

**See Also**

plotrix::twoord.plot

**Examples**

```
## Not run:
x <- seq(0, 20, by = 0.01)
y1 <- 200*exp(-0.05*x)*sin(x)
y2 <- 0.8*exp(-0.5*x)*sin(10*x)

plotyy(x, y1, x, y2, main = "Two-ordinates Plot")

## End(Not run)
```

---

poisson2disk

*Poisson Disk Sampling*

---

**Description**

Approximate Poisson disk distribution of points in a rectangle.

**Usage**

```
poisson2disk(n, a = 1, b = 1, m = 10, info = TRUE)
```

**Arguments**

n	number of points to generate in a rectangle.
a, b	width and height of the rectangle
m	number of points to try in each step.
info	shall additional info be printed.

**Details**

Realizes Mitchell's best-candidate algorithm for creating a Poisson disk distribution on a rectangle. Can be used for sampling, and will be more appropriate in some sampling applications than uniform sampling or grid-like sampling.

With  $m = 1$  uniform sampling will be generated.

**Value**

Returns the points as a matrix with two columns for x- and y-coordinates. Prints the minimal distance between points generated.

**Note**

Bridson's algorithm for Poisson disk sampling may be added later as an alternative. Also a variant that generates points in a circle.

## References

A. Lagae and Ph. Dutre. A Comparison of Methods for Generating Poisson Disk Distributions. Computer Graphics Forum, Vol. 27(1), pp. 114-129, 2008. URL: [citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.192.58](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.192.58)

## Examples

```
set.seed(1111)
P <- poisson2disk(n = 20, m = 10)
head(P)
##           [,1]           [,2]
## [1,] 0.46550264 0.41292487
## [2,] 0.13710541 0.98737065
## [3,] 0.96028255 0.83222920
## [4,] 0.06044078 0.09325431
## [5,] 0.78579426 0.09267546
## [6,] 0.49670274 0.99852771

# Plotting points
# plot(P, pch = 'x', col = "blue")
```

---

polar

*Polar Coordinate Plot (Matlab Style)*

---

## Description

The polar function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

## Usage

```
polar(t, r, type="l",
      col = "blue", grcol = "darkgrey", bxcol = "black",
      main = "Polar Plot", add = FALSE, ...)
```

## Arguments

t, r	vectors specifying angle and radius.
type	type of the plot, lines, points, or no plotting.
col	color of the graph.
grcol, bxcol	color of grid and box around the plot.
main	plot title.
add	logical; if true, the graph will be plotted into the coordinate system of an existing plot.
...	plotting parameters to be passed to the points function.

**Details**

`polar(theta, rho)` creates a polar coordinate plot of the angle `theta` versus the radius `rho`. `theta` is the angle from the x-axis to the radius vector specified in radians; `rho` is the length of the radius vector.

**Value**

Generates a plot; no returns.

**Examples**

```
## Not run:
t <- deg2rad(seq(0, 360, by = 2))
polar(t, cos(2*t), bxcyl = "white", main = "Sine and Cosine")
polar(t, sin(2*t), col = "red", add = TRUE)

## End(Not run)
```

---

Poly

*Define Polynomial by Roots*

---

**Description**

Define a polynomial by its roots.

**Usage**

`Poly(x)`

**Arguments**

`x` vector or square matrix, real or complex

**Details**

Computes the characteristic polynomial of an  $(n \times n)$ -Matrix.

If `x` is a vector, `Poly(x)` is the vector of coefficients of the polynomial whose roots are the elements of `x`.

**Value**

Vector representing a polynomial.

**Note**

In Matlab/Octave this function is called `poly()`.



**See Also**[polyval](#), [roots](#)**Examples**

```
Poly(c(1, -1, 1i, -1i)) # Solves x^4 -1 = 0
# Wilkinson's example:
roots(Poly(1:20))
```

---

poly2str

*Print Polynomial*

---

**Description**

Print polynomial as a character string.

**Usage**

```
poly2str(p, svar = "x", smul = "*", d = options("digits")$digits)
```

**Arguments**

p	numeric vector representing a polynomial
svar	character representing the unknown, default x.
smul	multiplication symbol, default *.
d	significant digits, default options("digits").

**Details**

Simple string manipulation.

**Value**

Returns the usual string representing a polynomial in mathematics.

**Examples**

```
poly2str(c(0))
poly2str(c(1, -1, 1, -1, 1))
poly2str(c(0, 1e-6, 1e6), d = 2)
```

---

polyadd

*Adding Polynomials*

---

### Description

Add two polynomials given as vectors.

### Usage

```
polyadd(p, q)
```

### Arguments

p, q            Vectors representing two polynomials.

### Details

Polynomial addition realized simply by multiplying and summing up all the coefficients after extending vectors to the same length.

### Value

Vector representing a polynomial.

### Note

There is no such function in Matlab or Octave.

### See Also

[conv](#)

### Examples

```
polyadd(c(1, 1, 1), 1)
polyadd(c(1, 1, 1), c(0, 0, 1))
polyadd(c(-0.5, 1, -1), c(0.5, 0, 1))
```

---

polyApprox

*Polynomial Approximation*

---

### Description

Generate a polynomial approximation.

### Usage

```
polyApprox(f, a, b, n, ...)
```

### Arguments

f	function to be approximated.
a, b	end points of the interval.
n	degree of the polynomial.
...	further variables for function f.

### Details

Uses the Chebyshev coefficients to derive polynomial coefficients.

### Value

List with four components:

p	the approximating polynomial.
f	a function evaluating this polynomial.
cheb.coeff	the Chebyshev coefficients.
estim.prec	the estimated precision over the given interval.

### Note

The Chebyshev approximation is optimal in the sense of the  $L^1$  norm, but not as a solution of the *minimax* problem; for this, an application of the Remez algorithm is needed.

### References

Carothers, N. L. (1998). *A Short Course on Approximation Theory*. Bowling Green State University.

### See Also

[chebApprox](#), [polyfit](#)

**Examples**

```
## Example
# Polynomial approximation for sin
polyApprox(sin, -pi, pi, 9)
# $p
# [1] 2.197296e-06 0.000000e+00 -1.937495e-04 0.000000e+00 8.317144e-03
# [6] 0.000000e+00 -1.666468e-01 0.000000e+00 9.999961e-01 0.000000e+00
#
# $f
# function (x)
# polyval(p, x)
#
# $cheb.coeff
# [1] 0.06549943 0.00000000 -0.58518036 0.00000000 2.54520983 0.00000000
# [7] -5.16709776 0.00000000 3.14158037 0.00000000
#
# $estim.prec
# [1] 1.151207e-05

## Not run:
f <- polyApprox(sin, -pi, pi, 9)$f
x <- seq(-pi, pi, length.out = 100)
y <- sin(x) - f(x)
plot(x, y, type = "l", col = "blue")
grid()
## End(Not run)
```

---

polyarea

*Area of a Polygon*

---

**Description**

Calculates the area and length of a polygon given by the vertices in the vectors *x* and *y*.

**Usage**

polyarea(*x*, *y*)

poly\_length(*x*, *y*)

poly\_center(*x*, *y*)

poly\_crossings(L1, L2)

**Arguments**

*x* x-coordinates of the vertices defining the polygon

*y* y-coordinates of the vertices

L1, L2 matrices of type 2xn with x- and y-coordinates.

**Details**

polyarea calculates the area of a polygon defined by the vertices with coordinates  $x$  and  $y$ . Areas to the left of the vertices are positive, those to the right are counted negative.

The computation is based on the Gauss polygon area formula. The polygon automatically be closed, that is the last point need not be / should not be the same as the first.

If some points of self-intersection of the polygon line are not in the vertex set, the calculation will be inexact. The sum of all areas will be returned, parts that are circulated in the mathematically negative sense will be counted as negative in this sum.

If  $x$ ,  $y$  are matrices of the same size, the areas of all polygons defined by corresponding columns are computed.

poly\_center calculates the center (of mass) of the figure defined by the polygon. Self-intersections should be avoided in this case. The mathematical orientation of the polygon does not have influence on the center coordinates.

poly\_length calculates the length of the polygon

poly\_crossings calculates the crossing points of two polygons given as matrices with  $x$ - and  $y$ -coordinates in the first and second row. Can be used for finding the crossing points of parametrised curves.

**Value**

Area or length of the polygon resp. sum of the enclosed areas; or the coordinates of the center of gravity.

poly\_crossings returns a matrix with column names  $x$  and  $y$  representing the crossing points.

**See Also**

[trapz](#), [arclength](#)

**Examples**

```
# Zu Chongzhi's calculation of pi (China, about 480 A.D.),
# approximating the circle from inside by a regular 12288-polygon(!):
phi <- seq(0, 2*pi, len=3*2^12+1)
x <- cos(phi)
y <- sin(phi)
pi_approx <- polyarea(x, y)
print(pi_approx, digits=8)      #=> 3.1415925 or 355/113

poly_length(x, y)              #=> 6.2831852 where 2*pi is 6.2831853

x1 <- x + 0.5; y1 <- y + 0.5
x2 <- rev(x1); y2 <- rev(y1)
poly_center(x1, y1)            #=> 0.5 0.5
poly_center(x2, y2)            #=> 0.5 0.5

# A simple example
L1 <- matrix(c(0, 0.5, 1, 1, 2,
              0, 1, 1, 0.5, 0), nrow = 2, byrow = TRUE)
```

```

L2 <- matrix(c(0.5, 0.75, 1.25, 1.25,
              0, 0.75, 0.75, 0 ), nrow = 2, byrow = TRUE)
P <- poly_crossings(L1, L2)
P
##           x      y
## [1,] 1.00 0.750
## [2,] 1.25 0.375

## Not run:
# Crossings of Logarithmic and Archimedian spirals
# Logarithmic spiral
a <- 1; b <- 0.1
t <- seq(0, 5*pi, length.out = 200)
x1 <- a*exp(b*t)*cos(t) - 1
y1 <- a*exp(b*t)*sin(t)
plot(x1, y1, type = "l", lwd = 2, col = "blue",
     xlim = c(-6, 3), ylim = c(-3, 4), xlab = "", ylab = "",
     main = "Intersecting Logarithmic and Archimedian spirals")
grid()

# Archimedian spiral
a <- 0; b <- 0.25
r <- a + b*t
xa <- r * cos(t)
ya <- r*sin(t)
lines(xa, ya, type = "l", lwd = 2, col = "red")
legend(-6.2, -1.0, c("Logarithmic", "Archimedian"),
      lwd = 2, col = c("blue", "red"), bg = "whitesmoke")

L1 <- rbind(x1, y1)
L2 <- rbind(xa, ya)
P <- poly_crossings(L1, L2)
points(P)

## End(Not run)

```

---

polyder

*Derivative of Polynomial*

---

### Description

Differentiate polynomials.

### Usage

```
polyder(p, q)
```

### Arguments

p	polynomial p given as a vector
q	polynomial q given as a vector

**Details**

Calculates the derivative of polynomials and polynomial products.

polyder(p) returns the derivative of p while polyder(p, q) returns the derivative of the product of the polynomials p and q.

**Value**

a vector representing a polynomial

**See Also**

[polyval](#), [polyint](#)

**Examples**

```
polyder(c(3, 6, 9), c(1, 2, 0)) # 12 36 42 18
```

---

polyfit,polyfix      *Fitting by Polynomial*

---

**Description**

Polynomial curve fitting

**Usage**

```
polyfit(x, y, n)
```

```
polyfix(x, y, n, xfix, yfix)
```

**Arguments**

x	x-coordinates of points
y	y-coordinates of points
n	degree of the fitting polynomial
xfix,yfix	x- and y-coordinates of points to be fixed

**Details**

polyfit finds the coefficients of a polynomial of degree n fitting the points given by their x, y coordinates in a least-squares sense. In polyfit, if x, y are matrices of the same size, the coordinates are taken elementwise. Complex values are not allowed.

polyfix finds a polynomial that fits the data in a least-squares sense, but also passes exactly through all the points with coordinates xfix and yfix. Degree n should be greater or equal to the number of fixed points, but not too big to avoid 'singular matrix' or similar error messages

**Value**

vector representing a polynomial.

**Note**

Please note that `polyfit2` has been removed since 1.9.3; please use `polyfix` instead.

**See Also**

[poly](#), [polyval](#)

**Examples**

```
# Fitting the sine function by a polynomial
x <- seq(0, pi, length.out=25)
y <- sin(x)
p <- polyfit(x, y, 6)

## Not run:
# Plot sin and fitted polynomial
plot(x, y, type="b")
yf <- polyval(p, x)
lines(x, yf, col="red")
grid()
## End(Not run)

## Not run:
n <- 3
N <- 100
x <- linspace(0, 2*pi, N); y = sin(x) + 0.1*rnorm(N)
xfix <- c(0, 2*pi); yfix = c(0, 0)

xs <- linspace(0, 2*pi); ys <- sin(xs)
plot(xs, ys, type = 'l', col = "gray",
     main = "Polynom Approximation of Degree 3")
grid()
points(x, y, pch='o', cex=0.5)
points(xfix, yfix, col = "darkred")

p0 <- polyfit(x, y, n)
lines(xs, polyval(p0, xs), col = "blue")

p1 <- polyfix(x, y, n, xfix, yfix)
lines(xs, polyval(p1, xs), col = "red")

legend(4, 1, c("sin", "polyfit", "polyfix"),
      col=c("gray", "blue", "red"), lty=c(1,1,1))
## End(Not run)
```



---

polyint	<i>Anti-derivative of Polynomial</i>
---------	--------------------------------------

---

**Description**

Integrate polynomials.

**Usage**

```
polyint(p, k)
```

**Arguments**

p	polynomial p given as a vector
k	an integration constant

**Details**

Calculates the integral, i.e. the antiderivative, of a polynomial and adds a constant of integration k if given, else 0.

**Value**

a vector representing a polynomial

**See Also**

[polyval](#), [polyder](#)

**Examples**

```
polyint(c(1, 1, 1, 1, 1), 1)
```

---

polylog	<i>Polylogarithm Function</i>
---------	-------------------------------

---

**Description**

Computes the n-based polylogarithm of z:  $Li_n(z)$ .

**Usage**

```
polylog(z, n)
```

**Arguments**

z	real number or vector, all entries satisfying $\text{abs}(z) < 1$ .
n	base of polylogarithm, integer greater or equal -4.

**Details**

The Polylogarithm is also known as Jonquiere's function. It is defined as

$$\sum_{k=1}^{\infty} z^k / k^n = z + z^2 / 2^n + \dots$$

The polylogarithm function arises, e.g., in Feynman diagram integrals. It also arises in the closed form of the integral of the Fermi-Dirac and the Bose-Einstein distributions.

The special cases  $n=2$  and  $n=3$  are called the dilogarithm and trilogarithm, respectively.

Approximation should be correct up to at least 5 digits for  $|z| > 0.55$  and on the order of 10 digits for  $|z| \leq 0.55$ .

**Value**

Returns the function value (not vectorized).

**Note**

Based on some equations, see references. A Matlab implementation is available in the Matlab File Exchange.

**References**

V. Bhagat, et al. (2003). On the evaluation of generalized BoseEinstein and FermiDirac integrals. Computer Physics Communications, Vol. 155, p.7.

**Examples**

```
polylog(0.5, 1)    # polylog(z, 1) = -log(1-z)
polylog(0.5, 2)    # (p1^2 - 6*log(2)^2) / 12
polylog(0.5, 3)    # (4*log(2)^3 - 2*pi^2*log(2) + 21*zeta(3)) / 24
polylog(0.5, 0)    # polylog(z, 0) = z/(1-z)
polylog(0.5, -1)   # polylog(z, -1) = z/(1-z)^2
```

---

polymul, polydiv      *Multiplying and Dividing Polynomials*

---

**Description**

Multiply or divide two polynomials given as vectors.

**Usage**

```
polymul(p, q)
```

```
polydiv(p, q)
```

**Arguments**

p, q                  Vectors representing two polynomials.

**Details**

Polynomial multiplication realized simply by multiplying and summing up all the coefficients. Division is an alias for deconv. Polynomials are defined from highest to lowest coefficient.

**Value**

Vector representing a polynomial. For division, it returns a list with 'd' the result of the division and 'r' the rest.

**Note**

conv also realizes polynomial multiplication, through Fast Fourier Transformation, with the drawback that small imaginary parts may evolve. deconv can also be used for polynomial division.

**See Also**

conv, deconv

**Examples**

```
# Multiply x^2 + x + 1 with itself
polymul(c(1, 1, 1), c(0, 1, 1, 1))  #=> 1 2 3 2 1

polydiv(c(1, 2, 3, 2, 1), c(1, 1, 1))
#=> d = c(1,1,1); #=> r = c(0.000000e+00 -1.110223e-16)
```

---

polypow

*Polynomial Powers*

---

### Description

Power of a polynomial.

### Usage

polypow(p, n)

### Arguments

p                    vector representing a polynomial.  
n                    positive integer, the exponent.

### Details

Uses `polymul` to multiply the polynomial `p` `n` times with itself.

### Value

Vector representing a polynomial.

### Note

There is no such function in Matlab or Octave.

### See Also

[polymul](#)

### Examples

```
polypow(c(1, -1), 6)                    #=> (x - 1)^6 = (1 -6 15 -20 15 -6 1)
polypow(c(1, 1, 1, 1, 1, 1), 2)        # 1 2 3 4 5 6 5 4 3 2 1
```

---

polytrans, polygcf      *Polynomial Transformations*

---

**Description**

Transform a polynomial, find a greatest common factor, or determine the multiplicity of a root.

**Usage**

```
polytrans(p, q)
```

```
polygcf(p, q, tol = 1e-12)
```

**Arguments**

p, q                    vectors representing two polynomials.  
tol                    tolerance for coefficients to tolerate.

**Details**

Transforms polynomial p replacing occurrences of x with another polynomial q in x.

Finds a greatest common divisor (or factor) of two polynomials. Determines the multiplicity of a possible root; returns 0 if not a root. This is in general only true to a certain tolerance.

**Value**

polytrans and polygcf return vectors representing polynomials. rootsmult returns a natural number (or 0).

**Note**

There are no such functions in Matlab or Octave.

**See Also**

[polyval](#)

**Examples**

```
# (x+1)^2 + (x+1) + 1
polytrans(c(1, 1, 1), c(1, 1))    #=> 1 3 3
polytrans(c(1, 1, 1), c(-1, -1)) #=> 1 1 1

p <- c(1,-1,1,-1,1)            #=> x^4 - x^3 + x^2 - x + 1
q <- c(1,1,1)                   #=> x^2 + x + 1
polygcf(polyml(p, q), q)        #=> [1] 1 1 1

p = polypow(c(1, -1), 6)        #=> [1] 1 -6 15 -20 15 -6 1
rootsmult(p, 1)                #=> [1] 6
```

---

polyval, polyvalm      *Evaluating a Polynomial*

---

### Description

Evaluate polynomial on vector or matrix.

### Usage

```
polyval(p, x)
```

```
polyvalm(p, A)
```

### Arguments

p	vector representing a polynomial.
x	vector of values where to evaluate the polynomial.
A	matrix; needs to be square.

### Details

polyval evaluates the polynomial given by p at the values specified by the elements of x. If x is a matrix, the polynomial will be evaluated at each element and a matrix returned.

polyvalm will evaluate the polynomial in the matrix sense, i.e., matrix multiplication is used instead of element by element multiplication as used in 'polyval'. The argument matrix A must be a square matrix.

### Value

Vector of values, resp. a matrix.

### See Also

[poly](#), [roots](#)

### Examples

```
# Evaluate 3 x^2 + 2 x + 1 at x = 5, 7, and 9
p = c(3, 2, 1);
polyval(p, c(5, 7, 9))    # 86 162 262

# Apply the characteristic polynomial to its matrix
A <- pascal(4)
p <- pracma::Poly(A)    # characteristic polynomial of A
polyvalm(p, A)        # almost zero 4x4-matrix
```

---

pow2

*Base 2 Power*

---

### Description

Power with base 2.

### Usage

```
pow2(f, e)
```

### Arguments

f                    numeric vector of factors  
e                    numeric vector of exponents for base 2

### Details

Computes the expression  $f * 2^e$ , setting e to f and f to 1 in case e is missing. Complex values are only processed if e is missing.

### Value

Returns a numeric vector computing  $f 2^e$ .

### See Also

[nextpow2](#)

### Examples

```
pow2(c(0, 1, 2, 3))                    #=> 1 2 4 8  
pow2(c(0, -1, 2, 3), c(0,1,-2,3))    #=> 0.0 -2.0 0.5 24.0  
pow2(1i)                                #=> 0.7692389+0.6389613i
```

---

ppfit

*Piecewise Polynomial Fit*

---

### Description

Piecewise linear or cubic fitting.

### Usage

```
ppfit(x, y, xi, method = c("linear", "cubic"))
```

**Arguments**

<code>x, y</code>	x-, y-coordinates of given points.
<code>xi</code>	x-coordinates of the chosen support nodes.
<code>method</code>	interpolation method, can be 'constant', 'linear', or 'cubic' (i.e., 'spline').

**Details**

ppfit fits a piece-wise polynomial to the input independent and dependent variables, `x` and `y`, respectively. A weighted linear least squares solution is provided. The weighting vector `w` must be of the same size as the input variables.

**Value**

Returns a pp (i.e., piecewise polynomial) structure.

**Note**

Following an idea of Copyright (c) 2012 Ben Abbott, Martin Helm for Octave.

**See Also**

[mkpp](#), [ppval](#)

**Examples**

```
x <- 0:39
y <- c( 8.8500, 32.0775, 74.7375, 107.6775, 132.0975, 156.6675,
       169.0650, 187.5375, 202.2575, 198.0750, 225.9600, 204.3550,
       233.8125, 204.5925, 232.3625, 204.7550, 220.1925, 199.5875,
       197.3025, 175.3050, 218.6325, 163.0775, 170.6625, 148.2850,
       154.5950, 135.4050, 138.8600, 125.6750, 118.8450, 99.2675,
       129.1675, 91.1925, 89.7000, 76.8825, 83.6625, 74.1950,
       73.9125, 55.8750, 59.8675, 48.1900)

xi <- linspace(0, 39, 8)
pplin <- ppfit(x, y, xi) # method = "linear"
ppcub <- ppfit(x, y, xi, method = "cubic")

## Not run:
plot(x, y, type = "b", main = "Piecewise polynomial approximation")
xs <- linspace(0, 39, 100)
yslin <- ppval(pplin, xs)
yscub <- ppval(ppcub, xs)
lines(xs, yscub, col="red", lwd = 2)
lines(xs, yslin, col="blue")
grid()
## End(Not run)
```



**Description**

Make or evaluate a piecewise polynomial.

