Package ‘pracma’

April 9, 2019

Type Package
Version 2.2.5
Date 2019-04-08
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
Author Hans W. Borchers [aut, cre]
Maintainer Hans W. Borchers <hborchers@googlemail.com>
Repository CRAN

R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>pracma-package</td>
<td>8</td>
</tr>
<tr>
<td>abm3pc</td>
<td>11</td>
</tr>
<tr>
<td>accumarray</td>
<td>13</td>
</tr>
<tr>
<td>agmean</td>
<td>15</td>
</tr>
<tr>
<td>aitken</td>
<td>16</td>
</tr>
</tbody>
</table>
akimaInterp ................................................. 18
and, or .................................................. 19
andrewsplot ............................................. 20
angle ...................................................... 21
anms ....................................................... 22
approx_entropy ........................................... 24
arclength ................................................ 26
arnoldi .................................................... 28
barylag ................................................... 29
barylag2d ............................................... 31
bermoulli ............................................... 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 ............................................... 54
clear, who(s), ver ..................................... 56
crakns .curtis .......................................... 57
combs ................................................... 58
compan .................................................. 58
complexstep ............................................ 59
cond ..................................................... 61
conv ..................................................... 62
cot,csc,sec, etc. ....................................... 63
cotes ..................................................... 64
coth,csc,sech, etc. .................................... 65
cranknic ............................................... 66
cross .................................................... 68
crossn .................................................. 69
cubic spline ........................................... 70
curvehfit ............................................... 71
cutpoints ............................................... 73
dblquad .................................................. 74
deconv .................................................. 76
topics documented:

deve ................................................................. 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
fxsolve .......................................................... 130
gammainc ......................................................... 131
R topics documented:
gammaz .......................................................... 132
gaussHermite ................................................... 133
gaussLaguerre .................................................. 134
gaussLegendre .................................................. 136
gaussNewton .................................................... 137
gauss_kronrod .................................................. 139
gcd, lcm .......................................................... 140
geo_median, harmmean .......................................... 141
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
hilb ............................................................... 158
histe ............................................................... 159
histss ........................................................... 160
hooke_jeeves .................................................. 161
horner ............................................................. 163
householder ..................................................... 165
humps .............................................................. 166
hurstexp ........................................................ 167
hypot .............................................................. 169
ifft ................................................................. 170
inpolygon ........................................................ 171
integral .......................................................... 172
integral2 ........................................................ 174
interp1 ............................................................ 177
interp2 ............................................................ 179
inv ................................................................. 180
invlap ............................................................. 181
isempty ........................................................ 183
isposdef ........................................................ 184
isprime .......................................................... 185
itersolve ....................................................... 186
jacobian .......................................................... 187
kriging ............................................................ 188
kron ............................................................... 189
L.1linreg ........................................................ 190
<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>laguerre</td>
<td>192</td>
</tr>
<tr>
<td>lambertWp</td>
<td>193</td>
</tr>
<tr>
<td>laplacian</td>
<td>194</td>
</tr>
<tr>
<td>lebesgue</td>
<td>196</td>
</tr>
<tr>
<td>legendre</td>
<td>197</td>
</tr>
<tr>
<td>lineaproj, affineproj</td>
<td>198</td>
</tr>
<tr>
<td>line_integral</td>
<td>201</td>
</tr>
<tr>
<td>linprog</td>
<td>202</td>
</tr>
<tr>
<td>linspace</td>
<td>205</td>
</tr>
<tr>
<td>logspace</td>
<td>206</td>
</tr>
<tr>
<td>lsqlin</td>
<td>207</td>
</tr>
<tr>
<td>lsqlincon</td>
<td>209</td>
</tr>
<tr>
<td>lsqnonlin</td>
<td>210</td>
</tr>
<tr>
<td>lu</td>
<td>214</td>
</tr>
<tr>
<td>magic</td>
<td>216</td>
</tr>
<tr>
<td>matlab</td>
<td>217</td>
</tr>
<tr>
<td>meshgrid</td>
<td>217</td>
</tr>
<tr>
<td>mexpfit</td>
<td>218</td>
</tr>
<tr>
<td>mldivide</td>
<td>220</td>
</tr>
<tr>
<td>mod, rem</td>
<td>221</td>
</tr>
<tr>
<td>Mode</td>
<td>222</td>
</tr>
<tr>
<td>moler</td>
<td>223</td>
</tr>
<tr>
<td>movavg</td>
<td>224</td>
</tr>
<tr>
<td>muller</td>
<td>225</td>
</tr>
<tr>
<td>nchoosek</td>
<td>227</td>
</tr>
<tr>
<td>ndims</td>
<td>228</td>
</tr>
<tr>
<td>nearest_spd</td>
<td>229</td>
</tr>
<tr>
<td>nelder_mead</td>
<td>230</td>
</tr>
<tr>
<td>neville</td>
<td>232</td>
</tr>
<tr>
<td>newmark</td>
<td>233</td>
</tr>
<tr>
<td>newtonHorner</td>
<td>234</td>
</tr>
<tr>
<td>newtonInterp</td>
<td>236</td>
</tr>
<tr>
<td>newtonRaphson</td>
<td>237</td>
</tr>
<tr>
<td>newtonsys</td>
<td>238</td>
</tr>
<tr>
<td>nextpow2</td>
<td>239</td>
</tr>
<tr>
<td>nile</td>
<td>240</td>
</tr>
<tr>
<td>nnz</td>
<td>241</td>
</tr>
<tr>
<td>Norm</td>
<td>242</td>
</tr>
<tr>
<td>normest</td>
<td>243</td>
</tr>
<tr>
<td>nthroot</td>
<td>244</td>
</tr>
<tr>
<td>nullspace</td>
<td>245</td>
</tr>
<tr>
<td>numberinv</td>
<td>246</td>
</tr>
<tr>
<td>numel</td>
<td>247</td>
</tr>
<tr>
<td>ode23</td>
<td>248</td>
</tr>
<tr>
<td>odregress</td>
<td>251</td>
</tr>
<tr>
<td>orth</td>
<td>253</td>
</tr>
<tr>
<td>pade</td>
<td>254</td>
</tr>
<tr>
<td>pascal</td>
<td>255</td>
</tr>
</tbody>
</table>
### Topics Documented