**Usage**

mkpp(x, P)

ppval(pp, xx)

**Arguments**

x	increasing vector of real numbers.
P	matrix containing the coefficients of polynomials in each row.
pp	a piecewise polynomial structure, generated by mkpp.
xx	numerical vector

**Details**

pp←mkpp(x,P) builds a piecewise polynomial from its breaks x and coefficients P. x is a monotonically increasing vector of length L+1, and P is an L-by-k matrix where each row contains the coefficients of the polynomial of order k, from highest to lowest exponent, on the interval [x[i],x[i+1]).

ppval(pp,xx) returns the values of the piecewise polynomial pp at the entries of the vector xx. The first and last polynomial will be extended to the left resp. right of the interval [x[1],x[L+1]).

**Value**

mkpp will return a piecewise polynomial structure, that is a list with components breaks=x, pieces=P, order=k and dim=1 for scalar-valued functions.

**Note**

Matlab allows to generate vector-valued piecewise polynomials. This may be included in later versions.

**See Also**

[cubicspline](#)

### Examples

```
## Example: Linear interpolation of the sine function
xs <- linspace(0, pi, 10)
ys <- sin(xs)
P <- matrix(NA, nrow = 9, ncol = 2)
for (i in 1:9) {
  P[i, ] <- c((ys[i+1]-ys[i])/(xs[i+1]-xs[i]), ys[i])
}
ppsin <- mkpp(xs, P)

## Not run:
plot(xs, ys); grid()
x100 <- linspace(0, pi, 100)
lines(x100, sin(x100), col="darkgray")
ypp <- ppval(ppsin, x100)
lines(x100, ypp, col="red")

## End(Not run)
```

---

primes

*Prime Numbers*

---

### Description

Generate a list of prime numbers less or equal  $n$ , resp. between  $n_1$  and  $n_2$ .

### Usage

```
primes(n)
```

### Arguments

$n$  nonnegative integer greater than 1.

### Details

The list of prime numbers up to  $n$  is generated using the "sieve of Erasthostenes". This approach is reasonably fast, but may require a lot of main memory when  $n$  is large.

In double precision arithmetic integers are represented exactly only up to  $2^{53} - 1$ , therefore this is the maximal allowed value.

### Value

vector of integers representing prime numbers

### See Also

[isprime](#), [factors](#)

**Examples**

```

primes(1000)
## Not run:
## Appendix: Logarithmic Integrals and Prime Numbers (C.F.Gauss, 1846)

library('gsl')
# 'European' form of the logarithmic integral
Li <- function(x) expint_Ei(log(x)) - expint_Ei(log(2))

# No. of primes and logarithmic integral for 10^i, i=1..12
i <- 1:12; N <- 10^i
# piN <- numeric(12)
# for (i in 1:12) piN[i] <- length(primes(10^i))
piN <- c(4, 25, 168, 1229, 9592, 78498, 664579,
        5761455, 50847534, 455052511, 4118054813, 37607912018)
cbind(i, piN, round(Li(N)), round((Li(N)-piN)/piN, 6))

# i      pi(10^i)      Li(10^i)  rel.err
# -----
# 1         4          5  0.280109
# 2        25         29  0.163239
# 3       168        177  0.050979
# 4      1229       1245  0.013094
# 5      9592       9629  0.003833
# 6     78498      78627  0.001637
# 7    664579     664917  0.000509
# 8   5761455    5762208  0.000131
# 9  50847534   50849234  0.000033
# 10 455052511  455055614  0.000007
# 11 4118054813 4118066400  0.000003
# 12 37607912018 37607950280  0.000001
# -----
## End(Not run)

```

---

procrustes

*Solving the Procrustes Problem*


---

**Description**

procrustes solves for two matrices A and B the 'Procrustes Problem' of finding an orthogonal matrix Q such that  $A - B \cdot Q$  has the minimal Frobenius norm.

kabsch determines a best rotation of a given vector set into a second vector set by minimizing the weighted sum of squared deviations. The order of vectors is assumed fixed.

**Usage**

```
procrustes(A, B)
```

```
kabsch(A, B, w = NULL)
```

**Arguments**

A, B                two numeric matrices of the same size.  
w                    weights , influence the distance of points

**Details**

The function `procrustes(A, B)` uses the `svd` decomposition to find an orthogonal matrix  $Q$  such that  $A - B * Q$  has a minimal Frobenius norm, where this norm for a matrix  $C$  is defined as  $\sqrt{\text{Trace}(t(C) * C)}$ , or  $\text{norm}(C, 'F')$  in R.

Solving it with  $B = I$  means finding a nearest orthogonal matrix.

`kabsch` solves a similar problem and uses the Procrustes procedure for its purpose. Given two sets of points, represented as columns of the matrices  $A$  and  $B$ , it determines an orthogonal matrix  $U$  and a translation vector  $R$  such that  $U * A + R - B$  is minimal.

**Value**

`procrustes` returns a list with components  $P$ , which is  $B * Q$ , then  $Q$ , the orthogonal matrix, and  $d$ , the Frobenius norm of  $A - B * Q$ .

`kabsch` returns a list with  $U$  the orthogonal matrix applied,  $R$  the translation vector, and  $d$  the least root mean square between  $U * A + R$  and  $B$ .

**Note**

The `kabsch` function does not take into account scaling of the sets, but this could easily be integrated.

**References**

Golub, G. H., and Ch. F. van Loan (1996). *Matrix Computations*. 3rd Edition, The John Hopkins University Press, Baltimore London. [Sect. 12.4, p. 601]

Kabsch, W. (1976). A solution for the best rotation to relate two sets of vectors. *Acta Cryst A*, Vol. 32, p. 9223.

**See Also**

[svd](#)

**Examples**

```
## Procrustes
U <- randortho(5)           # random orthogonal matrix
P <- procrustes(U, eye(5))

## Kabsch
P <- matrix(c(0, 1, 0, 0, 1, 1, 0, 1,
              0, 0, 1, 0, 1, 0, 1, 1,
              0, 0, 0, 1, 0, 1, 1, 1), nrow = 3, ncol = 8, byrow = TRUE)
R <- c(1, 1, 1)
```

```

phi <- pi/4
U <- matrix(c(1, 0, 0,
              0, cos(phi), -sin(phi),
              0, sin(phi), cos(phi)), nrow = 3, ncol = 3, byrow = TRUE)

Q <- U %*% P + R
K <- kabsch(P, Q)
# K$R == R and K$U %*% P + c(K$R) == Q

```

psi

*Psi (Polygamma) Function***Description**

Arbitrary order Polygamma function valid in the entire complex plane.

**Usage**

```
psi(k, z)
```

**Arguments**

**k** order of the polygamma function, whole number greater or equal 0.  
**z** numeric complex number or vector.

**Details**

Computes the Polygamma function of arbitrary order, and valid in the entire complex plane. The polygamma function is defined as

$$\psi(n, z) = \frac{d^{n+1}}{dz^{n+1}} \log(\Gamma(z))$$

If  $n$  is 0 or absent then  $\psi$  will be the Digamma function. If  $n=1, 2, 3, 4, 5$  etc. then  $\psi$  will be the tri-, tetra-, penta-, hexa-, hepta- etc. gamma function.

**Value**

Returns a complex number or a vector of complex numbers.

**Examples**

```

psi(2) - psi(1)      # 1
-psi(1)             # Eulers constant: 0.57721566490153 [or, -psi(0, 1)]
psi(1, 2)           # pi^2/6 - 1      : 0.64493406684823
psi(10, -11.5-0.577007813568142i)
# is near a root of the decagamma function

```

---

qpspecial, qpsolve      *Special Quadratic Programming Solver*

---

### Description

Solves a special Quadratic Programming problem.

### Usage

```
qpspecial(G, x, maxit = 100)
```

```
qpsolve(d, A, b, meq = 0, tol = 1e-07)
```

### Arguments

G	m x n-matrix.
x	column vector of length n, the initial (feasible) iterate; if not present (or requirements on x0 not met), x0 will be found.
maxit	maximum number of iterates allowed; default 100.
d	Linear term of the quadratic form.
A, b	Linear equality and inequality constraints.
meq	First meq rows are used as equality constraints.
tol	Tolerance used for stopping the iteration.

### Details

qpspecial solves the special QP problem:

$$\min q(x) = \|G*x\|_2^2 = x'*(G'*G)*x$$

s. t.  $\sum(x) = 1$   
and  $x \geq 0$

The problem corresponds to finding the smallest vector (2-norm) in the convex hull of the columns of G.

qpsolve solves the more general QP problem:

$$\min q(x) = 0.5 t(x)*x - d x$$

s. t.  $A x \geq b$

with  $A x = b$  for the first meq rows.

### Value

Returns a list with the following components:

- x – optimal point attaining optimal value;
- d = G\*x – smallest vector in the convex hull;
- q – optimal value found, = t(d) %\*% d;

- niter – number of iterations used;
- info – error number:
  - = 0: everything went well, q is optimal,
  - = 1: maxit reached and final x is feasible,
  - = 2: something went wrong.

### Note

x may be missing, same as if requirements are not met; may stop with an error if x is not feasible.

### Author(s)

Matlab code by Anders Skajaa, 2010, under GPL license (HANSO toolbox); converted to R by Abhirup Mallik and Hans W. Borchers, with permission.

### References

[Has to be found.]

### Examples

```
G <- matrix(c(0.31, 0.99, 0.54, 0.20,
             0.56, 0.97, 0.40, 0.38,
             0.81, 0.06, 0.44, 0.80), 3, 4, byrow =TRUE)

qpspecial(G)
# $x
#           [,1]
# [1,] 1.383697e-07
# [2,] 5.221698e-09
# [3,] 8.648168e-01
# [4,] 1.351831e-01
# $d
#           [,1]
# [1,] 0.4940377
# [2,] 0.3972964
# [3,] 0.4886660
# $q
# [1] 0.6407121
# $niter
# [1] 6
# $info
# [1] 0

# Example from quadprog::solve.QP
d <- c(0,5,0)
A <- matrix(c(-4, -3, 0, 2, 1, 0, 0, -2, 1), 3, 3)
b <- c(-8, 2, 0)
qpsolve(d, A, b)
## $sol
## [1] 0.4761905 1.0476190 2.0952381
## $val
## [1] -2.380952
```

```
## $niter  
## [1] 3
```

---

qrSolve

*LSE Solution*

---

### Description

Systems of linear equations via QR decomposition.

### Usage

```
qrSolve(A, b)
```

### Arguments

A numerical matrix with  $nrow(A) \geq ncol(A)$ .  
b numerical vector with  $length(b) == nrow(A)$ .

### Details

Solves (overdetermined) systems of linear equations via QR decomposition.

### Value

The solution of the system as vector.

### References

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Society for Industrial and Applied Mathematics, Philadelphia.

### See Also

[householder](#)

### Examples

```
A <- matrix(c(0,-4,2, 6,-3,-2, 8,1,-1), 3, 3, byrow=TRUE)  
b <- c(-2, -6, 7)  
qrSolve(A, b)  
  
## Solve an overdetermined linear system of equations  
A <- matrix(c(1:8,7,4,2,3,4,2,2), ncol=3, byrow=TRUE)  
b <- rep(6, 5)  
x <- qrSolve(A, b)  
qr.solve(A, rep(6, 5)); x
```



---

quad *Adaptive Simpson Quadrature*

---

**Description**

Adaptive quadrature of functions of one variable over a finite interval.

**Usage**

```
quad(f, xa, xb, tol = .Machine$double.eps^0.5, trace = FALSE, ...)
```

**Arguments**

f	a one-dimensional function; needs to be vectorized.
xa	lower limit of integration; must be finite
xb	upper limit of integration; must be finite
tol	accuracy requested.
trace	logical; shall a trace be printed?
...	additional arguments to be passed to f.

**Details**

Realizes adaptive Simpson quadrature in R through recursive calls.

The function *f* needs to be vectorized though this could be changed easily. *quad* is not suitable for functions with singularities in the interval or at end points.

**Value**

A single numeric value, the computed integral.

**Note**

More modern adaptive methods based on Gauss-Kronrod or Clenshaw-Curtis quadrature are now generally preferred.

**Author(s)**

Copyright (c) 1998 Walter Gautschi for the Matlab version published as part of the referenced article. R implementation by Hans W Borchers 2011.

**References**

Gander, W. and W. Gautschi (2000). "Adaptive Quadrature — Revisited". BIT, Vol. 40, 2000, pp. 84-101.

**See Also**

[integrate](#), [quad1](#)

**Examples**

```
# options(digits=15)
f <- function(x) x * cos(0.1*exp(x)) * sin(0.1*pi*exp(x))
quad(f, 0, 4) # 1.2821290747821
quad(f, 0, 4, tol=10^-15) # 1.2821290743501
integrate(f, 0, 4)
# 1.28212907435010 with absolute error < 4.1e-06

## Not run:
xx <- seq(0, 4, length.out = 200)
yy <- f(xx)
plot(xx, yy, type = 'l')
grid()
## End(Not run)
```

---

quad2d

*2-d Gaussian Quadrature*


---

**Description**

Two-dimensional Gaussian Quadrature.

**Usage**

```
quad2d(f, xa, xb, ya, yb, n = 32, ...)
```

**Arguments**

f	function of two variables; needs to be vectorized.
xa, ya	lower limits of integration; must be finite.
xb, yb	upper limits of integration; must be finite.
n	number of nodes used per direction.
...	additional arguments to be passed to f.

**Details**

Extends the Gaussian quadrature to two dimensions by computing two sets of nodes and weights (in x- and y-direction), evaluating the function on this grid and multiplying weights appropriately.

The function f needs to be vectorized in both variables such that f(X, Y) returns a matrix when X and Y are matrices (of the same size).

quad is not suitable for functions with singularities.

**Value**

A single numerical value, the computed integral.

**Note**

The extension of Gaussian quadrature to two dimensions is obvious, but see also the example ‘integral2d.m’ at Nick Trefethens “10 digits 1 page”.

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[quad](#), `cubature::adaptIntegrate`

**Examples**

```
## Example: f(x, y) = (y+1)*exp(x)*sin(16*y-4*(x+1)^2)
f <- function(x, y)
  (y+1) * exp(x) * sin(16*y-4*(x+1)^2)
# this is even faster than cubature::adaptIntegral():
quad2d(f, -1, 1, -1, 1)
# 0.0179515583236958 # true value 0.01795155832370

## Volume of the sphere: use polar coordinates
f0 <- function(x, y) sqrt(1 - x^2 - y^2) # for x^2 + y^2 <= 1
fp <- function(x, y) y * f0(y*cos(x), y*sin(x))
quad2d(fp, 0, 2*pi, 0, 1, n = 101) # 2.09439597740074
2/3 * pi # 2.0943951023932
```

---

quadcc

*Adaptive Clenshaw-Curtis Quadrature*


---

**Description**

Adaptive Clenshaw-Curtis Quadrature.

**Usage**

```
quadcc(f, a, b, tol = .Machine$double.eps^0.5, ...)
```

**Arguments**

f	integrand as function, may have singularities at the endpoints.
a, b	endpoints of the integration interval.
tol	relative tolerance.
...	Additional parameters to be passed to the function f.

**Details**

Adaptive version of the Clenshaw-Curtis quadrature formula with an (4, 8)-point error term.

**Value**

List with two components, value the value of the integral and the relative error error.

**See Also**

clenshaw\_curtis

**Examples**

```
## Not run:
## Dilogarithm function
flog <- function(t) log(1-t)/t
quadcc(flog, 1, 0, tol = 1e-12)
# 1.644934066848128 - pi^2/6 < 1e-13

## End(Not run)
```

---

quadgk

*Adaptive Gauss-Kronrod Quadrature*

---

**Description**

Adaptive Gauss-Kronrod Quadrature.

**Usage**

```
quadgk(f, a, b, tol = .Machine$double.eps^0.5, ...)
```

**Arguments**

f	integrand as function; needs to be vectorized, but may have singularities at the endpoints.
a, b	endpoints of the integration interval.
tol	relative tolerance.
...	Additional parameters to be passed to the function f.

**Details**

Adaptive version of the (7, 15)-point Gauss-Kronrod quadrature formula, where in each recursion the error is taken as the difference between these two estimated integrals.

The function f must be vectorized, though this will not be checked and may lead to strange errors. If it is not, use `F = Vectorize(f)`.

**Value**

Value of the integration. The relative error should be of the same order of magnitude as the relative tolerance (or much smaller).

**Note**

Uses the same nodes and weights as the quadQK15 procedure in the QUADPACK library.

**See Also**

gauss\_kronrod

**Examples**

```
## Dilogarithm function
flog <- function(t) log(1-t)/t
quadgk(flog, 1, 0, tol = 1e-12)
# 1.644934066848128 - pi^2/6 < 1e-13
```

---

quadgr

*Gaussian Quadrature with Richardson Extrapolation*


---

**Description**

Gaussian 12-point quadrature with Richardson extrapolation.

**Usage**

```
quadgr(f, a, b, tol = .Machine$double.eps^(1/2), ...)
```

**Arguments**

f	integrand as function, may have singularities at the endpoints.
a, b	endpoints of the integration interval.
tol	relative tolerance.
...	Additional parameters to be passed to the function f.

**Details**

quadgr uses a 12-point Gauss-Legendre quadrature. The error estimate is based on successive interval bisection. Richardson extrapolation accelerates the convergence for some integrals, especially integrals with endpoint singularities.

Through some preprocessing infinite intervals can also be handled.

**Value**

List with value and rel.err.

**Author(s)**

Copyright (c) 2009 Jonas Lundgren for the Matlab function quadgr available on MatlabCentral under the BSD license.

R re-implementation by HwB, email: <hwborchers@googlemail.com>, in 2011.

**See Also**

gaussLegendre

**Examples**

```
## Dilogarithm function
flog <- function(t) log(1-t)/t
quadgr(flog, 1, 0, tol = 1e-12)
# value
# 1.6449340668482 , is pi^2/6 = 1.64493406684823
# rel.err
# 2.07167616395054e-13
```

---

quadinf

*Infinite Integrals*


---

**Description**

Iterative quadrature of functions over finite, semifinite, or infinite intervals.

**Usage**

```
quadinf(f, xa, xb, tol = 1e-12, ...)
```

**Arguments**

f	univariate function; needs not be vectorized.
xa	lower limit of integration; can be infinite
xb	upper limit of integration; can be infinite
tol	accuracy requested.
...	additional arguments to be passed to f.

**Details**

quadinf implements the ‘double exponential method’ for fast numerical integration of smooth real functions on finite intervals. For infinite intervals, the tanh-sinh quadrature scheme is applied, that is the transformation  $g(t) = \tanh(\pi/2 * \sinh(t))$ .

Please note that this algorithm does work very accurately for ‘normal’ function, but should not be applied to (heavily) oscillating functions. The maximal number of iterations is 7, so if this is returned the iteration may not have converged.

The integrand function needs *not* be vectorized.

**Value**

A list with components `Q` the integral value, `relerr` the relative error, and `niter` the number of iterations.

**Note**

See also my remarks on R-help in September 2010 in the thread “bivariate vector numerical integration with infinite range”.

**References**

D. H. Bayley. Tanh-Sinh High-precision Quadrature. 2006.  
URL: <https://www.davidhbailey.com/dhbpapers/dhb-tanh-sinh.pdf>

**See Also**

[integrate](#), [quadgk](#)

**Examples**

```
## We will look at the error function exp(-x^2)
f <- function(x) exp(-x^2)          # sqrt(pi)/2          theory
quadinf(f, 0, Inf)                  # 0.8862269254527413
quadinf(f, -Inf, 0)                 # 0.8862269254527413

f = function(x) sqrt(x) * exp(-x)  # 0.8862269254527579 exact
quadinf(f, 0, Inf)                  # 0.8862269254527579

f = function(x) x * exp(-x^2)      # 1/2
quadinf(f, 0, Inf)                  # 0.5

f = function(x) 1 / (1+x^2)        # 3.141592653589793 = pi
quadinf(f, -Inf, Inf)               # 3.141592653589784
```

---

quadl

*Adaptive Lobatto Quadrature*


---

**Description**

Adaptive quadrature of functions of one variable over a finite interval.

**Usage**

```
quadl(f, xa, xb, tol = .Machine$double.eps^0.5, trace = FALSE, ...)
```

**Arguments**

f	a one-dimensional function; needs to be vectorized.
xa	lower limit of integration; must be finite
xb	upper limit of integration; must be finite
tol	accuracy requested.
trace	logical; shall a trace be printed?
...	additional arguments to be passed to f.

**Details**

Realizes adaptive Lobatto quadrature in R through recursive calls.

The function f needs to be vectorized though this could be changed easily.

**Value**

A single numeric value, the computed integral.

**Note**

Compared to Gaussian quadrature, Lobatto integration include the end points of the integration interval. It is accurate for polynomials up to degree  $2n-3$ , where n is the number of integration points.

**Author(s)**

Copyright (c) 1998 Walter Gautschi for the Matlab version published as part of the referenced article. R implementation by Hans W Borchers 2011.

**References**

Gander, W. and W. Gautschi (2000). "Adaptive Quadrature — Revisited". BIT, Vol. 40, 2000, pp. 84-101.

**See Also**

[quad](#)

**Examples**

```
# options(digits=15)
f <- function(x) x * cos(0.1*exp(x)) * sin(0.1*pi*exp(x))
quadl(f, 0, 4)          # 1.2821290743501
integrate(f, 0, 4)
# 1.28212907435010 with absolute error < 4.1e-06

## Not run:
xx <- seq(0, 4, length.out = 200)
yy <- f(xx)
```



```

plot(xx, yy, type = 'l')
grid()
## End(Not run)

```

---

quadprog

*Quadratic Programming*


---

### Description

Solves quadratic programming problems with linear and box constraints.

### Usage

```

quadprog(C, d, A = NULL, b = NULL,
         Aeq = NULL, beq = NULL, lb = NULL, ub = NULL)

```

### Arguments

C	symmetric matrix, representing the quadratic term.
d	vector, representing the linear term.
A	matrix, represents the linear constraint coefficients.
b	vector, constant vector in the constraints.
Aeq	matrix, linear equality constraint coefficients.
beq	vector, constant equality constraint vector.
lb	elementwise lower bounds.
ub	elementwise upper bounds.

### Details

Finds a minimum for the quadratic programming problem specified as:

$$\min 1/2x'Cx + d'x$$

such that the following constraints are satisfied:

$$Ax \leq b$$

$$Aeqx = beq$$

$$lb \leq x \leq ub$$

The matrix should be symmetric and positive definite, in which case the solution is unique, indicated when the exit flag is 1.