- pchip .......................... 256
- 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 ................................ 275
- polypow ................................ 276
- polytrans ................................ 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
- quadfit ................................ 294
- quadl ................................ 296
- quadprog ................................ 297
- quadv ................................ 299
- quiver ................................ 301
- rand ................................ 302
- randcomb ................................ 304
- randortho ................................ 305
- randperm ................................ 306
- Rank ................................ 307
- rat ................................ 308
- ratinterp ................................ 309
- rationalfit ................................ 310
- rectint ................................ 312
R topics documented:

- refindall ........................................... 313
- regexp ............................................. 314
- regexprep ......................................... 315
- repmat ............................................. 316
- Reshape ........................................... 317
- ridders ............................................ 317
- rk4, rk4sys ......................................... 320
- rkf54 ............................................... 321
- rmserr ............................................. 323
- romberg ............................................ 324
- roots ............................................... 325
- rosser ............................................. 326
- rot90 ............................................... 327
- ref .................................................. 327
- runge ............................................... 329
- savgol ............................................. 330
- segm_distance .................................... 331
- segm_intersect ................................... 332
- 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
- sqrtm, rootm ....................................... 350
- squareform ........................................ 352
- std ................................................ 353
- std_err ........................................... 354
- steep_descent .................................... 355
- str2num .......................................... 356
- strcat ............................................. 357
- strcmp ............................................. 358
- strfind .......................................... 359
- strjust .......................................... 360
- strRep ............................................ 361
- strTrim ........................................... 362
- subspace .......................................... 363
- sumalt ............................................. 364
- taylor .............................................. 365
- tic, toc ........................................... 366
- titanium .......................................... 367
## pracma-package

<table>
<thead>
<tr>
<th>Description</th>
<th>Practical Numerical Math Routines</th>
</tr>
</thead>
</table>

This package provides R implementations of more advanced functions in numerical analysis, with a special view on 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, Padé 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 sys-
tem.

Besides that, many of these functions could be called in R applications as they do not have compa-
rable 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, atan, atan2d,
beep, bernoulli, blank, blkdiag, bspfun,
cart2pol, cart2sph, cd, ceil, circshift, clear, compan, cond, conv,
cos, cot, cotd, coth, cross, csc, csd, 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,
gamma, gci, geo, geomean, gmres, gradient,
hadamard, hankel, harmmean, hilb, histc, humps, hypot,
ldivide, ifft, ifftshift, inpolygon, integral, integral2, integral3,
interp1, interp2, inv, isempty, isprime,
kron,
legendre, linprog, linspace, loglog, logm, logseq, logspace, lsqcurvefit,
lsqmin, lsqnonlin, lsqnonneg, lu,
magic, meshgrid, mkpp, mldivide, mod, mrdivide,
ncache, ndims, nextpow2, nnz, normest, nthroot, null, num2str, numel,
ode23, ode23s, ones, or, orth,
pascal, pchip, pdist, pdist2, peaks, perms, piecewise, pinv, plotyy,
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):

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

**Author(s)**

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

**References**


See Also

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

Examples

```r
## 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

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

\( f \) function in the differential equation \( y' = f(x, y) \).

\( a, b \) endpoints of the interval.

\( y_0 \) starting values at point \( a \).

\( n \) the number of steps from \( a \) to \( b \).

\( \ldots \) 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 \( \text{est.error} \) that should be looked at with caution.

Note

This function serves demonstration purposes only.

References


See Also

\texttt{rk4, ode23}

Examples

```R
## 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)
```
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

b               vector of unique elements of a.
m               vector of indices such that b = a[m]
n               vector of indices such that a = b[n]

Note

The Matlab function accumarray can also handle sparse matrices.
See Also
unique

Examples

## Examples for accumarray

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

```r
val = 101:105
subs <- matrix(c(1,2,2,2, 1,1,3,1,3, 1,2,2,2,2), ncol = 3)
accumarray(subs, val)
# [,1] [,2] [,3]
# [1,] 101  0  0
# [2,]  0  0  0
# [,1] [,2] [,3]
# [1,] 0  0  0
# [2,] 206 0 208
```

```r
val = 101:105
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,]  0  1  0  0
# [2,]  0  0  0  0
# [3,]  0  0  0  0
# [4,]  2  0  0  0
```

```r
val = 101:105
subs <- matrix(c(1, 1, 2, 1, 2, 3, 1, 2, 2, 3, 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

```r
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 6 7 3 4 8
```

```r
A <- uniq(a, first = TRUE); A
# A$m 1 3 4 5 6 8 9 12
```

## Example: Subset sum problem

```r
# Distribution of unique sums among all combinations of a vectors.
allsums <- function(a) {
```
agmean

\begin{verbatim}
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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
\end{verbatim}

---

\textbf{agmean} \hspace{2cm} \textit{Arithmetic-geometric Mean}

\section*{Description}

The arithmetic-geometric mean of real or complex numbers.

\section*{Usage}

\texttt{agmean(a, b)}

\section*{Arguments}

\begin{itemize}
\item \texttt{a, b} \hspace{1cm} vectors of real or complex numbers of the same length (or scalars).
\end{itemize}

\section*{Details}

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

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

\section*{Value}

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

\section*{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

http://mathworld.wolfram.com/Arithmetic-GeometricMean.html

See Also

Arithmetic, geometric, and harmonic mean.

Examples

```r
## Accuracy test: Gauss constant
1/agmean(1, sqrt(2))$agm - 0.834626841674073186  # 1.1e-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)
```

<table>
<thead>
<tr>
<th>aitken</th>
<th></th>
<th>Aitken' Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 accelerate Newton-Raphson (Steffensen’s method).

References


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 xi 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
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, 10.5, 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)
iy <- akimaInterp(x, y, xi)
lines(xi, iy, 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 l 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

```r
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)
```

Description

Plots Andrews’ curves in cartesian or polar coordinates.

Usage

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

Arguments

- `A`: numeric matrix with at least two columns.
- `f`: factor or integer vector with `nrow(A)` elements.
- `style`: character variable, only possible values ‘cart’ or ‘pol’.
- `scaled`: logical; if true scales each column to have mean 0 and standard deviation 1 (not yet implemented).
- `npts`: 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


See Also

polar, andrews::andrews

Examples

```r
## 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


See Also

- `optim`

Examples

```r
## 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)
}
```
approx_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 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


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 <- seq(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")
arclength

Arc Length of a Curve

Description

Calculates the arc length of a parametrized curve.

Usage

\texttt{arclength}(f, a, b, nmax = 20, tol = 1e-05, ...)

Arguments

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

Details

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

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

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

Value

Returns a list with components \texttt{length} the calculated arc length, \texttt{niter} the number of iterations, and \texttt{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 \texttt{polylength} below.

Author(s)

HwB <hwborchers@googlemail.com>
See Also

poly_length

Examples

```r
## Example: parametrized 3D-curve with t in 0..3*pi
f <- function(t) c(sin(2*pi*t), cos(t), t)
arclength(f, 0, 3*pi)
# $length: 17.22033  # true length 17.22032...

## Example: length of the sine curve
f <- function(t) c(t, sin(t))
arclength(f, 0, pi)  # true length 3.8209...

## 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*pi*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 Ferma\'s spiral
#--  _____________________________________________________________
# Fermat\'s spiral: r = a * sqrt(t)
f <- function(t) 0.25 * sqrt(t) * c(cos(t), sin(t))
```

arnoldi <- 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^2 q, ....
H  a Hessenberg matrix such that $A = Q * H * t(Q)$. 

barylag

References

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 0.000000 0.000000
## [2,]  0 0.998738 0.0502159
## [3,]  0 0.0502159 0.9987384
##
## $H
##   [,1]   [,2]   [,3]
## [1,] -149.0000 -42.20367124 156.316506
## [2,]  537.6783  512.55114875 -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


See Also

Lagrange or Newton interpolation.

Examples

```r
## 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
```
## Description

Two-dimensional barycentric Lagrange interpolation.

## Usage

```r
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 `length(xf)`-by-`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


## See Also

- `interp2`, `barylag`
Examples

```r
## 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)
# # 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:
xl <- linspace(0.3, 0.7, 2)
yl <- bernoulli(1, xl)
plot(xl, yl, type='l', col='red', lwd=2,

# xlab="", ylab="", main="Bernoulli Polynomials")
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

Examples

```r
## Example
f <- function(x) sin(2*pi*x)
x <- 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:
x <- 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 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


See Also

`ridders`

Examples

```r
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
```
**bits**

**Binary Representation**

**Description**

Literal bit representation.

**Usage**

```plaintext
bits(x, k = 54, pos_sign = FALSE, break0 = FALSE)
```

**Arguments**

- **x**: a positive or negative floating point number.
- **k**: number of binary digits after the decimal point.
- **pos_sign**: logical; shall the '+' sign be included.
- **break0**: logical; shall trailing zeros be included.

**Details**

The literal bit/binary representation of a floating point number is computed by subtracting powers of 2.

**Value**

Returns a string containing the binary representation.

**See Also**

`nextpow2`

**Examples**

```plaintext
bits(2*10)    # "1000000000"
bits(1 + 2^-10) # "1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
blanks  

String of Blank Carakters

Description
Create a string of blank characters.

Usage
blanks(n)

Arguments
n  integer greater or equal to 0.

Details
blanks(n) is a string of n blanks.

Value
String of n blanks.

See Also
deblock

Examples
blanks(6)

blkdiag  

Block Diagonal Matrix

Description
Build a block diagonal matrix.

Usage
blkdiag(...) 

Arguments
... sequence of non-empty, numeric matrices
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
```r
a1 <- matrix(c(1,2), 1)
a2 <- as.matrix(c(1,2))
blkdiag(a1, diag(1, 2, 2), a2)
```

---

**Description**
Find root of continuous function of one variable.

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

**Arguments**
- `fun` function whose root is to be found.
- `a, b` left and right end points of an interval; function values need to be of different sign at the endpoints.
- `maxiter` maximum number of iterations.
- `tol` relative tolerance.
- `...` 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


See Also

ridders, newtonRaphson

Examples