For more information, see ?solve.QP.

**Value**

Returns a list with components

xmin	minimum solution, subject to all bounds and constraints.
fval	value of the target expression at the arg minimum.
eflag	exit flag.

**Note**

This function is wrapping the active set quadratic solver in the quadprog package: `quadprog::solve.QP`, combined with a more MATLAB-like API interface.

**References**

Nocedal, J., and St. J. Wright (2006). Numerical Optimization. Second Edition, Springer Series in Operations Research, New York.

**See Also**

[lsqlincon](#), `quadprog::solve.QP`

**Examples**

```
## Example in ?solve.QP
# Assume we want to minimize: 1/2 x^T x - (0 5 0) %*% x
# under the constraints:      A x <= b
# with b = (8,-2, 0)
# and      ( 4 3 0)
#      A = (-2 -1 0)
#           ( 0 2,-1)
# and possibly equality constraint 3x1 + 2x2 + x3 = 1
# or upper bound c(1.5, 1.5, 1.5).

C <- diag(1, 3); d <- -c(0, 5, 0)
A <- matrix(c(4,3,0, -2,-1,0, 0,2,-1), 3, 3, byrow=TRUE)
b <- c(8, -2, 0)

quadprog(C, d, A, b)
# $xmin
# [1] 0.4761905 1.0476190 2.0952381
# $fval
# [1] -2.380952
# $eflag
# [1] 1

Aeq <- c(3, 2, 1); beq <- 1
quadprog(C, d, A, b, Aeq, beq)
# $xmin
# [1] 1.4 -0.8 -1.6
# $fval
# [1] 6.58
```

```

# $eflag
# [1] 1

quadprog(C, d, A, b, lb = 0, ub = 1.5)
# $xmin
# [1] 0.625 0.750 1.500
# $fval
# [1] -2.148438
# $eflag
# [1] 1

## Example help(quadprog)
C <- matrix(c(1, -1, -1, 2), 2, 2)
d <- c(-2, -6)
A <- matrix(c(1,1, -1,2, 2,1), 3, 2, byrow=TRUE)
b <- c(2, 2, 3)
lb <- c(0, 0)

quadprog(C, d, A, b, lb=lb)
# $xmin
# [1] 0.6666667 1.3333333
# $fval
# [1] -8.222222
# $eflag
# [1] 1

```

---

quadv

*Vectorized Integration*


---

## Description

Vectorized adaptive Simpson integration.

## Usage

```
quadv(f, a, b, tol = .Machine$double.eps^(1/2), ...)
```

## Arguments

f	univariate, vector-valued function; need not be vectorized.
a, b	endpoints of the integration interval.
tol	accuracy required for the recursion step.
...	further parameters to be passed to the function f.

## Details

Recursive version of the adaptive Simpson quadrature, recursion is based on the maximum of all components of the function calls.

quad is not suitable for functions with singularities in the interval or at end points.

**Value**

Returns a list with components `Q` the integral value, `fcnt` the number of function calls, and `estim.prec` the estimated precision that normally will be much too high.

**See Also**

[quad](#)

**Examples**

```
## Examples
f1 <- function(x) c(sin(x), cos(x))
quadv(f1, 0, pi)
# $Q
# [1] 2.000000e+00 1.110223e-16
# $fcnt
# [1] 65
# $estim.prec
# [1] 4.321337e-07

f2 <- function(x) x^c(1:10)
quadv(f2, 0, 1, tol = 1e-12)
# $Q
# [1] 0.50000000 0.33333333 0.25000000 0.20000000 0.16666667
# [6] 0.14285714 0.12500000 0.11111111 0.10000000 0.09090909
# $fcnt
# [1] 505
# $estim.prec
# [1] 2.49e-10
```

---

quiver

*Quiver or Velocity Plot*

---

**Description**

A quiver plot displays velocity vectors as arrows with components  $(u, v)$  at the points  $(x, y)$ .

**Usage**

```
quiver(x, y, u, v,
       scale = 0.05, angle = 10, length = 0.1, ...)
```

**Arguments**

<code>x, y</code>	<code>x, y</code> -coordinates of start points of the arrows.
<code>u, v</code>	<code>x, y</code> -coordinates of start points.
<code>scale</code>	scales the length of the arrows.
<code>angle</code>	angle between shaft and edge of the arrows.

length            length of the arrow edges.  
 ...                more options presented to the arrows primitive.

### Details

The matrices  $x$ ,  $y$ ,  $u$ ,  $v$  must all be the same size and contain corresponding position and velocity components. However,  $x$  and  $y$  can also be vectors.

### Value

Opens a graph window and plots the velocity vectors.

### See Also

[vectorfield](#), [arrows](#)

---

rand	<i>Create Random Matrices</i>
------	-------------------------------

---

### Description

Create random matrices or random points in a unit circle (Matlab style).

### Usage

```
rand(n = 1, m = n)
randn(n = 1, m = n)
randi(imax, n = 1, m = n)
randsample(n, k, w = NULL, replacement = FALSE)
```

```
rands(n = 1, N = 1, r = 1)
randp(n = 1, r = 1)
```

### Arguments

n, m	integers specifying the size of the matrix
imax	integer or pair of integers
k	number of elements to return.
w	weight vector, used for discrete probabilities.
replacement	logical; sampling with or without replacement.
N	dimension of a sphere, N=1 for the unit circle
r	radius of circle, default 1.

**Details**

rand(), randn(), randi() create random matrices of size  $n \times m$ , where the default is square matrices if  $m$  is missing.

rand() uses the uniform distribution on  $]0, 1[$ , while randn() uses the normal distribution with mean 0 and standard deviation 1.

randi() generates integers between `imax[1]` and `imax[2]` resp. 1 and `imax`, if `imax` is a scalar.

randsample() samples  $k$  elements from  $1:n$ , with or without replacement, or returns a weighted sample (with replacement), using the weight vector  $w$  for probabilities.

rands() generates uniformly random points on an  $N$ -sphere in the  $N+1$ -dimensional space. To generate uniformly random points in the  $N$ -dim. unit cube, take points in  $S^{N-1}$  and multiply with  $\text{unif}(n)^{(1/(N-1))}$ .

randp() generates uniformly random points in the unit circle (or in a circle of radius  $r$ ).

**Value**

Matrices of size  $n \times m$  resp. a vector of length  $n$ .

randp() returns a pair of values representing a point in the circle, or a matrix of size  $(n, 2)$ .

rands() returns a matrix of size  $(n, N+1)$  with all rows being vectors of length 1.

**Note**

The Matlab style of setting a seed is not available; use R style `set.seed(...)`.

**References**

Knuth, D. (1981). The Art of Computer programming; Vol. 2: Seminumerical Algorithms; Chapt. 3: Random Numbers. Addison-Wesley, Reading.

**See Also**

[set.seed](#)

**Examples**

```
rand(3)
randn(1, 5)
randi(c(1,6), 1, 10)
randsample(10, 5, replacement = TRUE, w = c(0,0,0, 1, 1, 1, 1, 0,0,0))
```

```
P <- rands(1000, N = 1, r = 2)
U <- randp(1000, 2)
## Not run:
plot(U[, 1], U[, 2], pch = "+", asp = 1)
points(P, pch = ".")
## End(Not run)
```

```
#-- v is 2 independent normally distributed elements
# u <- randp(1); r <- t(u) %*% u
```

```
# v <- sqrt(-2 * log(r)/r) * u

n <- 5000; U <- randp(n)
R <- apply(U*U, 1, sum)
P <- sqrt(-2 * log(R)/R) * U # rnorm(2*n)
## Not run:
hist(c(P))
## End(Not run)
```

---

randcomb

*Random Combination*

---

### Description

Generates a random combination.

### Usage

```
randcomb(a, m)
```

### Arguments

a	numeric vector of some length n
m	integer with $0 \leq m \leq n$

### Details

Generates one random combination of the elements a of length m.

### Value

vector of combined elements of a

### Note

This behavior is different from Matlab/Octave, but does better correspond with the behavior of the perms() function.

### See Also

[combs](#), [randperm](#)

### Examples

```
randcomb(seq(2, 10, by=2), m = 3)
```

---

 randortho

 Generate Random Orthonormal or Unitary Matrix
 

---

### Description

Generates random orthonormal or unitary matrix of size  $n$ .

Will be needed in applications that explore high-dimensional data spaces, for example optimization procedures or Monte Carlo methods.

### Usage

```
randortho(n, type = c("orthonormal", "unitary"))
```

### Arguments

<code>n</code>	positive integer.
<code>type</code>	orthonormal (i.e., real) or unitary (i.e., complex) matrix.

### Details

Generates orthonormal or unitary matrices  $Q$ , that is  $t(Q)$  resp  $t(\text{Conj}(Q))$  is inverse to  $Q$ . The randomness is meant with respect to the (additively invariant) Haar measure on  $O(n)$  resp.  $U(n)$ .

Stewart (1980) describes a way to generate such matrices by applying Householder transformation. Here a simpler approach is taken based on the QR decomposition, see Mezzadri (2006),

### Value

Orthogonal (or unitary) matrix  $Q$  of size  $n$ , that is  $Q \%*\% t(Q)$  resp.  $Q \%*\% t(\text{Conj}(Q))$  is the unit matrix of size  $n$ .

### Note

`rortho` was deprecated and eventually removed in version 2.1.7.

### References

G. W. Stewart (1980). "The Efficient Generation of Random Orthogonal Matrices with an Application to Condition Estimators". *SIAM Journal on Numerical Analysis*, Vol. 17, No. 3, pp. 403-409.

F. Mezzadri (2006). "How to generate random matrices from the classical compact groups". *NOTICES of the AMS*, Vol. 54 (2007), 592-604. ([arxiv.org/abs/math-ph/0609050v2](http://arxiv.org/abs/math-ph/0609050v2))

### Examples

```
Q <- randortho(5)
zapsmall(Q \%*\% t(Q))
zapsmall(t(Q) \%*\% Q)
```



---

randperm	<i>Random Permutation</i>
----------	---------------------------

---

**Description**

Generates a random permutation.

**Usage**

```
randperm(a, k)
```

**Arguments**

a	integer or numeric vector of some length n.
k	integer, smaller as a or length(a).

**Details**

Generates one random permutation of k of the elements a, if a is a vector, or of 1 : a if a is a single integer.

**Value**

Vector of permuted elements of a or 1 : a.

**Note**

This behavior is different from Matlab/Octave, but does better correspond with the behavior of the perms() function.

**See Also**

[perms](#)

**Examples**

```
randperm(1:6, 3)
randperm(6, 6)
randperm(11:20, 5)
randperm(seq(2, 10, by=2))
```

---

Rank

*Matrix Rank*

---

### Description

Provides an estimate of the rank of a matrix M.

### Usage

Rank(M)

### Arguments

M                      Numeric matrix; vectors will be considered as column vectors.

### Details

Provides an estimate of the number of linearly independent rows or columns of a matrix M. Compares an approach using QR-decomposition with one counting singular values larger than a certain tolerance (Matlab).

### Value

Matrix rank as integer between 0 and  $\min(\text{ncol}(M), \text{nrow}(M))$ .

### Note

The corresponding function in Matlab is called rank, but that term has a different meaning in R.

### References

Trefethen, L. N., and D. Bau III. (1997). Numerical Linear Algebra. SIAM, Philadelphia.

### See Also

[nullspace](#)

### Examples

```
Rank(magic(10))    #=> 7
Rank(magic(100))  #=> 3 (!)
Rank(hilb(8))     #=> 8 , but qr(hilb(8))$rank says, rank is 7.
# Warning message:
# In Rank(hilb(8)) : Rank calculation may be problematic.
```

---

rat *Continuous Fractions (Matlab Style)*

---

**Description**

Generate continuous fractions for numeric values.

**Usage**

```
rat(x, tol = 1e-06)
rats(x, tol = 1e-06)
```

**Arguments**

x                    a numeric scalar or vector.  
tol                  tolerance; default 1e-6 to make a nicer appearance for pi.

**Details**

rat generates continuous fractions, while rats prints the the corresponding rational representation and returns the numeric values.

**Value**

rat returns a character vector of string representations of continuous fractions in the format [b0; b1, ..., b\_{n-1}].

rats prints the rational number and returns a numeric vector.

**Note**

Essentially, these functions apply `contfrac`.

**See Also**

`numbers::contfrac`

**Examples**

```
rat(pi)
rats(pi)
rat(sqrt(c(2, 3, 5)), tol = 1e-15)
rats(sqrt(c(2, 3, 5)), tol = 1e-15)
```

---

ratinterp	<i>Rational Interpolation</i>
-----------	-------------------------------

---

**Description**

Burlisch-Stoer rational interpolation.

**Usage**

```
ratinterp(x, y, xs = x)
```

**Arguments**

x	numeric vector; points on the x-axis; needs to be sorted; at least three points required.
y	numeric vector; values of the assumed underlying function; x and y must be of the same length.
xs	numeric vector; points at which to compute the interpolation; all points must lie between $\min(x)$ and $\max(x)$ .

**Details**

The Burlisch-Stoer approach to rational interpolation is a recursive procedure (similar to the Newton form of polynomial interpolation) that produces a “diagonal” rational function, that is the degree of the numerator is either the same or one less than the degree of the denominator.

Polynomial interpolation will have difficulties if some kind of singularity exists in the neighborhood, even if the pole occurs in the complex plane. For instance, Runge’s function has a pole at  $z = 0.2i$ , quite close to the interval  $[-1, 1]$ .

**Value**

Numeric vector representing values at points xs.

**Note**

The algorithm does not yield a simple algebraic expression for the rational function found.

**References**

Stoer, J., and R. Bulirsch (2002). Introduction to Numerical Analysis. Third Edition, Springer-Verlag, New York.

Fausett, L. V. (2008). Applied Numerical Analysis Using Matlab. Second Edition, Pearson Education.

**See Also**

[rationalfit](#), [pade](#)

**Examples**

```
## Rational interpolation of Runge's function
x <- c(-1, -0.5, 0, 0.5, 1.0)
y <- runge(x)
xs <- linspace(-1, 1)
ys <- runge(xs)
yy <- ratinterp(x, y, xs) # returns exactly the Runge function

## Not run:
plot(xs, ys, type="l", col="blue", lty = 2, lwd = 3)
points(x, y)
yy <- ratinterp(x, y, xs)
lines(xs, yy, col="red")
grid()
## End(Not run)
```

---

rationalfit

*Rational Function Approximation*


---

**Description**

Fitting a rational function to data points.

**Usage**

```
rationalfit(x, y, d1 = 5, d2 = 5)
```

**Arguments**

x	numeric vector; points on the x-axis; needs to be sorted; at least three points required.
y	numeric vector; values of the assumed underlying function; x and y must be of the same length.
d1, d2	maximal degrees of numerator (d1) and denominator (d1) of the requested rational function.

**Details**

A rational fit is a rational function of two polynomials  $p_1$  and  $p_2$  (of user specified degrees  $d_1$  and  $d_2$ ) such that  $p_1(x)/p_2(x)$  approximates  $y$  in a least squares sense.

$d_1$  and  $d_2$  must be large enough to get a good fit and usually  $d_1=d_2$  gives good results

**Value**

List with components  $p_1$  and  $p_2$  for the polynomials in numerator and denominator of the rational function.

**Note**

This implementation will later be replaced by a ‘barycentric rational interpolation’.

**Author(s)**

Copyright (c) 2006 by Paul Godfrey for a Matlab version available from the MatlabCentral under BSD license. R re-implementation by Hans W Borchers.

**References**

Press, W. H., S. A. Teukolsky, W. T Vetterling, and B. P. Flannery (2007). Numerical Recipes: The Art of Numerical Computing. Third Edition, Cambridge University Press, New York.

**See Also**

[ratinterp](#)

**Examples**

```
## Not run:
x <- linspace(0, 15, 151); y <- sin(x)/x
rA <- rationalfit(x, y, 10, 10); p1 <- rA$p1; p2 <- rA$p2
ys <- polyval(p1,x) / polyval(p2,x)
plot(x, y, type="l", col="blue", ylim=c(-0.5, 1.0))
points(x, Re(ys), col="red") # max(abs(y-ys), na.rm=TRUE) < 1e-6
grid()

# Rational approximation of the Zeta function
x <- seq(-5, 5, by = 1/16)
y <- zeta(x)
rA <- rationalfit(x, y, 10, 10); p1 <- rA$p1; p2 <- rA$p2
ys <- polyval(p1,x) / polyval(p2,x)
plot(x, y, type="l", col="blue", ylim=c(-5, 5))
points(x, Re(ys), col="red")
grid()

# Rational approximation to the Gamma function
x <- seq(-5, 5, by = 1/32); y <- gamma(x)
rA <- rationalfit(x, y, 10, 10); p1 <- rA$p1; p2 <- rA$p2
ys <- polyval(p1,x) / polyval(p2,x)
plot(x, y, type="l", col = "blue")
points(x, Re(ys), col="red")
grid()
## End(Not run)
```

---

rectint	<i>Rectangle Intersection Areas</i>
---------	-------------------------------------

---

**Description**

Calculates the area of intersection of rectangles, specified by position vectors  $x$  and  $y$ .

**Usage**

```
rectint(x, y)
```

**Arguments**

$x, y$  both vectors of length 4, or both matrices with 4 columns.

**Details**

Rectangles are specified as position vectors, that is  $c(x[1], x[2])$  is the lower left corner,  $x[3]$  and  $x[4]$  are width and height of the rectangle. When  $x$  and  $y$  are matrices, each row is assumed to be a position vector specifying a rectangle.

**Value**

Returns a scalar if  $x$  and  $y$  are vectors. If  $x$  is a  $n$ -by-4 and  $y$  a  $m$ -by-4 matrix, then it returns a  $n$ -by- $m$  matrix  $R$  with entry  $(i, j)$  being the area  $rectint(x[i, ], y[j, ])$ .

**See Also**

[polyarea](#)

**Examples**

```
x <- c(0.5, 0.5, 0.25, 1.00)
y <- c(0.3, 0.3, 0.35, 0.75)
rectint(x, y)
# [1] 0.0825
```

---

refindall *Find overlapping regular expression matches.*

---

### Description

Find overlapping matches for a regular expression.

### Usage

```
refindall(s, pat, over = 1, ignorecase = FALSE)
```

### Arguments

s	Single character string.
pat	Regular expression.
over	Natural number, indication how many steps to go forward after a match; defaults to 1.
ignorecase	logical, whether to ignore case.

### Details

Returns the starting position of all — even overlapping — matches of the regular expression pat in the character string s.

The syntax for pattern matching has to be PERL-like.

### Value

A numeric vector with the indices of starting positions of all matches.

### Note

This effect can also be reached with the R function `gregexpr()`, see the example below.

### See Also

[regexp](#)

### Examples

```
refindall("ababababa", 'aba')
gregexpr('a(?=ba)', "ababababa", perl=TRUE)

refindall("AbababaBa", 'aba')
refindall("AbababaBa", 'aba', ignorecase = TRUE)
```



---

regexp	<i>Match regular expression</i>
--------	---------------------------------

---

**Description**

Returns the positions of substrings that match the regular expression.

**Usage**

```
regexp(s, pat, ignorecase = FALSE, once = FALSE, split = FALSE)
```

```
regexpi(s, pat, once = FALSE, split = FALSE)
```

**Arguments**

<code>s</code>	Character string, i.e. of length 1.
<code>pat</code>	Matching pattern as character string.
<code>ignorecase</code>	Logical: whether case should be ignored; default: FALSE.
<code>once</code>	Logical: whether the first are all occurrences should be found; default: all.
<code>split</code>	Logical: should the string be splitted at the occurrences of the pattern?; default: no.

**Details**

Returns the start and end positions and the exact value of substrings that match the regular expression. If `split` is choosen, the splitted strings will also be returned.

**Value**

A list with components `start` and `end` as numeric vectors indicating the start and end positions of the matches.

`match` contains each exact match, and `split` contains the character vector of splitted strings.

If no match is found all components will be NULL, except `split` that will contain the whole string if `split = TRUE`.

**Note**

This is the behavior of the corresponding Matlab function, though the signature, options and return values do not match exactly. Notice the transposed parameters `s` and `pat` compared to the corresponding R function `regexpr`.

**See Also**

[regexpr](#)

## Examples

```
s <- "bat cat can car COAT court cut ct CAT-scan"  
pat <- 'c[aeiou]+t'  
regex(s, pat)  
regexpi(s, pat)
```

---

regexprep

*Replace string using regular expression*

---

## Description

Replace string using regular expression.

## Usage

```
regexprep(s, expr, repstr, ignorecase = FALSE, once = FALSE)
```

## Arguments

s	Single character string.
expr	Regular expression to be matched.
repstr	String that replaces the matched substring(s).
ignorecase	logical, whether to ignore case.
once	logical, shall only the first or all occurrences be replaced.

## Details

Matches the regular expression against the string and replaces the first or all non-overlapping occurrences with the replacement string.

The syntax for regular expression has to be PERL-like.

## Value

String with substrings replaced.

## Note

The Matlab/Octave variant allows a character vector. This is not possible here as it would make the return value quite complicated.

## See Also

[gsub](#)

**Examples**

```
s <- "bat cat can car COAT court cut ct CAT-scan"
pat <- 'c[aeiou]+t'
regexprep(s, pat, '---')
regexprep(s, pat, '---', once = TRUE)
regexprep(s, pat, '---', ignorecase = TRUE)
```

---

repmat

*Replicate Matrix*

---

**Description**

Replicate and tile matrix.

**Usage**

```
repmat(a, n, m = n)
```

**Arguments**

a                    vector or matrix to be replicated.  
n, m                 number of times to replicate in each dimension.

**Details**

repmat(a,m,n) creates a large matrix consisting of an m-by-n tiling of copies of a.

**Value**

Returns matrix with value a replicated to the number of times in each dimension specified. Defaults to square if dimension argument resolves to a single value.

**See Also**

[Reshape](#)

**Examples**

```
repmat(1, 3)                                 # same as ones(3)
repmat(1, 3, 3)
repmat(matrix(1:4, 2, 2), 3)
```

---

Reshape

*Reshape Matrix*

---

### Description

Reshape matrix or vector.

### Usage

```
Reshape(a, n, m)
```

### Arguments

a	matrix or vector
n, m	size of the result

### Details

Reshape(a, n, m) returns the n-by-m matrix whose elements are taken column-wise from a.

An error results if a does not have n\*m elements. If m is missing, it will be calculated from n and the size of a.

### Value

Returns matrix (or array) of the requested size containing the elements of a.

### Examples

```
a <- matrix(1:12, nrow=4, ncol=3)
Reshape(a, 6, 2)
Reshape(a, 6)      # the same
Reshape(a, 3, 4)
```

---

ridders

*Ridders' Root Finding Method*

---

### Description

Ridders' root finding method is a powerful variant of 'regula falsi' (and 'false position'). In reliability and speed, this method is competitive with Brent-Dekker and similar approaches.

### Usage

```
ridders(fun, a, b, maxiter = 500, tol = 1e-12, ...)
```

**Arguments**

fun	function whose root is to be found.
a, b	left and right interval bounds.
maxiter	maximum number of iterations (function calls).
tol	tolerance, length of the last interval.
...	additional parameters passed on to the function.

**Details**

Given a bracketing interval  $[x_1, x_2]$ , the method first calculates the midpoint  $x_3 = (x_1 + x_2)/2$  and then uses an updating formula

$$x_4 = x_3 + (x_3 - x_1) \frac{\text{sgn}(f(x_1) - f(x_2))f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}$$

**Value**

Returns a list with components

root	root of the function.
f.root	value of the function at the found root.
niter	number of iterations, or more specifically: number of function calls.
estim.prec	the estimated precision, coming from the last brackett.

**Note**

See function `f12` whose zero at  $\sqrt{e}$  is difficult to find exactly for all root finders.

**Author(s)**

HwB email: <hwborchers@googlemail.com>

**References**

Press, Teukolsky, Vetterling, and Flannery (1992). Numerical Recipes in C. Cambridge University Press.

**See Also**

[brent](#)

## Examples

```

## Test functions
f1 <- function(x) # [0, 1.2], 0.399 422 2917
      x^2 * (x^2/3 + sqrt(2)*sin(x)) - sqrt(3)/18
f2 <- function(x) 11*x^11 - 1 # [0.4, 1.6], 0.804 133 0975
f3 <- function(x) 35*x^35 - 1 # [-0.5, 1.9], 0.903 407 6632
f4 <- function(x) # [-0.5, 0.7], 0.077 014 24135
      2*(x*exp(-9) - exp(-9*x)) + 1
f5 <- function(x) x^2 - (1 - x)^9 # [-1.4, 1], 0.259 204 4937
f6 <- function(x) (x-1)*exp(-9*x) + x^9 # [-0.8, 1.6], 0.536 741 6626
f7 <- function(x) x^2 + sin(x/9) - 1/4 # [-0.5, 1.9], 0.4475417621
f8 <- function(x) 1/8 * (9 - 1/x) # [0.001, 1.201], 0.111 111 1111
f9 <- function(x) tan(x) - x - 0.0463025 # [-0.9, 1.5], 0.500 000 0340
f10 <- function(x) # [0.4, 1], 0.679 808 9215
      x^2 + x*sin(sqrt(75)*x) - 0.2
f11 <- function(x) x^9 + 0.0001 # [-1.2, 0], -0.359 381 3664
f12 <- function(x) # [1, 3.4], 1.648 721 27070
      log(x) + x^2/(2*exp(1)) - 2 * x/sqrt(exp(1)) + 1

r <- ridders(f1 , 0, 1.2); r$root; r$niter # 18
r <- ridders(f2 , 0.4, 1.6); r$root; r$niter # 14
r <- ridders(f3 , -0.5, 1.9); r$root; r$niter # 20
r <- ridders(f4 , -0.5, 0.7); r$root; r$niter # 12
r <- ridders(f5 , -1.4, 1); r$root; r$niter # 16
r <- ridders(f6 , -0.8, 1.6); r$root; r$niter # 20
r <- ridders(f7 , -0.5, 1.9); r$root; r$niter # 16
r <- ridders(f8 , 0.001, 1.201); r$root; r$niter # 18
r <- ridders(f9 , -0.9, 1.5); r$root; r$niter # 20
r <- ridders(f10, 0.4, 1); r$root; r$niter # 14
r <- ridders(f11, -1.2, 0); r$root; r$niter # 12
r <- ridders(f12, 1, 3.4); r$root; r$niter # 30, err = 1e-5

## Not run:
## Use ridders() with Rmpfr
options(digits=16)
library("Rmpfr") # unirootR
prec <- 256
.N <- function(.) mpfr(., precBits = prec)

f12 <- function(x) {
  e1 <- exp(.N(1))
  log(x) + x^2/(2*e1) - 2*x/sqrt(e1) + 1
}
sqрте <- sqrt(exp(.N(1))) # 1.648721270700128...
f12(sqрте) # 0

unirootR(f12, interval=mpfr(c(1, 3.4), prec), tol=1e-20)
# $root
# 1 'mpfr' number of precision 200 bits
# [1] 1.648721270700128...

ridders(f12, .N(1), .N(3.4), maxiter=200, tol=1e-20)

```

```
# $root
# 1 'mpfr' number of precision 200 bits
# [1] 1.648721270700128...

## End(Not run)
```

---

rk4, rk4sys

*Classical Runge-Kutta*


---

### Description

Classical Runge-Kutta of order 4.

### Usage

```
rk4(f, a, b, y0, n)
```

```
rk4sys(f, a, b, y0, n)
```

### Arguments

f	function in the differential equation $y' = f(x, y)$ ; defined as a function $R \times R^m \rightarrow R^m$ , where $m$ is the number of equations.
a, b	endpoints of the interval.
y0	starting values; for $m$ equations y0 needs to be a vector of length $m$ .
n	the number of steps from a to b.

### Details

Classical Runge-Kutta of order 4 for (systems of) ordinary differential equations with fixed step size.

### Value

List with components x for grid points between a and b and y an n-by-m matrix with solutions for variables in columns, i.e. each row contains one time stamp.

### Note

This function serves demonstration purposes only.

### References

Fausett, L. V. (2007). Applied Numerical Analysis Using Matlab. Second edition, Prentice Hall.

### See Also

[ode23](#), [deval](#)

**Examples**

```
## Example1: ODE
# y' = y*(-2*x + 1/x) for x != 0, 1 if x = 0
# solution is x*exp(-x^2)
f <- function(x, y) {
  if (x != 0) dy <- y * (- 2*x + 1/x)
  else      dy <- rep(1, length(y))
  return(dy)
}
sol <- rk4(f, 0, 2, 0, 50)
## Not run:
x <- seq(0, 2, length.out = 51)
plot(x, x*exp(-x^2), type = "l", col = "red")
points(sol$x, sol$y, pch = "*")
grid()
## End(Not run)

## Example2: Chemical process
f <- function(t, u) {
  u1 <- u[3] - 0.1 * (t+1) * u[1]
  u2 <- 0.1 * (t+1) * u[1] - 2 * u[2]
  u3 <- 2 * u[2] - u[3]
  return(c(u1, u2, u3))
}
u0 <- c(0.8696, 0.0435, 0.0870)
a <- 0; b <- 40
n <- 40
sol <- rk4sys(f, a, b, u0, n)
## Not run:
matplot(sol$x, sol$y, type = "l", lty = 1, lwd = c(2, 1, 1),
  col = c("darkred", "darkblue", "darkgreen"),
  xlab = "Time [min]", ylab= "Concentration [Prozent]",
  main = "Chemical composition")
grid()
## End(Not run)
```

rkf54

*Runge-Kutta-Fehlberg***Description**

Runge-Kutta-Fehlberg with adaptive step size.

**Usage**

```
rkf54(f, a, b, y0, tol = .Machine$double.eps^0.5,
  control = list(), ...)
```



**Arguments**

f	function in the differential equation $y' = f(x, y)$ .
a, b	endpoints of the interval.
y0	starting values at a.
tol	relative tolerance, used for determining the step size.
control	list for influencing the step size with components hmin, hmax, the minimal, maximal step size jmax, the maximally allowed number of steps.
...	additional parameters to be passed to the function.

**Details**

Runge-Kutta-Fehlberg is a kind of Runge-Kutta method of solving ordinary differential equations of order (5, 4) with variable step size.

“At each step, two different approximations for the solution are made and compared. If the two answers are in close agreement, the approximation is accepted. If the two answers do not agree to a specified accuracy, the step size is reduced. If the answers agree to more significant digits than required, the step size is increased.”

Some textbooks promote the idea to use the five-order formula as the accepted value instead of using it for error estimation. This approach is taken here, that is why the function is called rkf54. The idea is still debated as the accuracy determinations appears to be diminished.

**Value**

List with components x for grid points between a and b and y the function values of the numerical solution.

**Note**

This function serves demonstration purposes only.

**References**

Stoer, J., and R. Bulirsch (2002). Introduction to Numerical Analysis. Third Edition, Springer-Verlag, New York.

Mathematica code associated with the book:

Mathews, J. H., and K. D. Fink (2004). Numerical Methods Using Matlab. Fourth Edition, Prentice Hall.

**See Also**

[rk4](#), [ode23](#)

**Examples**

```

# Example:  $y' = 1 + y^2$ 
f1 <- function(x, y) 1 + y^2
sol11 <- rkf54(f1, 0, 1.1, 0.5, control = list(hmin = 0.01))
sol12 <- rkf54(f1, 0, 1.1, 0.5, control = list(jmax = 250))

# Riccati equation:  $y' = x^2 + y^2$ 
f2 <- function(x, y) x^2 + y^2
sol21 <- rkf54(f2, 0, 1.5, 0.5, control = list(hmin = 0.01))
sol22 <- rkf54(f2, 0, 1.5, 0.5, control = list(jmax = 250))

## Not run:
plot(0, 0, type = "n", xlim = c(0, 1.5), ylim = c(0, 20),
     main = "Riccati", xlab = "", ylab = "")
points(sol11$x, sol11$y, pch = "*", col = "darkgreen")
lines(sol12$x, sol12$y)
points(sol21$x, sol21$y, pch = "*", col = "blue")
lines(sol22$x, sol22$y)
grid()
## End(Not run)

```

rmserr

*Accuracy Measures***Description**

Calculates different accuracy measures, most prominently RMSE.

**Usage**

```
rmserr(x, y, summary = FALSE)
```

**Arguments**

x, y	two vectors of real numbers
summary	logical; should a summary be printed to the screen?

**Details**

Calculates six different measures of accuracy for two given vectors or sequences of real numbers:

MAE	Mean Absolute Error
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
MAPE	Mean Absolute Percentage Error
LMSE	Normalized Mean Squared Error
rSTD	relative Standard Deviation

**Value**

Returns a list with different accuracy measures.

**Note**

Often used in Data Mining for *predicting* the accuracy of predictions.

**References**

Gentle, J. E. (2009). Computational Statistics, section 10.3. Springer Science+Business Media LCC, New York.

**Examples**

```
x <- rep(1, 10)
y <- rnorm(10, 1, 0.1)
rmserr(x, y, summary = TRUE)
```

---

romberg

*Romberg Integration*

---

**Description**

Romberg Integration

**Usage**

```
romberg(f, a, b, maxit = 25, tol = 1e-12, ...)
```

**Arguments**

f	function to be integrated.
a, b	end points of the interval.
maxit	maximum number of iterations.
tol	requested tolerance.
...	variables to be passed to the function.

**Details**

Simple Romberg integration with an explicit Richardson method applied to a series of trapezoidal integrals. This scheme works best with smooth and non-oscillatory functions and needs the least number of function calls among all integration routines.

The function does *not* need to be vectorized.

**Value**

List of value, number of iterations, and relative error.

**Note**

Using a trapezoid formula Romberg integration will use  $2*(2^{\text{iter}-1})+\text{iter}$  function calls. By remembering function values this could be reduced to  $2^{\text{iter}+1}$  calls.

**References**

Chapra, S. C., and R. P. Canale (2006). Numerical Methods for Engineers. Fifth Edition, McGraw-Hill, New York.

**See Also**

[integrate](#), [quadgr](#)

**Examples**

```
romberg(sin, 0, pi, tol = 1e-15) # 2 , rel.error 1e-15
romberg(exp, 0, 1, tol = 1e-15) # 1.718281828459044 , rel error 1e-15
# 1.718281828459045 , i.e. exp(1) - 1

f <- function(x, p) sin(x) * cos(p*x)
romberg(f, 0, pi, p = 2) # 2/3 , abs.err 1.5e-14
# value: -0.6666667, iter: 7, rel.error: 1e-12
```

---

roots, polyroots      *Polynomial Roots*

---

**Description**

Computes the roots (and multiplicities) of a polynomial.

**Usage**

```
roots(p)
polyroots(p, ntol = 1e-04, ztol = 1e-08)

rootsmult(p, r, tol=1e-12)
```

**Arguments**

**p**                    vector of real or complex numbers representing the polynomial.  
**r**                    a possible root of the polynomial.  
**tol, ntol, ztol**    norm tolerance and accuracy for polyroots.

## Details

The function `roots` computes roots of a polynomial as eigenvalues of the companion matrix.

`polyroots` attempts to refine the results of `roots` with special attention to multiple roots. For a reference of this implementation see F. C. Chang, "Solving multiple-root polynomials", IEEE Antennas and Propagation Magazine Vol. 51, No. 6 (2010), pp. 151-155.

`rootsmult` determines the order of a possible root `r`. As this computation is problematic in double precision, the result should be taken with a grain of salt.

## Value

`roots` returns a vector holding the roots of the polynomial, `rootsmult` the multiplicity of a root as an integer. And `polyroots` returns a data frame with a column 'root' and a column 'mult' giving the multiplicity of that root.

## See Also

[polyroot](#)

## Examples

```
roots(c(1, 0, 1, 0, 0))           # 0 0 1i -1i
p <- Poly(c(-2, -1, 0, 1, 2))     # 1*x^5 - 5*x^3 + 4*x
roots(p)                          # 0 -2 2 -1 1

p <- Poly(c(rep(1, 4), rep(-1, 4), 0, 0)) # 1 0 -4 0 6 0 -4 0 1
rootsmult(p, 1.0); rootsmult(p, -1.0) # 4 4
polyroots(p)
##  root mult
##  1    0    2
##  2    1    4
##  3   -1    4
```

---

rosser

*Rosser Matrix*

---

## Description

Generate the Rosser matrix.

## Usage

```
rosser()
```

## Details

This is a classic symmetric eigenvalue test problem. It has a double eigenvalue, three nearly equal eigenvalues, dominant eigenvalues of opposite sign, a zero eigenvalue, and a small, nonzero eigenvalue.

**Value**

matrix of size 8 x 8

**See Also**

[wilkinson](#)

**Examples**

```
rosser()
```

---

rot90

*Matrix Rotation*

---

**Description**

Rotate matrices for 90, 180, or 270 degrees..

**Usage**

```
rot90(a, k = 1)
```

**Arguments**

a                    numeric or complex matrix  
k                    scalar integer number of times the matrix will be rotated for 90 degrees; may be negative.

**Details**

Rotates a numeric or complex matrix for 90 (k = 1), 180 (k = 2) or 270 (k = 3 or k = -1) degrees.  
Value of k is taken mod 4.

**Value**

the original matrix rotated

**Examples**

```
a <- matrix(1:12, nrow=3, ncol=4, byrow=TRUE)
rot90(a)
rot90(a, 2)
rot90(a, -1)
```

---

rref

*Reduced Row Echelon Form*

---

### Description

Produces the reduced row echelon form of A using Gauss Jordan elimination with partial pivoting.

### Usage

rref(A)

### Arguments

A                    numeric matrix.

### Details

A matrix of "row-reduced echelon form" has the following characteristics:

1. All zero rows are at the bottom of the matrix
2. The leading entry of each nonzero row after the first occurs to the right of the leading entry of the previous row.
3. The leading entry in any nonzero row is 1.
4. All entries in the column above and below a leading 1 are zero.

Roundoff errors may cause this algorithm to compute a different value for the rank than rank, or th or null.

### Value

A matrix the same size as m.

### Note

This serves demonstration purposes only; don't use for large matrices.

### References

Weisstein, Eric W. "Echelon Form." From MathWorld – A Wolfram Web Resource.  
<https://mathworld.wolfram.com/EchelonForm.html>

### See Also

[qr.solve](#)

**Examples**

```

A <- matrix(c(1, 2, 3, 1, 3, 2, 3, 2, 1), 3, 3, byrow = TRUE)
rref(A)
#      [,1] [,2] [,3]
# [1,]  1   0   0
# [2,]  0   1   0
# [3,]  0   0   1

A <- matrix(data=c(1, 2, 3, 2, 5, 9, 5, 7, 8, 20, 100, 200),
            nrow=3, ncol=4, byrow=FALSE)
rref(A)
#  1   0   0  120
#  0   1   0   0
#  0   0   1 -20

# Use rref on a rank-deficient magic square:
A = magic(4)
R = rref(A)
zapsmall(R)
#  1   0   0   1
#  0   1   0   3
#  0   0   1  -3
#  0   0   0   0

```

---

runge

*Runge Function*


---

**Description**

Runge's test function for interpolation techniques.

**Usage**

```
runge(x)
```

**Arguments**

x                    numeric scalar.

**Details**

Runge's function is a classical test function for interpolation and approximation techniques, especially for equidistant nodes.

For example, when approximating the Runge function on the interval  $[-1, 1]$ , the error at the endpoints will diverge when the number of nodes is increasing.

**Value**

Numerical value of the function.



**See Also**[fnorm](#)**Examples**

```
## Not run:
x <- seq(-1, 1, length.out = 101)
y <- runge(x)
plot(x, y, type = "l", lwd = 2, col = "navy", ylim = c(-0.2, 1.2))
grid()

n <- c(6, 11, 16)
for (i in seq(along=n)) {
  xp <- seq(-1, 1, length.out = n[i])
  yp <- runge(xp)
  p <- polyfit(xp, yp, n[i]-1)
  y <- polyval(p, x)
  lines(x, y, lty=i) }

## End(Not run)
```

---

savgol

*Savitzky-Golay Smoothing*

---

**Description**

Polynomial filtering method of Savitzky and Golay.

**Usage**

```
savgol(T, fl, forder = 4, dorder = 0)
```

**Arguments**

T	Vector of signals to be filtered.
fl	Filter length (for instance fl = 51..151), has to be odd.
forder	Filter order (2 = quadratic filter, 4 = quartic).
dorder	Derivative order (0 = smoothing, 1 = first derivative, etc.).

**Details**

Savitzky-Golay smoothing performs a local polynomial regression on a series of values which are treated as being equally spaced to determine the smoothed value for each point. Methods are also provided for calculating derivatives.

**Value**

Vector representing the smoothed time series.

**Note**

For derivatives T2 has to be divided by the step size to the order (and to be multiplied by  $k!$  — the sign appears to be wrong).

**Author(s)**

Peter Riegler implemented a Matlab version in 2001. Based on this, Hans W. Borchers published an R version in 2003.

**References**

See Numerical Recipes, 1992, Chapter 14.8, for details.

**See Also**

RTisean::sav\_gol, signal::sgolayfilt, [whittaker](#).

**Examples**

```
# *** Sinosoid test function ***
ts <- sin(2*pi*(1:1000)/200)
t1 <- ts + rnorm(1000)/10
t2 <- savgol(t1, 51)
## Not run:
plot( 1:1000, t1, col = "grey")
lines(1:1000, ts, col = "blue")
lines(1:1000, t2, col = "red")
## End(Not run)
```

---

segm\_distance

*Segment Distance*

---

**Description**

The minimum distance between a point and a segment, or the minimum distance between points of two segments.

**Usage**

```
segm_distance(p1, p2, p3, p4 = c())
```

**Arguments**

p1, p2           end points of the first segment.  
p3, p4           end points of the second segment, or the point p3 alone if p4 is NULL.

**Details**

If `p4=c()`, determines the orthogonal line to the segment through the single point and computes the distance to the intersection point.

Otherwise, it computes the distances of all four end points to the other segment and takes the minimum of those.

**Value**

Returns a list with component 1 the minimum distance and components p, q the two nearest points.

If `p4=c()` then point p lies on the segment and q is p4.

**Note**

The interfaces of `segm_intersect` and `segm_distance` should be brought into line.

**See Also**

[segm\\_intersect](#)

**Examples**

```
## Not run:
plot(c(0, 1), c(0, 1), type = "n", asp=1,
      xlab = "", ylab = "", main = "Segment Distances")
grid()
for (i in 1:20) {
  s1 <- matrix(runif(4), 2, 2)
  s2 <- matrix(runif(4), 2, 2)
  lines(s1[, 1], s1[, 2], col = "red")
  lines(s2[, 1], s2[, 2], col = "darkred")
  S <- segm_distance(s1[,1,], s1[,2,], s2[,1,], s2[,2,])
  S$d
  points(c(S$p[1], S$q[1]), c(S$p[2], S$q[2]), pch=20, col="navy")
  lines(c(S$p[1], S$q[1]), c(S$p[2], S$q[2]), col="gray")
}
## End(Not run)
```

---

segm\_intersect

*Segment Intersection*

---

**Description**

Do two segments have at least one point in common?

**Usage**

```
segm_intersect(s1, s2)
```

**Arguments**

`s1, s2` Two segments, represented by their end points; i.e., `s <- rbind(p1, p2)` when `p1, p2` are the end points.

**Details**

First compares the ‘bounding boxes’, and if those intersect looks at whether the other end points lie on different sides of each segment.

**Value**

Logical, TRUE if these segments intersect.

**Note**

Should be written without reference to the cross function. Should also return the intersection point, see the example.

**References**

Cormen, Th. H., Ch. E. Leiserson, and R. L. Rivest (2009). Introduction to Algorithms. Third Edition, The MIT Press, Cambridge, MA.

**See Also**

[segm\\_distance](#)

**Examples**

```
## Not run:
plot(c(0, 1), c(0, 1), type="n",
      xlab = "", ylab = "", main = "Segment Intersection")
grid()
for (i in 1:20) {
  s1 <- matrix(runif(4), 2, 2)
  s2 <- matrix(runif(4), 2, 2)
  if (segm_intersect(s1, s2)) {
    clr <- "red"
    p1 <- s1[1, ]; p2 <- s1[2, ]; p3 <- s2[1, ]; p4 <- s2[2, ]
    A <- cbind(p2 - p1, p4 - p3)
    b <- (p3 - p1)
    a <- solve(A, b)
    points((p1 + a[1]*(p2-p1))[1], (p1 + a[1]*(p2-p1))[2], pch = 19, col = "blue")
  } else
    clr <- "darkred"
  lines(s1[,1], s1[, 2], col = clr)
  lines(s2[,1], s2[, 2], col = clr)
}
## End(Not run)
```

---

**semilogx,semilogy**      *Semi-logarithmic Plots (Matlab Style)*

---

**Description**

Generates semi- and double-logarithmic plots.