```r
# 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
```

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.

brown72

Brownian Motion

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.”

Source

http://www.bearcave.com/misl/misl_tech/wavelets/hurst/

Examples

```r
## 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**

```r
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 \( x0 \).
- **...** 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


See Also

newtonsys, fsolve

Examples

```r
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

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. z0 = 0.2555 + 0.8948i
# fnorm: 7.284374e-10
# niter: 13

broyden(F3, c(0, 0))
# $zero: 0.5671433 0.5671433
# $x: exp(-x)

F4 <- function(x) # Dennis Schnabel
```
bsxfun

Elementwise Function Application (Matlab Style)

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 pa-
rameters:
mapply(f, x, y, z, ...)
but the array structure will not be preserved in this case.

See Also

Vectorize
Examples

```r
X <- matrix(rep(1:10, each = 10), 10, 10)
Y <- t(X)
bsxfun("^", 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

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

```r
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 Richardson 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 \( t0 \) to \( tfinal \) 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 \( t \in [t0, tfinal] \) 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


See Also

ode23, ode23s

Examples

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

## Not run:
S <- ode23(f1, 0, rt, c(0.0, 1.0))
S$y[nrow(S$y), 1] # 2.5e-11
S$y[nrow(S$y), 1] # -7.1e-04

## Example: \( y' = -200 \times y^2 \) # \( y(x) = 1 / (1 + 100 \times x^2) \)
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)
```
**bvp**  
*Boundary Value Problems*

**Description**

Solves boundary value problems of linear second order differential equations.

**Usage**

\[ \text{bvp}(f, g, h, x, y, n = 50) \]

**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 \( \text{list}(xs, ys) \) 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 \texttt{qr.solve} for large values of \( n \).

**References**


**See Also**

- \texttt{shooting}
Examples

```r
## Solve y'' = 2x/(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
x <- c(0.0, 4.0)
y <- c(1.25, -0.95)
sol <- bvpc(x, f1, f2, f3, x, y)
## Not run:
plot(sol$xx[, 2], sol$ys[, 1], ylim = c(-2, 2),
xlab = "", ylab = "", main = "Boundary Value Problem")
# The analytic solution is
sfun <- function(x) 1.25 + 0.4868896526*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)
```

Description

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

Usage

```r
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

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
\[
x \leftarrow 0.5 \cdot \cos(\pi/6); \quad y \leftarrow 0.5 \cdot \sin(\pi/6); \quad z \leftarrow \sqrt{1 - x^2 - y^2}
\]
\[
(s \leftarrow \text{cart2sph}(c(x, y, z))) \quad \# \ 0.5235988 \ 1.0471976 \ 1.0000000
\]
sph2cart(s) \quad \# \ 0.4330127 \ 0.2500000 \ 0.8660254
\[
cart2pol(c(1,1)) \quad \# \ 0.7853982 \ 1.4142136
\]
cart2pol(c(1,1,0)) \quad \# \ 0.7853982 \ 1.4142136 \ 0.0000000
\[
pol2cart(c(\pi/2, 1)) \quad \# \ 6.123234e-17 \ 1.0000000e+00
\]
pol2cart(c(\pi/4, 1, 1)) \quad \# \ 0.7071068 \ 0.7071068 \ 1.0000000

## Transform spherical to cylindrical coordinates and vice versa
\[
sph2cyl \leftarrow \text{function}(\text{th.nph.r}) \ \text{cart2pol}(\text{sph2cart}(\text{th.nph.r}))
\]
cyl2sph \leftarrow \text{function}(\text{phi.r.z}) \ \text{cart2sph}(\text{pol2cart}(\text{phi.r.z}))

---

### cd, pwd, what

**Directory Functions (Matlab style)**

**Description**
Displays or changes working directory, or lists files therein.

**Usage**
\[
\text{cd}(\text{dname})
\]
\[
\text{pwd}()
\]
\[
\text{what}(\text{dname} = \text{getwd})()
\]

**Arguments**
\[
\text{dname} \quad \text{(relative or absolute) directory path.}
\]
**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**

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

---

### ceil

**Integer Functions (Matlab Style)**

**Description**

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

**Usage**

```r
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 ml prefix to indicate Matlab style.

The corresponding functions `floor()` (rounding to the largest integer 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

\begin{verbatim}
x <- c(-1.2, -0.8, 0, 0.5, 1.1, 2.9)
ceil(x)
fix(x)
\end{verbatim}

\begin{verbatim}
charpoly

Characteristic Polynomial

Description

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

Usage

\begin{verbatim}
charpoly(a, info = FALSE)
\end{verbatim}

Arguments

\begin{verbatim}
a      quadratic matrix; size should not be much larger than 100.
info   logical; if true, the inverse matrix will also be reported.
\end{verbatim}

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 \texttt{cp}, the characteristic polynomial, \texttt{det}, the determinant, and \texttt{inv}, the inverse matrix, will be returned.

References


Examples

\begin{verbatim}
a <- magic(5)
A <- charpoly(a, info = TRUE)
A$cp
roots(A$cp)
A$det
zapsmall(A$inv %*% a)
\end{verbatim}
Function approximation through Chebyshev polynomials (of the first kind).

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


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 >= 0.

Details

For a function fun on 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” <http://www.maths.ox.ac.uk/chebfun/> by Nick Trefethen.

References


See Also

chebPoly, chebApprox

Examples

```r
## 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
cc <- chebPoly(n)        # summing up the polynomial coefficients
p <- cC %*% cc           # 0 0 1 0 1
```

Description

Chebyshev polynomials and their values.