**Usage**

```
semilogx(x, y, ...)
```

```
semilogy(x, y, ...)
```

```
loglog(x, y, ...)
```

**Arguments**

x, y                    x-, y-coordinates.

...                    additional graphical parameters passed to the plot function.

**Details**

Plots data in logarithmic scales for the x-axis or y-axis, or uses logarithmic scales in both axes, and adds grid lines.

**Value**

Generates a plot, returns nothing.

**Note**

Matlab's logarithmic plots find a more appropriate grid.

**See Also**

[plot](#) with log= option.

**Examples**

```
## Not run:  
x <- logspace(-1, 2)  
loglog(x, exp(x), type = 'b')  
## End(Not run)
```

shooting

*Shooting Method***Description**

The shooting method solves the boundary value problem for second-order differential equations.

**Usage**

```
shooting(f, t0, tfinal, y0, h, a, b,
        itermax = 20, tol = 1e-6, hmax = 0)
```

**Arguments**

f	function in the differential equation $y'' = f(x, y, y')$ .
t0, tfinal	start and end points of the interval.
y0	starting value of the solution.
h	function defining the boundary condition as a function at the end point of the interval.
a, b	two guesses of the derivative at the start point.
itermax	maximum number of iterations for the secant method.
tol	tolerance to be used for stopping and in the ode45 solver.
hmax	maximal step size, to be passed to the solver.

**Details**

A second-order differential equation is solved with boundary conditions  $y(t_0) = y_0$  at the start point of the interval, and  $h(y(t_{\text{final}}), dy/dt(t_{\text{final}})) = 0$  at the end. The zero of  $h$  is found by a simple secant approach.

**Value**

Returns a list with two components,  $t$  for grid (or 'time') points between  $t_0$  and  $t_{\text{final}}$ , and  $y$  the solution of the differential equation evaluated at these points.

**Note**

Replacing secant with Newton's method would be an easy exercise. The same for replacing ode45 with some other solver.

**References**

L. V. Fausett (2008). Applied Numerical Analysis Using MATLAB. Second Edition, Pearson Education Inc.

**See Also**[bvp](#)**Examples**

```
#-- Example 1
f <- function(t, y1, y2) -2*y1*y2
h <- function(u, v) u + v - 0.25

t0 <- 0; tfinal <- 1
y0 <- 1
sol <- shooting(f, t0, tfinal, y0, h, 0, 1)
## Not run:
plot(sol$t, sol$y[, 1], type='l', ylim=c(-1, 1))
xs <- linspace(0, 1); ys <- 1/(xs+1)
lines(xs, ys, col="red")
lines(sol$t, sol$y[, 2], col="gray")
grid()
## End(Not run)

#-- Example 2
f <- function(t, y1, y2) -y2^2 / y1
h <- function(u, v) u - 2
t0 <- 0; tfinal <- 1
y0 <- 1
sol <- shooting(f, t0, tfinal, y0, h, 0, 1)
```

---

shubert

*Shubert-Piyavskii Method*

---

**Description**

Shubert-Piyavskii Univariate Function Maximization

**Usage**

```
shubert(f, a, b, L, crit = 1e-04, nmax = 1000)
```

**Arguments**

f	function to be optimized.
a, b	search between a and b for a maximum.
L	a Lipschitz constant for the function.
crit	critical value
nmax	maximum number of steps.

## Details

The Shubert-Piyavskii method, often called the Sawtooth Method, finds the global maximum of a univariate function on a known interval. It is guaranteed to find the global maximum on the interval under certain conditions:

The function  $f$  is Lipschitz-continuous, that is there is a constant  $L$  such that

$$|f(x) - f(y)| \leq L|x - y|$$

for all  $x, y$  in  $[a, b]$ .

The process is stopped when the improvement in the last step is smaller than the input argument `crit`.

## Value

Returns a list with the following components:

<code>xopt</code>	the x-coordinate of the minimum found.
<code>fopt</code>	the function value at the minimum.
<code>nopt</code>	number of steps.

## References

Y. K. Yeo. Chemical Engineering Computation with MATLAB. CRC Press, 2017.

## See Also

[findmins](#)

## Examples

```
# Determine the global minimum of sin(1.2*x)+sin(3.5*x) in [-3, 8].
f <- function(x) sin(1.2*x) + sin(3.5*x)
shubert(function(x) -f(x), -3, 8, 5, 1e-04, 1000)
## $xopt
## [1] 3.216231      # 3.216209
## $fopt
## [1] 1.623964
## $nopt
## [1] 481
```



**Description**

Computes the sine and cosine integrals through approximations.

**Usage**

Si(x)

Ci(x)

**Arguments**

x                      Scalar or vector of real numbers.

**Details**

The sine and cosine integrals are defined as

$$Si(x) = \int_0^x \frac{\sin(t)}{t} dt$$

$$Ci(x) = - \int_x^\infty \frac{\cos(t)}{t} dt = \gamma + \log(x) + \int_0^x \frac{\cos(t) - 1}{t} dt$$

where  $\gamma$  is the Euler-Mascheroni constant.

**Value**

Returns a scalar of sine resp. cosine integrals applied to each element of the scalar/vector. The value Ci(x) is not correct, it should be Ci(x)+pi\*i, only the real part is returned!

The function is not truly vectorized, for vectors the values are calculated in a for-loop. The accuracy is about  $10^{-13}$  and better in a reasonable range of input values.

**References**

Zhang, S., and J. Jin (1996). Computation of Special Functions. Wiley-Interscience.

**See Also**

[sinc](#), [expint](#)

**Examples**

```

x <- c(-3:3) * pi
Si(x); Ci(x)

## Not run:
xs <- linspace(0, 10*pi, 200)
ysi <- Si(xs); yci <- Ci(xs)
plot(c(0, 35), c(-1.5, 2.0), type = 'n', xlab = '', ylab = '',
      main = "Sine and cosine integral functions")
lines(xs, ysi, col = "darkred", lwd = 2)
lines(xs, yci, col = "darkblue", lwd = 2)
lines(c(0, 10*pi), c(pi/2, pi/2), col = "gray")
lines(xs, cos(xs), col = "gray")
grid()
## End(Not run)

```

---

 sigmoid

*Sigmoid Function*


---

**Description**

Sigmoid function (aka sigmoidal curve or logistic function).

**Usage**

```

sigmoid(x, a = 1, b = 0)
logit(x, a = 1, b = 0)

```

**Arguments**

x	numeric vector.
a, b	parameters.

**Details**

The sigmoidal function with parameters a, b is the function

$$y = 1/(1 + e^{-a(x-b)})$$

The sigmoid function is also the solution of the ordinary differentialequation

$$y' = y(1 - y)$$

with  $y(0) = 1/2$  and has an indefinite integral  $\ln(1 + e^x)$ .

The logit function is the inverse of the sigmoid function and is (therefore) only defined between 0 and 1. Its definition is

$$y = b + 1/a \log(x/(1 - x))$$

The parameters must be scalars; if they are vectors, only the first component will be taken.

**Value**

Numeric/complex scalar or vector.

**Examples**

```
x <- seq(-6, 6, length.out = 101)
y1 <- sigmoid(x)
y2 <- sigmoid(x, a = 2)
## Not run:
plot(x, y1, type = "l", col = "darkblue",
      xlab = "", ylab = "", main = "Sigmoid Function(s)")
lines(x, y2, col = "darkgreen")
grid()
## End(Not run)

# The slope in 0 (in x = b) is a/4
# sigmf with slope 1 and range [-1, 1].
sigmf <- function(x) 2 * sigmoid(x, a = 2) - 1

# logit is the inverse of the sigmoid function
x <- c(-0.75, -0.25, 0.25, 0.75)
y <- sigmoid(x)
logit(y)      #=> -0.75 -0.25  0.25  0.75
```

---

simpadpt

*Adaptive Simpson Quadrature*


---

**Description**

Numerically evaluate an integral using adaptive Simpson's rule.

**Usage**

```
simpadpt(f, a, b, tol = 1e-6, ...)
```

**Arguments**

f	univariate function, the integrand.
a, b	lower limits of integration; must be finite.
tol	relative tolerance
...	additional arguments to be passed to f.

**Details**

Approximates the integral of the function f from a to b to within an error of tol using recursive adaptive Simpson quadrature.

**Value**

A numerical value or vector, the computed integral.

**Note**

Based on code from the book by Quarteroni et al., with some tricks borrowed from Matlab and Octave.

**References**

Quarteroni, A., R. Sacco, and F. Saleri (2007). Numerical Mathematics. Second Edition, Springer-Verlag, Berlin Heidelberg.

**See Also**

[quad](#), [simpson2d](#)

**Examples**

```
myf <- function(x, n) 1/(x+n) # 0.0953101798043249 , log((n+1)/n) for n=10
simpadpt(myf, 0, 1, n = 10) # 0.095310179804535

## Dilogarithm function
flog <- function(t) log(1-t) / t # singularity at t=1, almost at t=0
dilog <- function(x) simpadpt(flog, x, 0, tol = 1e-12)
dilog(1) # 1.64493406685615
          # 1.64493406684823 = pi^2/6

## Not run:
N <- 51
xs <- seq(-5, 1, length.out = N)
ys <- numeric(N)
for (i in 1:N) ys[i] <- dilog(xs[i])
plot(xs, ys, type = "l", col = "blue",
      main = "Dilogarithm function")
grid()
## End(Not run)
```

---

simpson2d

*Double Simpson Integration*

---

**Description**

Numerically evaluate double integral by 2-dimensional Simpson method.

**Usage**

```
simpson2d(f, xa, xb, ya, yb, nx = 128, ny = 128, ...)
```

**Arguments**

f	function of two variables, the integrand.
xa, xb	left and right endpoint for first variable.
ya, yb	left and right endpoint for second variable.
nx, ny	number of intervals in x- and y-direction.
...	additional parameters to be passed to the integrand.

**Details**

The 2D Simpson integrator has weights that are most easily determined by taking the outer product of the vector of weights for the 1D Simpson rule.

**Value**

Numerical scalar, the value of the integral.

**Note**

Copyright (c) 2008 W. Padden and Ch. Macaskill for Matlab code published under BSD License on MatlabCentral.

**See Also**

[dblquad](#), [quad2d](#)

**Examples**

```
f1 <- function(x, y) x^2 + y^2
simpson2d(f1, -1, 1, -1, 1) # 2.666666667 , i.e. 8/3 . err = 0

f2 <- function(x, y) y*sin(x)+x*cos(y)
simpson2d(f2, pi, 2*pi, 0, pi) # -9.869604401 , i.e. -pi^2, err = 2e-8

f3 <- function(x, y) sqrt((1 - (x^2 + y^2)) * (x^2 + y^2 <= 1))
simpson2d(f3, -1, 1, -1, 1) # 2.094393912 , i.e. 2/3*pi , err = 1e-6
```

---

sind,cosd,tand, etc. *Trigonometric Functions in Degrees*

---

**Description**

Trigonometric functions expecting input in degrees, not radians.

**Usage**

```
sind(x)
cosd(x)
tand(x)
cotd(x)
asind(x)
acosd(x)
atand(x)
acotd(x)
secd(x)
cscd(x)
asecd(x)
acscd(x)
atan2d(x1, x2)
```

**Arguments**

`x, x1, x2`          numeric or complex scalars or vectors

**Details**

The usual trigonometric functions with input values as scalar or vector in degrees. Note that `tand(x)` with fractional part does not return NaN as `tanpi(x)`, but is computed as `sind(x)/cosd(x)`.

For `atan2d` the inputs `x1, x2` can be both degrees or radians, but don't mix! The result is in degrees, of course.

**Value**

Returns a scalar or vector of numeric values.

**Note**

These function names are available in Matlab, that is the reason they have been added to the 'pracma' package.

**See Also**

Other trigonometric functions in R.

**Examples**

```
# sind(x) and cosd(x) are accurate for x which are multiples
# of 90 and 180 degrees, while tand(x) is problematic.

x <- seq(0, 720, by = 90)
sind(x)      # 0  1  0 -1  0  1  0 -1  0
cosd(x)      # 1  0 -1  0  1  0 -1  0  1
tand(x)      # 0  Inf  0 -Inf  0  Inf  0 -Inf  0
cotd(x)      # Inf  0 -Inf  0  Inf  0 -Inf  0  Inf
```

```
x <- seq(5, 85, by = 20)
asind(sind(x))          # 5 25 45 65 85
asecd(sec(x))
tand(x)                # 0.08748866  0.46630766  1.00000000  ...
atan2d(1, 1)           # 45
```

---

size	<i>Size of Matrix</i>
------	-----------------------

---

### Description

Provides the dimensions of x.

### Usage

```
size(x, k)
```

### Arguments

x	vector, matrix, or array
k	integer specifying a particular dimension

### Details

Returns the number of dimensions as `length(x)`.

Vector will be treated as a single row matrix.

### Value

vector containing the dimensions of x, or the k-th dimension if k is not missing.

### Note

The result will differ from Matlab when x is a character vector.

### See Also

[dim](#)

### Examples

```
size(1:8)
size(matrix(1:8, 2, 4)) # 2 4
size(matrix(1:8, 2, 4), 2) # 4
size(matrix(1:8, 2, 4), 3) # 1
```

---

`softline`*Soft (Inexact) Line Search*

---

**Description**

Fletcher's inexact line search algorithm.

**Usage**

```
softline(x0, d0, f, g = NULL)
```

**Arguments**

<code>x0</code>	initial point for linesearch.
<code>d0</code>	search direction from <code>x0</code> .
<code>f</code>	real function of several variables that is to be minimized.
<code>g</code>	gradient of objective function <code>f</code> ; computed numerically if not provided.

**Details**

Many optimization methods have been found to be quite tolerant to line search imprecision, therefore inexact line searches are often used in these methods.

**Value**

Returns the suggested inexact optimization parameter as a real number `a0` such that  $x0+a0*d0$  should be a reasonable approximation.

**Note**

Matlab version of an inexact linesearch algorithm by A. Antoniou and W.-S. Lu in their textbook "Practical Optimization". Translated to R by Hans W Borchers.

**References**

Fletcher, R. (1980). Practical Methods of Optimization, Volume 1., Section 2.6. Wiley, New York.

Antoniou, A., and W.-S. Lu (2007). Practical Optimization: Algorithms and Engineering Applications. Springer Science+Business Media, New York.

**See Also**

[gaussNewton](#)



### Examples

```
## Himmelblau function
f_himm <- function(x) (x[1]^2 + x[2] - 11)^2 + (x[1] + x[2]^2 - 7)^2
g_himm <- function(x) {
  w1 <- (x[1]^2 + x[2] - 11); w2 <- (x[1] + x[2]^2 - 7)
  g1 <- 4*w1*x[1] + 2*w2;      g2 <- 2*w1 + 4*w2*x[2]
  c(g1, g2)
}
# Find inexact minimum from [6, 6] in the direction [-1, -1] !
softline(c(6, 6), c(-1, -1), f_himm, g_himm)
# [1] 3.458463

# Find the same minimum by using the numerical gradient
softline(c(6, 6), c(-1, -1), f_himm)
# [1] 3.458463
```

---

sorting

*Sorting Routines*

---

### Description

R implementations of several sorting routines. These implementations are meant for demonstration and lecturing purposes.

### Usage

```
is.sorted(a)
testSort(n = 1000)

bubbleSort(a)
insertionSort(a)
selectionSort(a)
shellSort(a, f = 2.3)
heapSort(a)
mergeSort(a, m = 10)
mergeOrdered(a, b)
quickSort(a, m = 3)
quickSortx(a, m = 25)
```

### Arguments

a, b	Numeric vectors to be sorted or merged.
f	Retracting factor for shellSort.
m	Size of subsets that are sorted by insertionSort when the sorting procedure is called recursively.
n	Only in testSort: the length of a vector of random numbers to be sorted.

**Details**

`bubbleSort(a)` is the well-known “bubble sort” routine; it is forbiddingly slow.

`insertionSort(a)` sorts the array one entry at a time; it is slow, but quite efficient for small data sets.

`selectionSort(a)` is an in-place sorting routine that is inefficient, but noted for its simplicity.

`shellSort(a, f = 2.3)` exploits the fact that insertion sort works efficiently on input that is already almost sorted. It reduces the gaps by the factor  $f$ ;  $f=2.3$  is said to be a reasonable choice.

`heapSort(a)` is not yet implemented.

`mergeSort(a, m = 10)` works recursively, merging already sorted parts with `mergeOrdered`.  $m$  should be between 3 and  $1/1000$  of the size of  $a$ .

`mergeOrdered(a, b)` works only correctly if  $a$  and  $b$  are already sorted.

`quickSort(a, m = 3)` realizes the celebrated “quicksort algorithm” and is the fastest of all implementations here. To avoid too deeply nested recursion with R, `insertionSort` is called when the size of a subset is smaller than  $m$ .

Values between 3 . . 30 seem reasonable and smaller values are better, with the risk of running into a too deeply nested recursion.

`quickSort(a, m = 25)` is an extended version where the split is calculated more carefully, but in general this approach takes too much time.

Values for  $m$  are 20 . . 40 with  $m=25$  favoured.

`testSort(n = 1000)` is a test routine, e.g. for testing your computer power. On an iMac, `quickSort` will sort an array of size 1,000,000 in less than 15 secs.

**Value**

All routines return the vector sorted.

`is.sorted` indicates logically whether the vector is sorted.

**Note**

At the moment, only increasingly sorting is possible (if needed apply `rev` afterwards).

**Author(s)**

HwB <hwborchers@googlemail.com>

**References**

Knuth, D. E. (1973). The Art of Computer Programming, Volume 3: Sorting and Searching, Chapter 5: Sorting. Addison-Wesley Publishing Company.

**See Also**

[sort](#), the internal C-based sorting routine.

**Examples**

```
## Not run:
testSort(100)

a <- sort(runif(1000)); b <- sort(runif(1000))
system.time(y <- mergeSort(c(a, b)))
system.time(y <- mergeOrdered(a, b))
is.sorted(y)
## End(Not run)
```

---

sortrows

*Sort Rows of a Matrix (Matlab Style)*

---

**Description**

Sort rows of a matrix according to values in a column.

**Usage**

```
sortrows(A, k = 1)
```

**Arguments**

A                numeric matrix.  
k                number of column to sort the matrix accordingly.

**Details**

sortrows(A, k) sorts the rows of the matrix A such that column k is increasingly sorted.

**Value**

Returns the sorted matrix.

**See Also**

[sort](#)

**Examples**

```
A <- magic(5)
sortrows(A)
sortrows(A, k = 2)
```

---

 spinterp

---

*Monotone (Shape-Preserving) Interpolation*


---

**Description**

Monotone interpolation preserves the monotonicity of the data being interpolated, and when the data points are also monotonic, the slopes of the interpolant should also be monotonic.

**Usage**

```
spinterp(x, y, xp)
```

**Arguments**

x, y	x- and y-coordinates of the points that shall be interpolated.
xp	points that should be interpolated.

**Details**

This implementation follows a cubic version of the method of Delbourgo and Gregory. It yields ‘shaplier’ curves than the Stineman method.

The calculation of the slopes is according to recommended practice:

- monotonic and convex → harmonic
- monotonic and nonconvex → geometric
- nonmonotonic and convex → arithmetic
- nonmonotonic and nonconvex → circles (Stineman) [not implemented]

The choice of supplementary coefficients  $r[i]$  depends on whether the data are monotonic or convex or both:

- monotonic, but not convex
- otherwise

and that can be detected from the data. The choice  $r[i]=3$  for all  $i$  results in the standard cubic Hermitean rational interpolation.

**Value**

The interpolated values at all the points of  $xp$ .

**Note**

At the moment, the data need to be monotonic and the case of convexity is not considered.

**References**

Stan Wagon (2010). *Mathematica in Action*. Third Edition, Springer-Verlag.

**See Also**

stinepack::stinterp, demography::cm.interp

**Examples**

```

data1 <- list(x = c(1,2,3,5,6,8,9,11,12,14,15),
             y = c(rep(10,6), 10.5,15,50,60,95))
data2 <- list(x = c(0,1,4,6.5,9,10),
             y = c(10,4,2,1,3,10))
data3 <- list(x = c(7.99,8.09,8.19,8.7,9.2,10,12,15,20),
             y = c(0,0.000027629,0.00437498,0.169183,0.469428,
                 0.94374,0.998636,0.999919,0.999994))
data4 <- list(x = c(22,22.5,22.6,22.7,22.8,22.9,
                 23,23.1,23.2,23.3,23.4,23.5,24),
             y = c(523,543,550,557,565,575,
                 590,620,860,915,944,958,986))
data5 <- list(x = c(0,1.1,1.31,2.5,3.9,4.4,5.5,6,8,10.1),
             y = c(10.1,8,4.7,4.0,3.48,3.3,5.8,7,7.7,8.6))

data6 <- list(x = c(-0.8, -0.75, -0.3, 0.2, 0.5),
             y = c(-0.9, 0.3, 0.4, 0.5, 0.6))
data7 <- list(x = c(-1, -0.96, -0.88, -0.62, 0.13, 1),
             y = c(-1, -0.4, 0.3, 0.78, 0.91, 1))

data8 <- list(x = c(-1, -2/3, -1/3, 0.0, 1/3, 2/3, 1),
             y = c(-1, -(2/3)^3, -(1/3)^3, -(1/3)^3, (1/3)^3, (1/3)^3, 1))

## Not run:
opr <- par(mfrow=c(2,2))

# These are well-known test cases:
D <- data1
plot(D, ylim=c(0, 100)); grid()
xp <- seq(1, 15, len=51); yp <- spinterp(D$x, D$y, xp)
lines(spline(D), col="blue")
lines(xp, yp, col="red")

D <- data3
plot(D, ylim=c(0, 1.2)); grid()
xp <- seq(8, 20, len=51); yp <- spinterp(D$x, D$y, xp)
lines(spline(D), col="blue")
lines(xp, yp, col="red")

D <- data4
plot(D); grid()
xp <- seq(22, 24, len=51); yp <- spinterp(D$x, D$y, xp)
lines(spline(D), col="blue")
lines(xp, yp, col="red")

# Fix a horizontal slope at the end points
D <- data8
x <- c(-1.05, D$x, 1.05); y <- c(-1, D$y, 1)

```

```

plot(D); grid()
xp <- seq(-1, 1, len=101); yp <- spinterp(x, y, xp)
lines(spline(D, n=101), col="blue")
lines(xp, yp, col="red")

par(opr)
## End(Not run)

```

---

sqrtm,rootm

---

*Matrix Square and p-th Roots*


---

### Description

Computes the matrix square root and matrix p-th root of a nonsingular real matrix.

### Usage

```

sqrtm(A, kmax = 20, tol = .Machine$double.eps^(1/2))
signm(A, kmax = 20, tol = .Machine$double.eps^(1/2))

rootm(A, p, kmax = 20, tol = .Machine$double.eps^(1/2))

```

### Arguments

A	numeric, i.e. real, matrix.
p	p-th root to be taken.
kmax	maximum number of iterations.
tol	absolut tolerance, norm distance of A and B^p.

### Details

A real matrix may or may not have a real square root; if it has no real negative eigenvalues. The number of square roots can vary from two to infinity. A positive definite matrix has one distinguished square root, called the principal one, with the property that the eigenvalues lie in the segment  $\{z \mid -\pi/p < \arg(z) < \pi/p\}$  (for the p-th root).

The matrix square root  $\text{sqrtm}(A)$  is computed here through the Denman-Beavers iteration (see the references) with quadratic rate of convergence, a refinement of the common Newton iteration determining roots of a quadratic equation.

The matrix p-th root  $\text{rootm}(A)$  is computed as a complex integral

$$A^{1/p} = \frac{p \sin(\pi/p)}{\pi} A \int_0^\infty (x^p I + A)^{-1} dx$$

applying the trapezoidal rule along the unit circle.

One application is the computation of the matrix logarithm as

$$\log A = 2^k \log A^{1/2^k}$$

such that the argument to the logarithm is close to the identity matrix and the Pade approximation can be applied to  $\log(I + X)$ .

The matrix sector function is defined as  $\text{sectm}(A, m) = (A^m)^{-1/p} A$ ; for  $p=2$  this is the matrix sign function.

$S = \text{signm}(A)$  is real if  $A$  is and has the following properties:

$S^2 = \text{Id}$ ;  $S A = A S$

$\text{singm}([\theta A; B \ 0]) = [\theta C; C^{-1} \ 0]$  where  $C = A(BA)^{-1}$

These functions arise in control theory.

### Value

`sqrtm(A)` returns a list with components

<code>B</code>	square root matrix of $A$ .
<code>Binv</code>	inverse of the square root matrix.
<code>k</code>	number of iterations.
<code>acc</code>	accuracy or absolute error.

`rootm(A)` returns a list with components

<code>B</code>	square root matrix of $A$ .
<code>k</code>	number of iterations.
<code>acc</code>	accuracy or absolute error.

If  $k$  is negative the iteration has *not* converged.

`signm` just returns one matrix, even when there was no convergence.

### Note

The  $p$ -th root of a positive definite matrix can also be computed from its eigenvalues as

```
E <- eigen(A)
```

```
V <- E$vectors; U <- solve(V)
```

```
D <- diag(E$values)
```

```
B <- V %*% D^(1/p) %*% U
```

or by applying the functions `expm`, `logm` in package ‘`expm`’:

```
B <- expm(1/p * logm(A))
```

As these approaches all calculate the principal branch, the results are identical (but will numerically slightly differ).

### References

N. J. Higham (1997). Stable Iterations for the Matrix Square Root. *Numerical Algorithms*, Vol. 15, pp. 227–242.

D. A. Bini, N. J. Higham, and B. Meini (2005). Algorithms for the matrix  $p$ th root. *Numerical Algorithms*, Vol. 39, pp. 349–378.

**See Also**

`expm`, `expm::sqrtm`

**Examples**

```
A1 <- matrix(c(10, 7, 8, 7,
              7, 5, 6, 5,
              8, 6, 10, 9,
              7, 5, 9, 10), nrow = 4, ncol = 4, byrow = TRUE)

X <- sqrtm(A1)$B # accuracy: 2.352583e-13
X

A2 <- matrix(c(90.81, 8.33, 0.68, 0.06, 0.08, 0.02, 0.01, 0.01,
              0.70, 90.65, 7.79, 0.64, 0.06, 0.13, 0.02, 0.01,
              0.09, 2.27, 91.05, 5.52, 0.74, 0.26, 0.01, 0.06,
              0.02, 0.33, 5.95, 85.93, 5.30, 1.17, 1.12, 0.18,
              0.03, 0.14, 0.67, 7.73, 80.53, 8.84, 1.00, 1.06,
              0.01, 0.11, 0.24, 0.43, 6.48, 83.46, 4.07, 5.20,
              0.21, 0, 0.22, 1.30, 2.38, 11.24, 64.86, 19.79,
              0, 0, 0, 0, 0, 0, 0, 100
              ) / 100, nrow = 8, ncol = 8, byrow = TRUE)

X <- rootm(A2, 12) # k = 6, accuracy: 2.208596e-14

## Matrix sign function
signm(A1) # 4x4 identity matrix
B <- rbind(cbind(zeros(4,4), A1),
           cbind(eye(4), zeros(4,4)))
signm(B) # [0, signm(A1)$B; signm(A1)$Binv 0]
```

---

squareform

*Format Distance Matrix (Matlab Style)*

---

**Description**

Format or generate a distance matrix.

**Usage**

```
squareform(x)
```

**Arguments**

`x` numeric vector or matrix.



**Details**

If `x` is a vector as created by the `dist` function, it converts it into a full square, symmetric matrix. And if `x` is a distance matrix, i.e. square, symmetric and with zero diagonal elements, it returns the flattened lower triangular submatrix.

**Value**

Returns a matrix if `x` is a vector, and a vector if `x` is a matrix.

**See Also**

[dist](#)

**Examples**

```
x <- 1:6
y <- squareform(x)
# 0 1 2 3
# 1 0 4 5
# 2 4 0 6
# 3 5 6 0
all(squareform(y) == x)
# TRUE
```

---

std

*Standard Deviation (Matlab Style)*

---

**Description**

Standard deviation of the values of `x`.

**Usage**

```
std(x, flag=0)
```

**Arguments**

<code>x</code>	numeric vector or matrix
<code>flag</code>	numeric scalar. If 0, selects unbiased algorithm; and if 1, selects the biased version.

**Details**

If `flag = 0` the result is the square root of an unbiased estimator of the variance. `std(X, 1)` returns the standard deviation producing the second moment of the set of values about their mean.

**Value**

Return value depends on argument  $x$ . If vector, returns the standard deviation. If matrix, returns vector containing the standard deviation of each column.

**Note**

$flag = 0$  produces the same result as R's `sd()`.

**Examples**

```
std(1:10)          # 3.027650
std(1:10, flag=1) # 2.872281
```

---

std_err	<i>Standard Error</i>
---------	-----------------------

---

**Description**

Standard error of the values of  $x$ .

**Usage**

```
std_err(x)
```

**Arguments**

$x$                     numeric vector or matrix

**Details**

Standard error is computed as  $\text{var}(x)/\text{length}(x)$ .

**Value**

Returns the standard error of all elements of the vector or matrix.

**Examples**

```
std_err(1:10) #=> 0.9574271
```

---

steep\_descent                      *Steepest Descent Minimization*

---

**Description**

Function minimization by steepest descent.

**Usage**

```
steep_descent(x0, f, g = NULL, info = FALSE,  
              maxiter = 100, tol = .Machine$double.eps^(1/2))
```

**Arguments**

x0	start value.
f	function to be minimized.
g	gradient function of f; if NULL, a numerical gradient will be calculated.
info	logical; shall information be printed on every iteration?
maxiter	max. number of iterations.
tol	relative tolerance, to be used as stopping rule.

**Details**

Steepest descent is a line search method that moves along the downhill direction.

**Value**

List with following components:

xmin	minimum solution found.
fmin	value of f at minimum.
niter	number of iterations performed.

**Note**

Used some Matlab code as described in the book “Applied Numerical Analysis Using Matlab” by L. V.Fausett.

**References**

Nocedal, J., and S. J. Wright (2006). Numerical Optimization. Second Edition, Springer-Verlag, New York, pp. 22 ff.

**See Also**

[fletcher\\_powell](#)

**Examples**

```
## Rosenbrock function: The flat valley of the Rosenbrock function makes
## it infeasible for a steepest descent approach.
# rosenbrock <- function(x) {
#   n <- length(x)
#   x1 <- x[2:n]
#   x2 <- x[1:(n-1)]
#   sum(100*(x1-x2^2)^2 + (1-x2)^2)
# }
# steep_descent(c(1, 1), rosenbrock)
# Warning message:
# In steep_descent(c(0, 0), rosenbrock) :
# Maximum number of iterations reached -- not converged.

## Sphere function
sph <- function(x) sum(x^2)
steep_descent(rep(1, 10), sph)
# $xmin  0 0 0 0 0 0 0 0 0 0
# $fmin  0
# $niter 2
```

---

stereographic

*Stereographic Projection*


---

**Description**

The stereographic projection is a function that maps the  $n$ -dimensional sphere from the South pole  $(0, \dots, -1)$  to the tangent plane of the sphere at the north pole  $(0, \dots, +1)$ .

**Usage**

```
stereographic(p)
```

```
stereographic_inv(q)
```

**Arguments**

**p** point on the  $n$ -sphere ; can also be a set of points, each point represented as a column of a matrix.

**q** point on the tangent plane at the north pole (last coordinate = 1); can also be a set of such points.

**Details**

The stereographic projection is a smooth function from  $S^n - (0, \dots, -1)$  to the tangent hyperplane at the north pole. The south pole is mapped to infinity, that is why one speaks of  $S^n$  as a 'one-point compactification' of  $R^{n-1}$ .

All mapped points will have a last coordinate 1.0 (lying on the tangent plane.) Points mapped by 'stereographic\_inv' are assumed to have a last coordinate 1.0 (this will not be checked), otherwise the result will be different from what is expected – though the result is still correct in itself.

All points are column vectors: stereographic will transform a row vector to column; stereographic\_inv will return a single vector as column.

### Value

Returns a point (or a set of point) of (n-1) dimensions on the tangent plane resp. an n-dimensional point on the n-sphere, i.e.,  $\sum(x^2) = 1$ .

### Note

To map a region around the south pole, a similar function would be possible. Instead it is simpler to change the sign of the last coordinate.

### Author(s)

Original MATLAB code by J.Burkardt under LGPL license; rewritten in R by Hans W Borchers.

### References

See the "Stereographic projection" article on Wikipedia.

### Examples

```
# points in the xy-plane (i.e., z = 0)
A <- matrix(c(1,0,0, -1,0,0, 0,1,0, 0,-1,0), nrow = 3)
B <- stereographic(A); B
##      [,1] [,2] [,3] [,4]
## [1,]    2   -2    0    0
## [2,]    0    0    2   -2
## [3,]    1    1    1    1

stereographic_inv(B)
##      [,1] [,2] [,3] [,4]
## [1,]    1   -1    0    0
## [2,]    0    0    1   -1
## [3,]    0    0    0    0

stereographic_inv(c(2,0,2)) # not correct: z = 2
##      [,1]
## [1,]  1.0
## [2,]  0.0
## [3,]  0.5

## Not run:
# Can be used for optimization with sum(x^2) == 1
# Imagine to maximize the product x*y*z for x^2 + y^2 + z^2 == 1 !
fnObj <- function(x) { # length(x) = 2
  x1 <- stereographic_inv(c(x, 1)) # on S^2
```

```

    return( -prod(x1) )           # Maximize
  }
  sol <- optim(c(1, 1), fnObj)
  -sol$value                     # the maximal product
  ## [1] 0.1924501                # 1/3 * sqrt(1/3)
  stereographic_inv(c(sol$par, 1)) # the solution coordinates
                                # on S^2
  ## [1,] 0.5773374              # by symmetry must be
  ## [2,] 0.5773756              # sqrt(1/3) = 0.5773503...
  ## [3,] 0.5773378
  ## End(Not run)

```

---

str2num

*Converting string to number (Matlab style)*


---

## Description

Functions for converting strings to numbers and numbers to strings.

## Usage

```

str2num(S)
num2str(A, fmt = 3)

```

## Arguments

S	string containing numbers (in Matlab format).
A	numerical vector or matrix.
fmt	format string, or integer indicating number of decimals.

## Details

str2num converts a string containing numbers into a numerical object. The string can begin and end with '[' and ']', numbers can be separated with blanks or commas; a semicolon within the brackets indicates a new row for matrix input. When a semicolon appears behind the braces, no output is shown on the command line.

num2str converts a numerical object, vector or matrix, into a character object of the same size. `fmt` will be a format string for use in `sprintf`, or an integer `n` being used in `'%.nf'`.

## Value

Returns a vector or matrix of the same size, converted to strings, respectively numbers.

## See Also

[sprintf](#)

**Examples**

```
str1 <- " [1 2 3; 4, 5, 6; 7,8,9] "  
str2num(str1)  
# matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)  
  
# str2 <- " [1 2 3; 45, 6; 7,8,9] "  
# str2num(str2)  
# Error in str2num(str2) :  
# All rows in Argument 's' must have the same length.  
  
A <- matrix(c(pi, 0, exp(1), 1), 2, 2)  
B <- num2str(A, 2); b <- dim(B)  
B <- as.numeric(B); dim(B) <- b  
B  
#      [,1] [,2]  
# [1,] 3.14 2.72  
# [2,] 0.00 1.00
```

---

strcat

*String Concatenation*

---

**Description**

Concatenate all strings in a character vector

**Usage**

```
strcat(s1, s2 = NULL, collapse = "")
```

**Arguments**

s1	character string or vectors
s2	character string or vector, or NULL (default)
collapse	character vector of length 1 (at best a single character)

**Details**

Concatenate all strings in character vector s1, if s2 is NULL, or cross-concatenate all string elements in s1 and s2 using collapse as 'glue'.

**Value**

a character string or character vector

**See Also**

[paste](#)

**Examples**

```
strcat(c("a", "b", "c"))           #=> "abc"  
strcat(c("a", "b"), c("1", "2"), collapse="x") #=> "ax1" "ax2" "bx1" "bx2"
```

---

strcmp	<i>String Comparison</i>
--------	--------------------------

---

**Description**

Compare two strings or character vectors for equality.

**Usage**

```
strcmp(s1, s2)  
strcmpi(s1, s2)
```

**Arguments**

s1, s2            character strings or vectors

**Details**

For strcmp comparisons are case-sensitive, while for strcmpi the are case-insensitive. Leading and trailing blanks do count.

**Value**

logical, i.e. TRUE if s1 and s2 have the same length as character vectors and all elements are equal as character strings, else FALSE.

**See Also**

[strcat](#)

**Examples**

```
strcmp(c("yes", "no"), c("yes", "no"))  
strcmpi(c("yes", "no"), c("Yes", "No"))
```



---

strfind	<i>Find Substrings</i>
---------	------------------------

---

**Description**

Find substrings within strings of a character vector.

**Usage**

```
strfind(s1, s2, overlap = TRUE)
strfindi(s1, s2, overlap = TRUE)

findstr(s1, s2, overlap = TRUE)
```

**Arguments**

s1	character string or character vector
s2	character string (character vector of length 1)
overlap	logical (are overlapping substrings allowed)

**Details**

strfind finds positions of substrings within s1 that match exactly with s2, and is case sensitive; no regular patterns.

strfindi does not distinguish between lower and upper case.

findstr should only be used as internal function, in Matlab it is deprecated. It searches for the shorter string within the longer one.

**Value**

Returns a vector of indices, or a list of such index vectors if s2 is a character vector of length greater than 1.

**See Also**

[strcmp](#)

**Examples**

```
S <- c("", "ab", "aba", "aba aba", "abababa")
s <- "aba"
strfind(S, s)
strfindi(toupper(S), s)
strfind(S, s, overlap = FALSE)
```

strjust                      *Justify character vector*

---

### Description

Justify the strings in a character vector.

### Usage

```
strjust(s, justify = c("left", "right", "center"))
```

### Arguments

s	Character vector.
justify	Whether to justify left, right, or centered.

### Details

strjust(s) or strjust(s, justify = ``right'`) returns a right-justified character vector. All strings have the same length, the length of the longest string in s — but the strings in s have been trimmed before.

strjust(s, justify = ``left'`) does the same, with all strings left-justified.

strjust(s, justify = ``centered'`) returns all string in s centered. If an odd number of blanks has to be added, one blank more is added to the left than to the right.

### Value

A character vector of the same length.

### See Also

[strTrim](#)

### Examples

```
S <- c("abc", "letters", "1", "2 2")
strjust(S, "left")
```

---

strRep	<i>Find and replace substring</i>
--------	-----------------------------------

---

**Description**

Find and replace all occurrences of a substring with another one in all strings of a character vector.

**Usage**

```
strRep(s, old, new)
```

**Arguments**

s	Character vector.
old	String to be replaced.
new	String that replaces another one.

**Details**

Replaces all occurrences of old with new in all strings of character vector s. The matching is case sensitive.

**Value**

A character vector of the same length.

**See Also**

[gsub](#), [regexprep](#)

**Examples**

```
S <- c('This is a good example.', "He has a good character.",  
      'This is good, good food.', "How goodgood this is!")  
strRep(S, 'good', 'great')
```

---

strTrim	<i>Remove leading and trailing white space.</i>
---------	---

---

### Description

Removes leading and trailing white space from a string.

### Usage

```
strTrim(s)
deblank(s)
```

### Arguments

s                    character string or character vector

### Details

strTrim removes leading and trailing white space from a string or from all strings in a character vector.

deblank removes trailing white space only from a string or from all strings in a character vector.

### Value

A character string or character vector with (leading and) trailing white space.

### See Also

[strjust](#)

### Examples

```
s <- c(" abc", "abc  ", " abc ", " a b c ", "abc", "a b c")
strTrim(s)
deblank(s)
```

---

subspace	<i>Angle between two subspaces</i>
----------	------------------------------------

---

**Description**

Finds the angle between two subspaces.

**Usage**

```
subspace(A, B)
```

**Arguments**

A, B            Numeric matrices; vectors will be considered as column vectors. These matrices must have the same number of rows.

**Details**

Finds the angle between two subspaces specified by the columns of A and B.

**Value**

An angle in radians.

**Note**

It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

**References**

Strang, G. (1998). Introduction to Linear Algebra. Wellesley-Cambridge Press.

**See Also**

[orth](#)

**Examples**

```
180 * subspace(c(1, 2), c(2, 1)) / pi  #=> 36.87
180 * subspace(c(0, 1), c(1, 2)) / pi  #=> 26.565

H <- hadamard(8)
A <- H[, 2:4]
B <- H[, 5:8]
subspace(A, B)  #=> 1.5708 or pi/2, i.e. A and B are orthogonal
```

---

`sumalt`*Alternating Series Acceleration*

---

**Description**

Computes the value of an (infinite) alternating sum applying an acceleration method found by Cohen et al.

**Usage**

```
sumalt(f_alt, n)
```

**Arguments**

`f_alt` a function of  $k=0 \dots \text{Inf}$  that returns element  $a_k$  of the infinite alternating series.  
`n` number of elements of the series used for calculating.

**Details**

Computes the sum of an alternating series (whose entries are strictly decreasing), applying the acceleration method developed by H. Cohen, F. Rodriguez Villegas, and Don Zagier.

For example, to compute the Leibniz series (see below) to 15 digits exactly,  $10^{15}$  summands of the series will be needed. This acceleration approach here will only need about 20 of them!

**Value**

Returns an approximation of the series value.

**Author(s)**

Implemented by Hans W Borchers.

**References**

Henri Cohen, F. Rodriguez Villegas, and Don Zagier. Convergence Acceleration of Alternating Series. *Experimental Mathematics*, Vol. 9 (2000).

**See Also**

[aitken](#)

**Examples**

```

# Beispiel: Leibniz-Reihe 1 - 1/3 + 1/5 - 1/7 +- ...
a_pi4 <- function(k) (-1)^k / (2*k + 1)
sumalt(a_pi4, 20) # 0.7853981633974484 = pi/4 + eps()

# Beispiel: Van Wijngaarden transform needs 60 terms
n <- 60; N <- 0:n
a <- cumsum((-1)^N / (2*N+1))
for (i in 1:n) {
  a <- (a[1:(n-i+1)] + a[2:(n-i+2)]) / 2
}
a - pi/4 # 0.7853981633974483

# Beispiel: 1 - 1/2^2 + 1/3^2 - 1/4^2 +- ...
b_alt <- function(k) (-1)^k / (k+1)^2
sumalt(b_alt, 20) # 0.8224670334241133 = pi^2/12 + eps()

## Not run:
# Dirichlet eta() function: eta(s) = 1/1^s - 1/2^s + 1/3^s +- ...
eta_ <- function(s) {
  eta_alt <- function(k) (-1)^k / (k+1)^s
  sumalt(eta_alt, 30)
}
eta_(1) # 0.6931471805599453 = log(2)
abs(eta_(1+1i) - eta(1+1i)) # 1.24e-16

## End(Not run)

```

---

taylor

*Taylor Series Approximation*


---

**Description**

Local polynomial approximation through Taylor series.

**Usage**

```
taylor(f, x0, n = 4, ...)
```

**Arguments**

f	differentiable function.
x0	point where the series expansion will take place.
n	Taylor series order to be used; should be $n \leq 8$ .
...	more variables to be passed to function f.

**Details**

Calculates the first four coefficients of the Taylor series through numerical differentiation and uses some polynomial 'yoga'.

**Value**

Vector of length n+1 representing a polynomial of degree n.

**Note**

TODO: Pade approximation.

**See Also**

[fderiv](#)

**Examples**

```
taylor(sin, 0, 4) #=> -0.1666666 0.0000000 1.0000000 0.0000000
taylor(exp, 1, 4) #=> 0.04166657 0.16666673 0.50000000 1.00000000 1.00000000

f <- function(x) log(1+x)
p <- taylor(f, 0, 4)
p
# log(1+x) = 0 + x - 1/2 x^2 + 1/3 x^3 - 1/4 x^4 +- ...
# [1] -0.250004 0.333334 -0.500000 1.000000 0.000000

## Not run:
x <- seq(-1.0, 1.0, length.out=100)
yf <- f(x)
yp <- polyval(p, x)
plot(x, yf, type = "l", col = "gray", lwd = 3)
lines(x, yp, col = "red")
grid()
## End(Not run)
```

---

tic,toc

*MATLAB timer functions*

---

**Description**

Provides stopwatch timer. Function tic starts the timer and toc updates the elapsed time since the timer was started.

**Usage**

```
tic(gcFirst=FALSE)
toc(echo=TRUE)
```



**Arguments**

gcFirst            logical scalar. If TRUE, perform garbage collection prior to starting stopwatch  
echo               logical scalar. If TRUE, print elapsed time to screen

**Details**

Provides analog to `system.time`. Function `toc` can be invoked multiple times in a row.

**Value**

`toc` invisibly returns the elapsed time as a named scalar (vector).