Usage

`chebPoly(n, x = NULL)`

Arguments

- `n` an integer >= 0.
- `x` a numeric vector, possibly empty; default NULL.

Details

Determines an (n+1)-ny-(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 a \((n+1)\)-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” <http://www.maths.ox.ac.uk/chebfun/> by Nick Trefethen.

References

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, fast = FALSE)
circlefit

Arguments

xp, yp  Vectors representing the x and y coordinates of plane points
fast  logical; shall a fast, non-optimized solution be returned?

Details

This routine first finds an ‘algebraic’ solution based on a linear fit and then calls optim with this solution as starting point. If fast is TRUE the algebraic solution will not be improved.

The value to be minimized is the distance of the given points to the nearest point on the circle.

Value

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

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

Note

May be worth to apply nls instead of optim.

References


Examples

```r
# 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)
circlefit(xp, yp, fast = TRUE) #== 0.9899628 1.0044920 1.0256633
#== 0.9899628 1.0044920 1.0256633
# RMS error: 0.07631986
rslt <- circlefit(xp, yp) #== 0.9965782 1.0009066 1.0240452
#== 0.9965782 1.0009066 1.0240452
# RMS error: 0.07611598

## Not run:
x0 <- rslt[1]; y0 <- rslt[2]; r0 <- rslt[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)
```
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


See Also

gaussLegendre, gauss_kronrod

Examples

```r
## 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
```
Companion Matrix

Description
Computes the companion matrix of a vector.

Usage
companion(p)

Arguments
p vector representing a polynomial
Details

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

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

Value

A square matrix of \( length(p)-1 \) rows and columns

See Also

roots

Examples

\[
p <- c(1, 0, -7, 6)
\]
\[
\text{companion}(p)
\]
\[
\begin{bmatrix}
0 & 7 & -6 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\]

complexstep

Complex Step Derivatives

Description

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

Usage

\[
\text{complexstep}(f, x0, h = 1e-20, \ldots)
\]
\[
\text{grad_csd}(f, x0, h = 1e-20, \ldots)
\]
\[
\text{jacobian_csd}(f, x0, h = 1e-20, \ldots)
\]
\[
\text{hessian_csd}(f, x0, h = 1e-20, \ldots)
\]
\[
\text{laplacian_csd}(f, x0, h = 1e-20, \ldots)
\]

Arguments

\[
f \quad \text{Function that is to be differentiated.}
\]
\[
x0 \quad \text{Point at which to differentiate the function.}
\]
\[
h \quad \text{Step size to be applied; shall be very small.}
\]
\[
\ldots \quad \text{Additional variables to be passed to } f.
\]
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: $0 < h << 1$.

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@goolemail.com>

References


See Also

numderiv

Examples

```r
## 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)
# 4.05342789389868, 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)
```
cond 61

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 -0.00350053 0
## [2,] -0.06350053 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+y+z), 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

---

### 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 cond(M) and cond(M, p) near 1 are indications of a well-conditioned matrix.
Value

\( \text{cond}(M) \) returns the 2-norm condition number, the ratio of the largest singular value of \( M \) to the smallest.

\( c = \text{cond}(M, p) \) returns the matrix condition number in \( p \)-norm:

\[ \text{norm}(X, p) \ast \text{norm}(\text{inv}(X), p). \]

(Not yet implemented.)

Note

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

References


See Also

\text{normest}, \text{svd}

Examples

\[ \text{cond(hilb(8))} \]

---

\textbf{conv} \hspace{1cm} \textit{Polynomial Convolution}

Description

Convolution and polynomial multiplication.

Usage

\text{conv}(x, y)

Arguments

\( x, y \) real or complex vectors.

Details

\( r = \text{conv}(p, q) \) 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
decov, 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.
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**


**See Also**

`simpadpt`, `trapz`

**Examples**

```r
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)  # 0.999999999501637
cotes(sin, 0, pi/2, 20, 6)  # 0.9999999998927507
cotes(sin, 0, pi/2, 20, 7)  # 1.000000000000363 odd degree is better
cotes(sin, 0, pi/2, 20, 8)  # 1.000000000002231
```

**Description**

More hyperbolic functions not available in R.

**Usage**

```r
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
```

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 \( \mathbb{R} \times \mathbb{R}^m \rightarrow \mathbb{R}^m \), 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 \( t_0 \) and \( t_1 \), 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 \texttt{ode23} or \texttt{rkf54}.

References


See Also

\texttt{ode23}, \texttt{newmark}

Examples

```r
## 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")
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
```

```
```r
cross <- cranknic(fhh, 0, 60, c(0, 10))
plot(sol[, 1], sol[, 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**

```r
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**

```r
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

\[
\text{spatproduct} <- \text{function}(a, b, c) \ \text{dot}(a, \ \text{crossn}(b, c))
\]

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

**See Also**

cross, dot

**Examples**

\[
A <- \text{matrix}(c(1, 0, 0, 0, 1, 0), \text{nrow}=2, \text{ncol}=3, \text{byrow}=\text{TRUE})
\]

crossn(A)  # => 0 0 1

\[
\begin{align*}
x & <- c(1.0, 0.0, 0.0) \\
y & <- c(1.0, 0.5, 0.0) \\
z & <- c(0.0, 0.0, 1.0) \\
\text{identical}\left(\text{dot}(x, \ \text{crossn}(\text{rbind}(y, z))), \ \text{det}(\text{rbind}(x, y, z))\right)
\end{align*}
\]
**Description**

Computes the natural interpolation cubic spline.

**Usage**

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

**Arguments**

- `x, y`: x- and y-coordinates of points to be interpolated.
- `xi`: x-coordinates of points at which the interpolation is to be performed.
- `endp2nd`: logical; if true, the derivatives at the endpoints are prescribed by `der`.
- `der`: 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**


**See Also**

`spline`
## Examples

```r
## 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,
     -3.07, -3.17, -3.32, -3.30, -3.22, -3.10)
xp <- seq(-60, 70, by = 1)

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

## 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, xp, endp2nd = TRUE)  # der = 0 at endpoints
lines(xp, ys, col = "red")
ys <- cubicSpline(x, y, xp)  # no endpoint condition
lines(xp, 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**

```r
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 \texttt{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, \texttt{lsqlin}

Examples

```r
## Approximating half circle arc with small perturbations
N <- 50
u <- linspace(0, pi, N)
x <- cos(u) + 0.05 * rdn(1, N)
y <- sin(u) + 0.05 * rdn(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)
```
cutpoints

Find Cutting Points

description
Finds cutting points for vector \( s \) of real numbers.

Usage

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

Arguments

- \( x \): vector of real values.
- \( \text{nmax} \): the maximum number of cutting points to choose
- \( \text{quant} \): 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 \( \text{cutp} \), the cutting points selected, and \( \text{cutd} \), 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

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)
Description

Numerically evaluate double integral over rectangle.

Usage

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

```r
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

```r
f1 <- function(x, y) x^2 + y^2
dblquad(f1, -1, 1, -1, 1)  # 2.66666667, 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)))
```
deconv

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 = 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
Event Detection in ODE solution

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 values 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 x0 at which the event happens.

Note

The interpolation is linear only for the moment.

See Also

deval

Examples

## Damped pendulum: \( y'' = -0.3 \, y' - \sin(y) \)

# \( y1 = y, y2 = y': \ y1' = y2, \ y2' = -0.3 \times y2 - \sin(y1) \)

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**  
```matlab  
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**  
```matlab  
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**  
```matlab  
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

```r
  t <- 1:9
  x <- c(0, 2, 0, 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 values 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

```r
# 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()
```

---

**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**

```r
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)
```

---

**Utility functions (Matlab style)**

**Description**

Display text or array, or produce beep sound.

**Usage**

```r
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 newline.

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 - 2A.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

\[
\text{apply(outer(a, t(b), ","), c(1,4), function(x) sqrt(sum(diag(x*x))))}
\]

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

```r
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)
```

---

**dot**

**Scalar Product**

Description

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

Usage

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

Arguments

- \( x \) numeric vector or matrix
- \( y \) 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)))  \#=> 1.73205

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
eigjacobi

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

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


See Also

eig
Examples
A <- matrix(c(1.06, 0.73, -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,] 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 0.0208098

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:
\[
E_1(x) = \frac{x^2e^x}{(e^x - 1)^2}
\]
\[
E_2(x) = \frac{x}{e^x - 1}
\]
\[
E_3(x) = \ln(1 - e^{-x})
\]
\[
E_4(x) = \frac{x}{e^x - 1} - \ln(1 - e^{-x})
\]
E1 has an inflection point at \( x = 2.34694130 \ldots \)
Value

Numeric/complex scalar or vector.

Examples

```r
# 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)
```

---

**Description**

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

**Usage**

```r
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

\[ u = \int_0^\phi \frac{dt}{\sqrt{1 - m \sin^2 t}} \]

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


See Also

elliptic::sn, cn, dn

Examples

```r
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)

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
```
Floating Point Relative Accuracy

Description

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

Usage

\[ \varepsilon(x = 1.0) \]

Arguments

\[ x \]
scalar or numerical vector or matrix.

Details

\[ d = \varepsilon(x) \]

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

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

Value

Returns a scalar.

Examples

```r
for (i in -5:5) cat(\varepsilon(10^i), "\n")
```

# 1.6940666e-21
# 1.355253e-20
# 2.168404e-19
# 1.734723e-18
# 1.387779e-17
# 2.220446e-16
# 1.776357e-15
# 1.421085e-14
# 1.36868e-13
# 1.81889e-12
# 1.455192e-11
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
errorbar

Examples

```r
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] 0.1572992007050285
# [1] 0.1572992007050285
# [1] 0.1572992007050285
erfz(x)
# [1] 0.842700792949715
erfi(x)
# [1] 1.650425758797543
```

Description

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

Usage

```r
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

```r
etaz)
```

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.
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.
\( y_0 \) starting value at \( a \).
n number of grid points.
improved logical; shall the Heun method be used; default \( \text{TRUE} \).
... 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.
expint