**Author(s)**

P. Roebuck <proebuck@mdanderson.org>

**Examples**

```
tic()
for(i in 1:100) mad(runif(1000)) # kill time
toc()
```

---

titanium

*Titanium Test Data*

---

**Description**

The Titanium data set describes measurements of a certain property of titanium as a function of temperature.

**Usage**

```
data(titanium)
```

**Format**

The format is:  
Two columns called 'x' and 'y', the first being the temperature.

**Details**

These data have become a standard test for data fitting since they are hard to fit by classical techniques and have a significant amount of noise.

**Source**

Boor, C. de, and J. R. Rice (1968). Least squares cubic spline approximation II – Variable knots, CSD TR 21, Comp.Sci., Purdue Univ.

**Examples**

```
## Not run:  
data(titanium)  
plot(titanium)  
grid()  
## End(Not run)
```

---

Toeplitz

*Toeplitz Matrix*

---

**Description**

Generate Toeplitz matrix from column and row vector.

**Usage**

```
Toeplitz(a, b)
```

**Arguments**

a                    vector that will be the first column  
b                    vector that if present will form the first row.

**Details**

Toeplitz(a, b) returns a (non-symmetric) Toeplitz matrix whose first column is a and whose first row is b. The following rows are shifted to the left.

If the first element of b differs from the last element of a it is overwritten by this one (and a warning sent).

**Value**

Matrix of size (length(a), length(b)).

**Note**

stats::Toeplitz does not allow to specify the row vector, that is returns only the *symmetric* Toeplitz matrix.

**See Also**

[hankel](#)

**Examples**

```
Toeplitz(c(1, 2, 3, 4, 5))  
Toeplitz(c(1, 2, 3, 4, 5), c(1.5, 2.5, 3.5, 4.5, 5.5))
```

---

Trace	<i>Matrix trace</i>
-------	---------------------

---

**Description**

Sum of the main diagonal elements.

**Usage**

Trace(a)

**Arguments**

a                    a square matrix

**Details**

Sums the elements of the main diagonal of areal or complrx square matrix.

**Value**

scalar value

**Note**

The corresponding function in Matlab/Octave is called trace(), but this in R has a different meaning.

**See Also**

[Diag, diag](#)

**Examples**

```
Trace(matrix(1:16, nrow=4, ncol=4))
```

---

trapz	<i>Trapezoidal Integration</i>
-------	--------------------------------

---

**Description**

Compute the area of a function with values y at the points x.

**Usage**

```
trapz(x, y)  
cumtrapz(x, y)
```

```
trapzfun(f, a, b, maxit = 25, tol = 1e-07, ...)
```

**Arguments**

<code>x</code>	x-coordinates of points on the x-axis
<code>y</code>	y-coordinates of function values
<code>f</code>	function to be integrated.
<code>a, b</code>	lower and upper border of the integration domain.
<code>maxit</code>	maximum number of steps.
<code>tol</code>	tolerance; stops when improvements are smaller.
<code>...</code>	arguments passed to the function.

**Details**

The points  $(x, 0)$  and  $(x, y)$  are taken as vertices of a polygon and the area is computed using `polyarea`. This approach matches exactly the approximation for integrating the function using the trapezoidal rule with basepoints  $x$ .

`cumtrapz` computes the cumulative integral of  $y$  with respect to  $x$  using trapezoidal integration.  $x$  and  $y$  must be vectors of the same length, or  $x$  must be a vector and  $y$  a matrix whose first dimension is `length(x)`.

Inputs  $x$  and  $y$  can be complex.

`trapzfun` realizes trapezoidal integration and stops when the difference from one step to the next is smaller than tolerance (or the of iterations get too big). The function will only be evaluated once on each node.

**Value**

Approximated integral of the function, discretized through the points  $x, y$ , from  $\min(x)$  to  $\max(x)$ . Or a matrix of the same size as  $y$ .

`trapzfun` returns a list with components `value` the value of the integral, `iter` the number of iterations, and `rel.err` the relative error.

**See Also**

[polyarea](#)

**Examples**

```
# Calculate the area under the sine curve from 0 to pi:
n <- 101
x <- seq(0, pi, len = n)
y <- sin(x)
trapz(x, y) #=> 1.999835504

# Use a correction term at the boundary: -h^2/12*(f'(b)-f'(a))
h <- x[2] - x[1]
ca <- (y[2]-y[1]) / h
cb <- (y[n]-y[n-1]) / h
trapz(x, y) - h^2/12 * (cb - ca) #=> 1.999999969
```

```
# Use two complex inputs
z <- exp(1i*pi*(0:100)/100)
ct <- cumtrapz(z, 1/z)
ct[101]                                #=> 0+3.14107591i

f <- function(x) x^(3/2)                #
trapzfun(f, 0, 1)                       #=> 0.4 with 11 iterations
```

---

tri

*Triangular Matrices (Matlab Style)*

---

## Description

Extract lower or upper triangular part of a matrix.

## Usage

```
tril(M, k = 0)
triu(M, k = 0)
```

## Arguments

M                    numeric matrix.  
k                    integer, indicating a secondary diagonal.

## Details

**tril**  
Returns the elements on and below the kth diagonal of X, where k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

**triu**  
Returns the elements on and above the kth diagonal of X, where k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

## Value

Matrix the same size as the input matrix.

## Note

For k=0 it is simply an application of the R functions `lower.tri` resp. `upper.tri`.

## See Also

[Diag](#)

**Examples**

```
tril(ones(4,4), +1)
#  1  1  0  0
#  1  1  1  0
#  1  1  1  1
#  1  1  1  1
```

```
triu(ones(4,4), -1)
#  1  1  1  1
#  1  1  1  1
#  0  1  1  1
#  0  0  1  1
```

---

`trigApprox`*Trigonometric Approximation*

---

**Description**

Computes the trigonometric series.

**Usage**

```
trigApprox(t, x, m)
```

**Arguments**

<code>t</code>	vector of points at which to compute the values of the trigonometric approximation.
<code>x</code>	data from $t=0$ to $t=2*(n-1)*\pi/n$ .
<code>m</code>	degree of trigonometric regression.

**Details**

Calls `trigPoly` to get the trigonometric coefficients and then sums the finite series.

**Value**

Vector of values the same length as `t`.

**Note**

TODO: Return an approximating function instead.

**See Also**

[trigPoly](#)

**Examples**

```
## Not run:
## Example: Gauss' Pallas data (1801)
asc <- seq(0, 330, by = 30)
dec <- c(408, 89, -66, 10, 338, 807, 1238, 1511, 1583, 1462, 1183, 804)
plot(2*pi*asc/360, dec, pch = "+", col = "red", xlim = c(0, 2*pi), ylim = c(-500, 2000),
     xlab = "Ascension [radians]", ylab = "Declination [minutes]",
     main = "Gauss' Pallas Data")
grid()
points(2*pi*asc/360, dec, pch = "o", col = "red")
ts <- seq(0, 2*pi, len = 100)
xs <- trigApprox(ts, dec, 1)
lines(ts, xs, col = "black")
xs <- trigApprox(ts, dec, 2)
lines(ts, xs, col = "blue")
legend(3, 0, c("Trig. Regression of degree 1", "Trig. Regression of degree 2",
              "Astronomical position"), col = c("black", "blue", "red"),
      lty = c(1,1,0), pch = c("","",""+))
## End(Not run)
```

trigPoly

*Trigonometric Polynomial***Description**

Computes the trigonometric coefficients.

**Usage**

```
trigPoly(x, m)
```

**Arguments**

x                    data from  $t=0$  to  $t=2*(n-1)*\pi/n$ .  
m                    degree of trigonometric regression.

**Details**

Compute the coefficients of the trigonometric series of degree  $m$ ,

$$a_0 + \sum_k (a_k \cos(kt) + b_k \sin(kt))$$

by applying orthogonality relations.

**Value**

Coefficients as a list with components  $a_0$ ,  $a$ , and  $b$ .

**Note**

For irregular spaced data or data not covering the whole period, use standard regression techniques, see examples.

**References**

Fausett, L. V. (2007). Applied Numerical Analysis Using Matlab. Second edition, Prentice Hall.

**See Also**

[trigApprox](#)

**Examples**

```
# Data available only from 0 to pi/2
t <- seq(0, pi, len=7)
x <- 0.5 + 0.25*sin(t) + 1/3*cos(t) - 1/3*sin(2*t) - 0.25*cos(2*t)

# use standard regression techniques
A <- cbind(1, cos(t), sin(t), cos(2*t), sin(2*t))
ab <- qr.solve(A, x)
ab
# [1] 0.5000000 0.3333333 0.2500000 -0.2500000 -0.3333333
ts <- seq(0, 2*pi, length.out = 100)
xs <- ab[1] + ab[2]*cos(ts) +
      ab[3]*sin(ts) + ab[4]*cos(2*ts) + ab[5]*sin(2*ts)

## Not run:
# plot to make sure
plot(t, x, col = "red", xlim=c(0, 2*pi), ylim=c(-2,2),
      main = "Trigonometric Regression")
lines(ts, xs, col="blue")
grid()
## End(Not run)
```

---

triquad

*Gaussian Triangle Quadrature*


---

**Description**

Numerically integrates a function over an arbitrary triangular domain by computing the Gauss nodes and weights.

**Usage**

```
triquad(f, x, y, n = 10, tol = 1e-10, ...)
```



**Arguments**

f	the integrand as function of two variables.
x	x-coordinates of the three vertices of the triangle.
y	y-coordinates of the three vertices of the triangle.
n	number of nodes.
tol	relative tolerance to be achieved.
...	additional parameters to be passed to the function.

**Details**

Computes the  $N^2$  nodes and weights for a triangle with vertices given by  $3 \times 2$  vector. The nodes are produced by collapsing the square to a triangle.

Then  $f$  will be applied to the nodes and the result multiplied left and right with the weights (i.e., Gaussian quadrature).

By default, the function applies Gaussian quadrature with number of nodes  $n=10, 21, 43, 87, 175$  until the relative error is smaller than the tolerance.

**Value**

The integral as a scalar.

**Note**

A small relative tolerance is *not* really indicating a small absolute tolerance.

**Author(s)**

Copyright (c) 2005 Greg von Winckel Matlab code based on the publication mentioned and available from MatlabCentral (calculates nodes and weights). Translated to R (with permission) by Hans W Borchers.

**References**

Lyness, J. N., and R. Cools (1994). A Survey of Numerical Cubature over Triangles. Proceedings of the AMS Conference "Mathematics of Computation 1943–1993", Vancouver, CA.

**See Also**

[quad2d](#), [simpson2d](#)

**Examples**

```
x <- c(-1, 1, 0); y <- c(0, 0, 1)
f1 <- function(x, y) x^2 + y^2
(I <- triquad(f1, x, y))                    # 0.3333333333333333

# split the unit square
x1 <- c(0, 1, 1); y1 <- c(0, 0, 1)
```

```
x2 <- c(0, 1, 0); y2 <- c(0, 1, 1)
f2 <- function(x, y) exp(x + y)
I <- triquad(f2, x1, y1) + triquad(f2, x2, y2) # 2.952492442012557
quad2d(f2, 0, 1, 0, 1) # 2.952492442012561
simpson2d(f2, 0, 1, 0, 1) # 2.952492442134769
dblquad(f2, 0, 1, 0, 1) # 2.95249244201256
```

---



*Tridiagonal Linear System Solver*


---

### Description

Solves tridiagonal linear systems  $A*x=rhs$  efficiently.

### Usage

```
trisolve(a, b, d, rhs)
```

### Arguments

a	diagonal of the tridiagonal matrix A.
b, d	upper and lower secondary diagonal of A.
rhs	right hand side of the linear system $A*x=rhs$ .

### Details

Solves tridiagonal linear systems  $A*x=rhs$  by applying Givens transformations.

By only storing the three diagonals, `trisolve` has memory requirements of  $3*n$  instead of  $n^2$  and is faster than the standard `solve` function for larger matrices.

### Value

Returns the solution of the tridiagonal linear system as vector.

### Note

Has applications for spline approximations and for solving boundary value problems (ordinary differential equations).

### References

Gander, W. (1992). *Computermathematik*. Birkhaeuser Verlag, Basel.

### See Also

[qrSolve](#)

**Examples**

```

set.seed(8237)
a <- rep(1, 100)
e <- runif(99); f <- rnorm(99)
x <- rep(seq(0.1, 0.9, by = 0.2), times = 20)
A <- diag(100) + Diag(e, 1) + Diag(f, -1)
rhs <- A %*% x
s <- trisolve(a, e, f, rhs)
s[1:10]          #=> 0.1 0.3 0.5 0.7 0.9 0.1 0.3 0.5 0.7 0.9
s[91:100]       #=> 0.1 0.3 0.5 0.7 0.9 0.1 0.3 0.5 0.7 0.9

```

vander

*Vandermonde matrix***Description**

Generate Vandermonde matrix from a numeric vector.

**Usage**

```
vander(x)
```

**Arguments**

x                    Numeric vector

**Details**

Generates the usual Vandermonde matrix from a numeric vector, e.g. applied when fitting a polynomial to given points. Complex values are allowed.

**Value**

Vandermonde matrix of dimension n where n = length(x).

**Examples**

```
vander(c(1:10))
```

---

**vectorfield***Vector Field Plotting*

---

**Description**

Plotting a vector field

**Usage**

```
vectorfield(fun, xlim, ylim, n = 16,  
            scale = 0.05, col = "green", ...)
```

**Arguments**

fun	function of two variables — must be vectorized.
xlim	range of x values.
ylim	range of y values.
n	grid size, proposed 16 in each direction.
scale	scales the length of the arrows.
col	arrow color, proposed 'green'.
...	more options presented to the arrows primitive.

**Details**

Plots a vector field for a function  $f$ . Main usage could be to plot the solution of a differential equation into the same graph.

**Value**

Opens a graph window and plots the vector field.

**See Also**

[quiver](#), [arrows](#)

**Examples**

```
f <- function(x, y) x^2 - y^2  
xx <- c(-1, 1); yy <- c(-1, 1)  
## Not run:  
vectorfield(f, xx, yy, scale = 0.1)  
for (xs in seq(-1, 1, by = 0.25)) {  
  sol <- rk4(f, -1, 1, xs, 100)  
  lines(sol$x, sol$y, col="darkgreen")  
}  
grid()  
## End(Not run)
```

---

`whittaker`*Whittaker Smoothing*

---

**Description**

Smoothing of time series using the Whittaker-Henderson approach.

**Usage**

```
whittaker(y, lambda = 1600, d = 2)
```

**Arguments**

<code>y</code>	signal to be smoothed.
<code>lambda</code>	smoothing parameter (rough $50 \cdot 10^4$ smooth); the default value of 1600 has been recommended in the literature.
<code>d</code>	order of differences in penalty (generally 2)

**Details**

The Whittaker smoother family was first presented by Whittaker in 1923 for life tables, based on penalized least squares. These ideas were revived by Paul Eilers, Leiden University, in 2003. This approach is also known as Whittaker-Henderson smoothing.

The smoother attempts to both fit a curve that represents the raw data, but is penalized if subsequent points vary too much. Mathematically it is a large, but sparse optimization problem that can be expressed in a few lines of Matlab or R code.

**Value**

A smoothed time series.

**Note**

This is a version that avoids package 'SparseM'.

**Author(s)**

An R version, based on Matlab code by P. Eilers in 2002, has been published by Nicholas Lewin-Koh on the R-help mailing list in Feb. 2004, and in private communication to the author of this package.

**References**

P. H. C. Eilers (2003). A Perfect Smoother. *Analytical Chemistry*, Vol. 75, No. 14, pp. 3631–3636.  
Wilson, D. I. (2006). The Black Art of Smoothing. *Electrical and Automation Technology*, June/July issue.

**See Also**

[supsmu](#), [savgol](#), [ptw:whit2](#)

**Examples**

```
# **Sinosoid test function**
ts <- sin(2*pi*(1:1000)/200)
t1 <- ts + rnorm(1000)/10
t3 <- whittaker(t1, lambda = 1600)
## Not run:
plot(1:1000, t1, col = "grey")
lines(1:1000, ts, col="blue")
lines(1:1000, t3, col="red")
## End(Not run)
```

---

wilkinson

*wilkinson Matrix*

---

**Description**

Generate the Wilkinson matrix of size  $n \times n$ . The Wilkinson matrix for testing eigenvalue computations

**Usage**

```
wilkinson(n)
```

**Arguments**

n                    integer

**Details**

The Wilkinson matrix for testing eigenvalue computations is a symmetric matrix with three non-zero diagonals and with several pairs of nearly equal eigenvalues.

**Value**

matrix of size  $n \times n$

**Note**

The two largest eigenvalues of `wilkinson(21)` agree to 14, but not 15 decimal places.

**See Also**

[Toeplitz](#)

**Examples**

```
(a <- wilkinson(7))  
eig(a)
```

---

zeta

*Riemann Zeta Function*

---

**Description**

Riemann's zeta function valid in the entire complex plane.

**Usage**

```
zeta(z)
```

**Arguments**

z                    Real or complex number or a numeric or complex vector.

**Details**

Computes the zeta function for complex arguments using a series expansion for Dirichlet's eta function.

Accuracy is about 7 significant digits for  $\text{abs}(z) < 50$ , drops off with higher absolute values.

**Value**

Returns a complex vector of function values.

**Note**

Copyright (c) 2001 Paul Godfrey for a Matlab version available on Mathwork's Matlab Central under BSD license.

**References**

Zhang, Sh., and J. Jin (1996). Computation of Special Functions. Wiley-Interscience, New York.

**See Also**

[gammaz](#), [eta](#)

**Examples**

```
## First zero on the critical line  $s = 0.5 + i t$ 
## Not run:
x <- seq(0, 20, len=1001)
z <- 0.5 + x*1i
fr <- Re(zeta(z))
fi <- Im(zeta(z))
fa <- abs(zeta(z))
plot(x, fa, type="n", xlim = c(0, 20), ylim = c(-1.5, 2.5),
     xlab = "Imaginary part (on critical line)", ylab = "Function value",
     main = "Riemann's Zeta Function along the critical line")
lines(x, fr, col="blue")
lines(x, fi, col="darkgreen")
lines(x, fa, col = "red", lwd = 2)
points(14.1347, 0, col = "darkred")
legend(0, 2.4, c("real part", "imaginary part", "absolute value"),
      lty = 1, lwd = c(1, 1, 2), col = c("blue", "darkgreen", "red"))
grid()
## End(Not run)
```



# Index

## \* arith

- agmean, 14
- angle, 21
- bits, 37
- ceil, 49
- combs, 58
- eps, 89
- gcd, lcm, 140
- mod, rem, 222
- nchoosek, 228
- nextpow2, 240
- nthroot, 244
- perms, 258
- pow2, 279
- primes, 282
- randcomb, 303
- randperm, 305

## \* array

- accumarray, 12
- blkdiag, 38
- charpoly, 50
- compan, 58
- cond, 61
- cross, 68
- crossn, 69
- Diag, 80
- distmat, 82
- dot, 83
- eig, 84
- eye, 97
- givens, 143
- gramSchmidt, 149
- hessenberg, 156
- householder, 166
- hypot, 170
- ifft, 171
- inv, 182
- isposdef, 185
- itersolve, 187

- kron, 191
- lu, 215
- magic, 217
- meshgrid, 218
- ndims, 229
- nearest\_spd, 230
- nnz, 241
- Norm, 242
- normest, 243
- nullspace, 245
- numel, 247
- orth, 252
- pinv, 260
- poly2str, 265
- procrustes, 283
- qrSolve, 288
- Rank, 306
- repmat, 315
- Reshape, 316
- size, 343
- sorting, 345
- sortrows, 347
- squareform, 352
- subspace, 365
- Toeplitz, 370
- Trace, 371
- tri, 373
- trisolve, 378

## \* datasets

- brown72, 40
- titanium, 369

## \* fitting

- akimaInterp, 17
- circlefit, 55
- cubicspline, 70
- curvefit, 71
- fornberg, 122
- kriging, 189
- lsqlin, 208

- lsqnonlin, 211
- mexpfit, 219
- odregress, 251
- ppfit, 279
- ppval, 281
- ratinterp, 308
- rationalfit, 309
- \* **geom**
  - haversine, 155
  - inpolygon, 172
  - segm\_distance, 330
  - segm\_intersect, 331
- \* **graphs**
  - andrewsplot, 20
  - errorbar, 91
  - ezcontour, ezsurf, ezmesh, 98
  - ezplot, 99
  - ezpolar, 101
  - figure, 106
  - peaks, 257
  - plotyy, 261
  - polar, 263
  - quiver, 300
  - semilogx, semilogy, 333
  - vectorfield, 380
- \* **logic**
  - and, or, 19
  - findintervals, 107
  - finds, 110
  - isempty, 184
- \* **manip**
  - flipdim, 114
  - histc, 161
  - linspace, 206
  - logspace, 207
  - Mode, 223
  - rot90, 326
  - stereographic, 356
  - str2num, 358
- \* **math**
  - aitken, 16
  - arclength, 26
  - arnoldi, 28
  - barylag, 29
  - barylag2d, 31
  - bernstein, 34
  - bisect, 35
  - brentDekker, 39
  - broyden, 41
  - bsxfun, 43
  - cart2sph, 47
  - chebApprox, 51
  - chebCoeff, 52
  - chebPoly, 53
  - clenshaw\_curtis, 57
  - complexstep, 59
  - cot, csc, sec, etc., 63
  - cotes, 64
  - coth, csch, sech, etc., 65
  - dblquad, 74
  - deg2rad, 78
  - detrend, 78
  - eigjacobi, 85
  - expm, 96
  - fact, 102
  - factors, 103
  - fderiv, 104
  - findzeros, 111
  - fnorm, 121
  - fractalcurve, 124
  - fzero, 129
  - fzsolve, 130
  - gauss\_kronrod, 139
  - gaussHermite, 133
  - gaussLaguerre, 134
  - gaussLegendre, 136
  - gaussNewton, 137
  - gmres, 144
  - grad, 146
  - gradient, 147
  - halley, 151
  - hausdorff\_dist, 154
  - hessian, 157
  - Hessian utilities, 158
  - horner, 165
  - integral, 174
  - integral2, 176
  - interp1, 179
  - interp2, 180
  - isprime, 186
  - jacobian, 188
  - laguerre, 193
  - laplacian, 196
  - lebesgue, 197
  - legendre, 198
  - line\_integral, 202