Value

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

References


See Also

cranknic

Examples

```r
## 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 \( \expint(x) \) 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

\begin{align*}
\text{expint}(x) \\
\text{expint}_E1(x) \\
\text{expint}_Ei(x) \\
\text{li}(x)
\end{align*}

Arguments

\begin{itemize}
\item $x$ \hspace{1em} vector of real or complex numbers.
\end{itemize}

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 $li(10^i) - li(2)$ is an approximation of the number of primes below $10^i$, i.e., $\pi(10^i)$, see “?primes”.

References


See Also

\begin{itemize}
\item gsl::expint_E1, expint_Ei, primes
\end{itemize}

Examples

\begin{verbatim}
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:
estimP1 <- function(n) round(Re(li(n) - li(2))) # estimated number of primes
primesP1 <- function(n) length(primes(n)) # true number of primes <= n
N <- 1e6
\end{verbatim}
Matrix Exponential

Description

Computes the exponential of a matrix.

Usage

\[
\text{expm}(A, \text{np} = 128)
\]

\[
\text{logm}(A)
\]

Arguments

- \(A\): numeric square matrix.
- \(\text{np}\): 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.

\(\text{logm}\) 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 \(\text{logm}\) function).

Author(s)

Idea and Matlab code for a cubic root by Nick Trefethen in his “10 digits 1 page” project, for realization see file ‘cuberoOA.m’ at \texttt{http://people.maths.ox.ac.uk/trefethen/tda.html}. 

\[
(\text{estimPi(N) - primesPi(N)}) / \text{estimPi(N)} \quad \# \text{deviation is 0.16 percent!}
\]

\#
# End(Not run)
References


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

Some Basic Matrices

Description

Create basic matrices.

Usage

eye(n, m = n)
one(n, m = n)
zeros(n, m = n)
Arguments

m, n  numeric scalars specifying size of the matrix

Value

Matrix of size n × 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)
one(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

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

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

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

Arguments

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

Description

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

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

```r
## 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)
```
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 mimic 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.
Fact

Factorial Function

Description
Factorial for non-negative integers \( n \leq 170 \).

Usage
\[
\text{fact}(n) \\
\text{factorial2}(n)
\]

Arguments
- \( n \): Vector of integers, for \text{fact}, resp. a single integer for \text{factorial2}.

Details
The factorial is computed by brute force; factorials for \( n > 171 \) are not representable as ‘double’ anymore.

Value
\text{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(\( n \)).
\text{factorial2} returns the product of all even resp. odd integers, depending on whether \( n \) is even or odd.

Note
The R core function \text{factorial} uses the \text{gamma} function, whose implementation is not accurate enough for larger input values.

See Also
\text{factorial}
Examples

```r
fact(c(-1, 0, 1, NA, 171)) # => NaN 1 1 NA Inf
fact(100) # => 9.332621544394411e+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**

```r
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 \( n = \prod \text{factors}(n) \).

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**

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

```r
# End(Not run)
```
Numerical Differentiation

Description

Numerical function differentiation for orders \( n=1 \ldots 4 \) using finite difference approximations.

Usage

\[
fderiv(f, x, n = 1, h = 0, \text{ method } = \text{c("central", "forward", "backward"), ...})
\]

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.
- \( \text{method} \) one of “central”, “forward”, or “backward”.
- \( \ldots \) 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 \( n=1, \ldots, 4 \) and precisions of about 10^-10, 10^-8, 5*10^-7, 5*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

Fibonacci search for function minimum.

**Usage**

\[
\text{fibsearch}(f, a, b, \ldots, \text{endp} = \text{FALSE}, \text{tol} = \text{.Machine}\$\text{double.}\text{eps}^{\cdot}(1/2))
\]

**Arguments**

- \(f\): Function or its name as a string.
- \(a, b\): Endpoints of the interval.
- \(\text{endp}\): Logical; shall the endpoints be considered as possible minima?
- \(\text{tol}\): Absolute tolerance; default \(\text{eps}^{\cdot}(1/2)\).
- \(\ldots\): 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
findintervals

Examples

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

## End(Not run)

---

**findintervals**  
*Find Interval Indices*

**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**

```r
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**

```r
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 “[+]1,[-]1,”; if a pattern is provided, the parameters `nups` and `ndowns` 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 forth 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

```r
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(1:length(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.
findzeros

Value

Indices of elements matching the expression x.

Examples

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

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 \texttt{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 \texttt{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

\texttt{findmins}
Examples

\[ f_1 \left( \frac{x}{\pi} \right) \sin(\pi x) \]
\[ \text{findzeros}(f_1, 1/10, 1) \]
\# 0.100000 0.111108 0.1250183 0.1428641 0.1666655
\# 0.200004 0.2499867 0.3333441 0.499979 1.000000

\[ f_2 \left( \frac{x}{\pi} \right) 0.5(1 + \sin(10\pi x)) \]
\[ \text{findzeros}(f_2, 0, 1) \]
\# 0.15 0.35 0.55 0.75 0.95

\[ f_3 \left( \frac{x}{\pi} \right) \sin(\pi x) + 1 \]
\[ \text{findzeros}(f_3, 0.1, 0.5) \]
\# 0.105232 0.13333333 0.1818182 0.2857143

\[ f_4 \left( \frac{x}{\pi} \right) \sin(\pi x) - 1 \]
\[ \text{findzeros}(f_4, 0.1, 0.5) \]
\# 0.1176471 0.1538462 0.22222222 0.4000000

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

## End(Not run)

---

### 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

\[ \text{fletcher\_powell}(x_0, f, g = \text{NULL}, \maxiter = 1000, \text{tol} = .Machine\$\text{double.\text{eps}}(2/3)) \]

### 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

\[ x_{\text{new}} = x - H^{-1}(x) \text{grad}(g)(x) \]

where \( H \) is the Hessian and \( \text{grad} \) 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:

- \( x_{\text{min}} \) minimum solution found.
- \( f_{\text{min}} \) value of \( f \) at minimum.
- \( n_{\text{iter}} \) number of iterations performed.

Note


References


See Also

steep_descent

Examples

```r
## 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
# $fmin
# [1] 1.774148e-27
# $niter
# [1] 14
```
**Description**

Flip matrices up and down or left and right; or circulating indices per dimension.