- linearproj, affineproj, 199
- mldivide, 221
- muller, 226
- neville, 233
- newtonHorner, 235
- newtonInterp, 237
- newtonRaphson, 238
- newtonsys, 239
- numderiv, 246
- pade, 253
- pchip, 255
- piecewise, 259
- Poly, 264
- polyadd, 266
- polyApprox, 267
- polyarea, 268
- polyder, 270
- polyfit, polyfix, 271
- polyint, 273
- polylog, 273
- polymul, polydiv, 275
- polypow, 276
- polytrans, polygcf, 277
- polyval, polyvalm, 278
- quad, 289
- quad2d, 290
- quadcc, 291
- quadgk, 292
- quadgr, 293
- quadinf, 294
- quadl, 295
- quadv, 299
- randortho, 304
- rat, 307
- rectint, 311
- ridders, 316
- romberg, 323
- roots, polyroots, 324
- rref, 327
- runge, 328
- simpadpt, 339
- simpson2d, 340
- sind, cosd, tand, etc., 341
- spinterp, 348
- sqrtm, rootm, 350
- sumalt, 366
- taylor, 367
- trapz, 371
- trigApprox, 374
- trigPoly, 375
- triquad, 376
- \* ode**
  - abm3pc, 11
  - bulirsch-stoer, 44
  - bvp, 46
  - cranknic, 66
  - deeve, 77
  - deval, 79
  - euler\_heun, 93
  - newmark, 234
  - ode23, 248
  - rk4, rk4sys, 319
  - rkf54, 320
  - shooting, 334
- \* optimize**
  - anms, 22
  - fibsearch, 105
  - findmins, 108
  - fletcher\_powell, 112
  - fminbnd, 115
  - fmincon, 116
  - fminsearch, 118
  - fminunc, 119
  - fsolve, 127
  - geo\_median, 142
  - golden\_ratio, 145
  - hooke\_jeeves, 163
  - L1linreg, 192
  - linprog, 203
  - lsqlincon, 210
  - nelder\_mead, 231
  - quadprog, 297
  - shubert, 335
  - softline, 344
  - steep\_descent, 355
- \* package**
  - pracma-package, 8
- \* specfun**
  - bernoulli, 32
  - ellipke, ellipj, 87
  - eta, 92
  - expint, 94
  - fresnelS/C, 126
  - gammainc, 131
  - gammaz, 132
  - lambertWp, 194

- Si, Ci, 337
- zeta, 383
- \* **specmat**
  - hadamard, 150
  - hankel, 153
  - hilb, 160
  - moler, 224
  - pascal, 255
  - rosser, 325
  - vander, 379
  - wilkinson, 382
- \* **stat**
  - erf, 90
  - geomean, harmmean, 141
  - poisson2disk, 262
  - rand, 301
  - rmserr, 322
  - std, 353
  - std\_err, 354
- \* **string**
  - blanks, 38
  - refindall, 312
  - regex, 313
  - regexprep, 314
  - strcat, 359
  - strcmp, 360
  - strfind, 361
  - strjust, 362
  - strRep, 363
  - strTrim, 364
- \* **timeseries**
  - approx\_entropy, 24
  - conv, 62
  - cutpoints, 73
  - deconv, 76
  - findpeaks, 109
  - hampel, 152
  - histss, 162
  - hurstexp, 168
  - invlap, 183
  - movavg, 225
  - savgol, 329
  - whittaker, 381
- \* **utilities**
  - cd, pwd, what, 48
  - clear, who(s), ver, 56
  - disp, beep, 81
  - fprintf, 123
  - tic, toc, 368
- abm3pc, 11
- abs, 21
- accumarray, 12
- acosd (sind, cosd, tand, etc.), 341
- acot (cot, csc, sec, etc.), 63
- acotd (sind, cosd, tand, etc.), 341
- acoth (coth, csch, sech, etc.), 65
- acsc (cot, csc, sec, etc.), 63
- acscd (sind, cosd, tand, etc.), 341
- acsch (coth, csch, sech, etc.), 65
- affineproj (linearproj, affineproj), 199
- agmean, 14
- aitken, 16, 366
- akimaInterp, 17, 190
- and (and, or), 19
- and, or, 19
- andrewsplot, 20
- angle, 21
- anms, 22
- approx, 180
- approx\_entropy, 24
- arclength, 26, 269
- arnoldi, 28
- arrayfun (bsxfun), 43
- arrows, 301, 380
- asec (cot, csc, sec, etc.), 63
- asecd (sind, cosd, tand, etc.), 341
- asech (coth, csch, sech, etc.), 65
- asind (sind, cosd, tand, etc.), 341
- atan2d (sind, cosd, tand, etc.), 341
- atand (sind, cosd, tand, etc.), 341
- barylag, 29, 31, 197, 233, 237
- barylag2d, 31, 190
- beep (disp, beep), 81
- bernoulli, 32
- bernstein, 34
- bernsteinb (bernstein), 34
- bisect, 35
- bits, 37
- blanks, 38
- blkdiag, 38
- brent, 130, 317
- brent (brentDekker), 39
- brentDekker, 39
- brown72, 40
- broyden, 41, 128, 240

- bsxfun, 43
- bubbleSort (sorting), 345
- bulirsch-stoer, 44
- bulirsch\_stoer (bulirsch-stoer), 44
- bvp, 46, 335
  
- cart2pol (cart2sph), 47
- cart2sph, 47
- cd (cd, pwd, what), 48
- cd, pwd, what, 48
- ceil, 49
- charpoly, 50
- chebApprox, 51, 53, 54, 267
- chebCoeff, 52, 54
- chebPoly, 53, 53, 198
- choose, 228
- Ci (Si, Ci), 337
- circlefit, 55, 72
- circshift (flipdim), 114
- clear (clear, who(s), ver), 56
- clear, who(s), ver, 56
- clenshaw\_curtis, 57
- combs, 58, 303
- compan, 58, 84
- complexstep, 59, 247
- cond, 61, 243
- contour, 99
- conv, 62, 76, 266
- cosd (sind, cosd, tand, etc.), 341
- cot (cot, csc, sec, etc.), 63
- cot, csc, sec, etc., 63
- cotd (sind, cosd, tand, etc.), 341
- cotes, 64
- coth (coth, csch, sech, etc.), 65
- coth, csch, sech, etc., 65
- cranknic, 66, 94, 235
- cross, 68, 69, 84
- crossn, 68, 69
- csc (cot, csc, sec, etc.), 63
- cscd (sind, cosd, tand, etc.), 341
- csch (coth, csch, sech, etc.), 65
- cubicspline, 70, 281
- cumtrapz (trapz), 371
- curve, 101
- curvefit, 71
- cut, 74
- cutpoints, 73
  
- dblquad, 74, 341
  
- deblank, 38
- deblank (strTrim), 364
- deconv, 63, 76
- deeve, 77, 80
- deg2rad, 78
- detrend, 78
- deval, 77, 79, 249, 319
- Diag, 39, 80, 98, 371, 373
- diag, 81, 371
- dim, 343
- disp (disp, beep), 81
- disp, beep, 81
- dist, 83, 353
- distmat, 82, 154
- dot, 68, 69, 83
  
- eig, 84, 85
- eigjacob, 85
- einsteinF, 86
- ellipj (ellipke, ellipj), 87
- ellipke (ellipke, ellipj), 87
- ellipke, ellipj, 87
- eps, 89
- erf, 90
- erfc (erf), 90
- erfcinv (erf), 90
- erfcx (erf), 90
- erfi (erf), 90
- erfinv (erf), 90
- erfz (erf), 90
- errorbar, 91
- eta, 92, 383
- euler\_heun, 93
- expint, 94, 337
- expint\_E1 (expint), 94
- expint\_Ei (expint), 94
- expm, 96, 352
- eye, 97
- ezcontour (ezcontour, ezsurf, ezmesh), 98
- ezcontour, ezsurf, ezmesh, 98
- ezmesh (ezcontour, ezsurf, ezmesh), 98
- ezplot, 99, 101
- ezpolar, 101
- ezsurf (ezcontour, ezsurf, ezmesh), 98
  
- fact, 102
- factorial, 102
- factorial2 (fact), 102
- factors, 103, 186, 282

- fderiv, 104, 147, 148, 247, 368
- fft, 172
- fftshift (ifft), 171
- fibsearch, 105, 116
- figure, 106
- find, 241
- findInterval, 161
- findintervals, 107
- findmins, 108, 111, 336
- findpeaks, 109, 152
- finds, 110
- findstr (strfind), 361
- findzeros, 111
- Fix (ceil), 49
- fletcher\_powell, 112, 355
- flipdim, 114
- fliplr (flipdim), 114
- flipud (flipdim), 114
- fminbnd, 115
- fmincon, 116, 120
- fminsearch, 117, 118, 120
- fminunc, 117, 119
- fnorm, 121, 329
- fornberg, 122
- fplot (ezplot), 99
- fprintf, 123
- fractalcurve, 124
- fresnelC (fresnelS/C), 126
- fresnelS (fresnelS/C), 126
- fresnelS/C, 126
- fsolve, 42, 127
- fzero, 129
- fzsolve, 130
  
- gamma, 132, 133
- gammainc, 131
- gammaz, 93, 132, 383
- gauss\_kronrod, 57, 139
- gaussHermite, 133, 135, 136
- gaussLaguerre, 134, 134, 136
- gaussLegendre, 57, 127, 134, 135, 136
- gaussNewton, 128, 137, 344
- gcd (gcd, lcm), 140
- gcd, lcm, 140
- geo\_median, 142
- geomean (geomean, harmmean), 141
- geomean, harmmean, 141
- getwd, 49
- givens, 143, 150, 167
  
- gmres, 144
- golden\_ratio, 116, 145
- grad, 146
- grad\_csd (complexstep), 59
- gradient, 147
- gramSchmidt, 149
- gsub, 314, 363
  
- hadamard, 150, 153
- halley, 151, 195
- hampel, 110, 152
- hankel, 150, 153, 370
- harmmean (geomean, harmmean), 141
- hausdorff\_dist, 154
- haversine, 155
- heapSort (sorting), 345
- hessdiag, 158
- hessdiag (Hessian utilities), 158
- hessenberg, 29, 156
- hessian, 157, 159, 196
- Hessian utilities, 158
- hessian\_csd (complexstep), 59
- hessvec, 158
- hessvec (Hessian utilities), 158
- hilb, 160
- hist, 161, 163
- histc, 161, 163
- histss, 161, 162
- hooke\_jeeves, 119, 163, 232
- horner, 165
- hornerdefl (horner), 165
- householder, 144, 150, 157, 166, 288
- humps, 167
- hurstexp, 168
- hypot, 170
  
- idivide (mod, rem), 222
- ifft, 171
- ifftshift (ifft), 171
- Imag (angle), 21
- image, 99
- incgam (gammainc), 131
- inpolygon, 172
- insertionSort (sorting), 345
- integral, 174, 178, 202
- integral2, 176
- integral3 (integral2), 176
- integrate, 75, 290, 295, 324
- interp1, 179, 181, 256

- interp2, [31](#), [180](#)
- inv, [182](#)
- invlap, [183](#)
- is.sorted (sorting), [345](#)
- isempty, [184](#)
- isposdef, [185](#)
- isprime, [103](#), [186](#), [282](#)
- itersolve, [187](#)
- jacobian, [188](#)
- jacobian\_csd (complexstep), [59](#)
- kabsch (procrustes), [283](#)
- kriging, [18](#), [189](#)
- kron, [191](#)
- L1linreg, [143](#), [192](#)
- lagrangeInterp (newtonInterp), [237](#)
- laguerre, [193](#)
- lambertWn (lambertWp), [194](#)
- lambertWp, [17](#), [194](#)
- laplacian, [158](#), [196](#)
- laplacian\_csd (complexstep), [59](#)
- Lcm (gcd, lcm), [140](#)
- lebesgue, [197](#)
- legendre, [198](#)
- li (expint), [94](#)
- line\_integral, [202](#)
- linearproj (linearproj, affineproj), [199](#)
- linearproj, affineproj, [199](#)
- linprog, [203](#)
- linspace, [206](#)
- list.files, [49](#)
- lm, [192](#), [251](#)
- logit (sigmoid), [338](#)
- loglog (semilogx, semilogy), [333](#)
- logm (expm), [96](#)
- logseq (logspace), [207](#)
- logspace, [206](#), [207](#), [207](#)
- ls, [56](#)
- lsqcurvefit (lsqnonlin), [211](#)
- lsqlin, [72](#), [208](#), [211](#)
- lsqlincon, [208](#), [210](#), [298](#)
- lsqnonlin, [192](#), [211](#), [220](#)
- lsqnonneg (lsqnonlin), [211](#)
- lsqsep, [220](#)
- lsqsep (lsqnonlin), [211](#)
- lu, [215](#)
- lu\_crout (lu), [215](#)
- lufact (lu), [215](#)
- lusys (lu), [215](#)
- magic, [217](#)
- matlab, [218](#)
- mean, [141](#)
- median, [223](#)
- mergeOrdered (sorting), [345](#)
- mergeSort (sorting), [345](#)
- meshgrid, [218](#), [257](#)
- mexpfir, [219](#)
- midpoint (bulirsch-stoer), [44](#)
- mkpp, [280](#)
- mkpp (ppval), [281](#)
- mldivide, [221](#)
- Mod, [21](#)
- mod (mod, rem), [222](#)
- mod, rem, [222](#)
- Mode, [223](#)
- moler, [224](#)
- movavg, [225](#)
- mrdivide (mldivide), [221](#)
- muller, [226](#)
- nchoosek, [228](#)
- ndims, [229](#)
- nearest\_spd, [230](#)
- nelder\_mead, [119](#), [164](#), [231](#)
- neville, [123](#), [233](#), [237](#)
- newmark, [67](#), [234](#)
- newton (newtonRaphson), [238](#)
- newtonHorner, [235](#), [238](#)
- newtonInterp, [123](#), [233](#), [237](#)
- newtonRaphson, [40](#), [151](#), [227](#), [236](#), [238](#), [240](#)
- newtonsys, [42](#), [130](#), [138](#), [227](#), [239](#)
- nextpow2, [37](#), [240](#), [279](#)
- nlm, [213](#)
- nls, [213](#)
- nnz, [241](#)
- Norm, [121](#), [242](#)
- norm, [242](#)
- normest, [62](#), [243](#)
- nthroot, [244](#)
- null (nullspace), [245](#)
- nullspace, [200](#), [208](#), [245](#), [253](#), [306](#)
- num2str (str2num), [358](#)
- numderiv, [60](#), [105](#), [246](#)
- numdiff (numderiv), [246](#)
- numel, [247](#)

ode23, [12](#), [45](#), [67](#), [235](#), [248](#), [319](#), [321](#)  
 ode23s, [45](#)  
 ode23s (ode23), [248](#)  
 ode45 (ode23), [248](#)  
 ode78 (ode23), [248](#)  
 odregress, [251](#)  
 ones (eye), [97](#)  
 optim, [23](#)  
 optimize, [108](#)  
 or (and, or), [19](#)  
 orth, [200](#), [245](#), [252](#), [365](#)  
 outer, [219](#)  
  
 pade, [253](#), [308](#)  
 pascal, [255](#)  
 paste, [359](#)  
 pchip, [255](#)  
 pchipfun (pchip), [255](#)  
 pdist (distmat), [82](#)  
 pdist2 (distmat), [82](#)  
 peaks, [257](#)  
 perms, [58](#), [258](#), [305](#)  
 persp, [99](#)  
 pgamma, [132](#)  
 piecewise, [259](#)  
 pinv, [208](#), [260](#)  
 plot, [333](#)  
 plotyy, [261](#)  
 pnorm, [90](#)  
 poisson2disk, [262](#)  
 pol2cart (cart2sph), [47](#)  
 polar, [21](#), [263](#)  
 Poly, [264](#)  
 poly, [272](#), [278](#)  
 poly2str, [265](#)  
 poly\_center (polyarea), [268](#)  
 poly\_crossings (polyarea), [268](#)  
 poly\_length, [26](#)  
 poly\_length (polyarea), [268](#)  
 polyadd, [63](#), [266](#)  
 polyApprox, [52](#), [267](#)  
 polyarea, [268](#), [311](#), [372](#)  
 polyder, [270](#), [273](#)  
 polydiv (polymul, polydiv), [275](#)  
 polyfit, [79](#), [267](#)  
 polyfit (polyfit, polyfix), [271](#)  
 polyfit, polyfix, [271](#)  
 polyfix (polyfit, polyfix), [271](#)  
 polygcf (polytrans, polygcf), [277](#)

polygon, [173](#)  
 polyint, [271](#), [273](#)  
 polylog, [273](#)  
 polymul, [76](#), [276](#)  
 polymul (polymul, polydiv), [275](#)  
 polymul, polydiv, [275](#)  
 polypow, [276](#)  
 polyroot, [325](#)  
 polyroots (roots, polyroots), [324](#)  
 polytrans (polytrans, polygcf), [277](#)  
 polytrans, polygcf, [277](#)  
 polyval, [166](#), [265](#), [271–273](#), [277](#)  
 polyval (polyval, polyvalm), [278](#)  
 polyval, polyvalm, [278](#)  
 polyvalm (polyval, polyvalm), [278](#)  
 pow2, [241](#), [279](#)  
 ppfit, [279](#)  
 ppval, [280](#), [281](#)  
 pracma (pracma-package), [8](#)  
 pracma-package, [8](#)  
 primes, [95](#), [103](#), [186](#), [282](#)  
 procrustes, [230](#), [283](#)  
 psi, [285](#)  
 psinc (humps), [167](#)  
 pwd (cd, pwd, what), [48](#)  
  
 qpsolve (qpspecial, qpsolve), [286](#)  
 qpspecial (qpspecial, qpsolve), [286](#)  
 qpspecial, qpsolve, [286](#)  
 qr, [216](#)  
 qr.solve, [327](#)  
 qrSolve, [188](#), [288](#), [378](#)  
 quad, [289](#), [291](#), [296](#), [300](#), [340](#)  
 quad2d, [75](#), [290](#), [341](#), [377](#)  
 quadcc, [291](#)  
 quadgk, [139](#), [175](#), [292](#), [295](#)  
 quadgr, [175](#), [293](#), [324](#)  
 quadinf, [175](#), [294](#)  
 quadl, [290](#), [295](#)  
 quadprog, [297](#)  
 quadv, [175](#), [299](#)  
 quickSort (sorting), [345](#)  
 quickSortx (sorting), [345](#)  
 quiver, [300](#), [380](#)  
  
 rad2deg (deg2rad), [78](#)  
 rand, [301](#)  
 randcomb, [58](#), [303](#)  
 randi (rand), [301](#)



- randn (rand), 301
- randortho, 230, 304
- randp (rand), 301
- randperm, 258, 303, 305
- rands (rand), 301
- randsample (rand), 301
- Rank, 245, 306
- rat, 307
- ratinterp, 308, 310
- rationalfit, 308, 309
- rats (rat), 307
- Real (angle), 21
- rectint, 311
- refindall, 312
- regexp, 312, 313
- regexp\_i (regexp), 313
- regexpr, 313
- regexprprep, 314
- regulaFalsi (bisection), 35
- rem (mod, rem), 222
- repmat, 315
- Reshape, 315, 316
- ridders, 36, 40, 316
- rk4, 12, 321
- rk4 (rk4, rk4sys), 319
- rk4, rk4sys, 319
- rk4sys, 249
- rk4sys (rk4, rk4sys), 319
- rkf54, 320
- rm, 56
- rmserr, 322
- romberg, 139, 175, 323
- rootm (sqrtm, rootm), 350
- roots, 59, 194, 265, 278
- roots (roots, polyroots), 324
- roots, polyroots, 324
- rootsmult (roots, polyroots), 324
- rosser, 325
- rot90, 326
- rref, 327
- runge, 328
- sample\_entropy (approx\_entropy), 24
- savgol, 329, 382
- sec (cot, csc, sec, etc.), 63
- secant, 227
- secant (bisection), 35
- secd (sind, cosd, tand, etc.), 341
- sech (coth, csch, sech, etc.), 65
- segm\_distance, 330, 332
- segm\_intersect, 331, 331
- selectionSort (sorting), 345
- semilogx (semilogx, semilogy), 333
- semilogx, semilogy, 333
- semilogy (semilogx, semilogy), 333
- seq, 206, 207
- sessionInfo, 56
- set.seed, 302
- setwd, 49
- shellSort (sorting), 345
- shooting, 46, 334
- shubert, 335
- Si (Si, Ci), 337
- Si, Ci, 337
- sigmoid, 338
- signm (sqrtm, rootm), 350
- simpadpt, 65, 175, 339
- simpson2d, 75, 340, 340, 377
- sinc, 337
- sinc (humps), 167
- sind (sind, cosd, tand, etc.), 341
- sind, cosd, tand, etc., 341
- size, 229, 248, 343
- softline, 138, 344
- solve, 145, 182
- sort, 346, 347
- sorting, 345
- sortrows, 347
- sph2cart (cart2sph), 47
- spinterp, 348
- spline, 70, 180
- sprintf, 124, 358
- sqrt, 244
- sqrtm (sqrtm, rootm), 350
- sqrtm, rootm, 350
- squareform, 352
- std, 353
- std\_err, 354
- steep\_descent, 113, 355
- stereographic, 356
- stereographic\_inv (stereographic), 356
- str2num, 358
- strcat, 359, 360
- strcmp, 360, 361
- strcmp\_i (strcmp), 360
- strfind, 361
- strfind\_i (strfind), 361

strjust, 362, 364  
strRep, 363  
strTrim, 362, 364  
subspace, 365  
sumalt, 366  
supsmu, 382  
svd, 62, 243, 284  
system.time, 369

tand (sind, cosd, tand, etc.), 341  
taylor, 105, 254, 367  
testSort (sorting), 345  
tic (tic, toc), 368  
tic, toc, 368  
titanium, 369  
toc (tic, toc), 368  
Toeplitz, 150, 153, 370, 382  
Trace, 81, 371  
trapz, 65, 259, 269, 371  
trapzfun (trapz), 371  
tri, 373  
trigApprox, 374, 376  
trigPoly, 374, 375  
tril (tri), 373  
trimmean (geomean, harmmean), 141  
triplequad (dblquad), 74  
triquad, 376  
trisolve, 378  
triu (tri), 373

uniq (accumarray), 12  
unique, 13  
uniroot, 106, 130, 146

vander, 160, 379  
vectorfield, 301, 380  
Vectorize, 43  
ver (clear, who(s), ver), 56

what (cd, pwd, what), 48  
whittaker, 330, 381  
who (clear, who(s), ver), 56  
whos (clear, who(s), ver), 56  
wilkinson, 224, 326, 382

zeros (eye), 97  
zeta, 33, 93, 383