**Usage**

```r
flipdim(a, dim)
flipud(a)
fliplr(a)
circshift(a, sz)
```

**Arguments**

- `a`: numeric or complex matrix
- `dim`: flipping dimension; can only be 1 (default) or 2
- `sz`: 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**

```r
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)
```
**Description**

Find minimum of single-variable function on fixed interval.

**Usage**

```r
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**

See Also

fibsearch, golden_ratio

Examples

```matlab
## 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**

```matlab
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 `solnl` 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)**

Xianyan Chen for the package NlcOptim.

**References**


**See Also**

`fminsearch`, `fminunc`.

**Examples**

```r
# 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
```
\[
\text{ub} <- \text{rep}(0.5, n)
\]

# Apply constraint minimization
res <- \text{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**

\[
f\text{minsearch}(fn, x0, ..., lower = NULL, upper = NULL, 
method = c("Nelder-Mead", "Hooke-Jeeves"), 
minimize = TRUE, maxiter = 1000, tol = 1e-08)
\]

**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.


**fminunc**

*Minimize Unconstrained Multivariable Function*

**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**


**See Also**

*melder_mead, hooke_jeves*

**Examples**

```r
# 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

fminsearch(rosena, c(-1.2, 1), a = sqrt(2), method="Hooke-Jeeves")
## $xmin
## $fmin
```

**Description**

Find minimum of unconstrained multivariable functions.

**Usage**

```r
fminunc(x0, fn, gr = NULL, ...,
        tol = 1e-08, maxiter = 0, maxfeval = 0)
```
Arguments

- **x0**: starting point.
- **fn**: objective function to be minimized.
- **gr**: gradient function of the objective.
- **...**: additional parameters to be passed to the function.
- **tol**: relative tolerance.
- **maxiter**: maximum number of iterations.
- **maxfeval**: 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:

- **par**: the best minimum found.
- **value**: function value at the minimum.
- **counts**: number of function and gradient calls.
- **convergence**: integer indicating the terminating situation.
- **message**: 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 Rvmmin), stripped-down version by Hans W. Borchers.

References


See Also

fminsearch, fmincon,

Examples

```r
defun = function(x)
fminunc(x0 = c(1, 2), fun)
## xmin: c(-0.6691, 0.0000); fmin: -0.4052```
Description

The `fnorm` function calculates several different types of function norms for depending on the argument \( p \).

Usage

```plaintext
fnorm(f, g, x1, x2, p = 2, npoints = 100)
```

Arguments

- \( f, g \) functions given by name or string.
- \( x1, x2 \) endpoints of the interval.
- \( p \) Numeric scalar or Inf, -Inf; default is 2.
- \( npoints \) 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 \( \text{norm} \) to their difference.

Especially \( p=\text{inf} \) returns the maximum norm, while \( \text{fnorm}(f, g, x1, x2, p = 1, \text{npoints}) / \text{npoints} \) 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

```r
taxp <- seq(-1, 1, length.out = 6)
typ <- runge(xp)
pt <- polyfit(xp, typ, 5)
f5 <- function(x) polyval(pt, 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, npoints = 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**

```r
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\(\leq\)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 (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

See Also
neville, newtoninterp

Examples
```r
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)
```

```r
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-
fractalcurve

Value

Returns invisibly the number of bytes printed (using nchar).

See Also

sprintf

Examples

```r
## Examples:
 nbytes <- nchar("Results are:\n", file = "")
 for (i in 1:10) {
     fprintf("\%4d \%15.7f\n", i, exp(i), file = "")
 }
```

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

```r
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 $2\times3^N+2$ points.
The Koch snowflake curve is a closed curve in the plane with $3\times2^N+1$ 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 $6\times3^N+1$ points.
fractalcurve

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 MatlabCentral under BSD license; here re-implemented in R with explicit allowance from the author.

References


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)
ggrid()
## 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")
```r
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)
```

### Description

(Normalized) Fresnel integrals \( S(x) \) and \( C(x) \)

#### Usage

```r
fresnelS(x)
fresnelC(x)
```

#### Arguments

- `x` numeric vector.

#### Details

The **normalized** Fresnel integrals are defined as

\[
S(x) = \int_0^x \sin\left(\frac{\pi}{2} t^2\right) dt
\]

\[
C(x) = \int_0^x \cos\left(\frac{\pi}{2} t^2\right) dt
\]

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


See Also

gaussLegendre

Examples

```r
## 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)
f <- sum(cgl$w * f1(cgl$x))
f <- sum(cgl$w * f2(cgl$x))
  cat(formatC(c(x, fresnelS(x), f, fresnelC(x), fc),
             digits = 8, width = 12, flag = "-"), \n    digits = 8, width = 12, flag = "-"))
}

## Not run:
x <- seq(0, 7.5, by = 0.025)
y <- fresnelS(x)
y <- fresnelC(x)

## Function plot of the Fresnel integrals
plot(x, y, type = "l", col = "darkgreen",
     xlim = c(0, 8), ylim = c(0, 1),
     xlab = "", ylab = "", main = "Fresnel Integrals")
lines(x, y, col = "blue")
legend(0.25, 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(y, x, col = "red")
lines(-y, -x, col = "red")
grid()
## End(Not run)
```

fsolve

Solve System of Nonlinear Equations

**Description**

Solve a system of m nonlinear equations of n variables.
Usage

```r
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


See Also

- `broyden`
- `gaussNewton`

Examples

```r
## 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)
```
fzero

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


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 $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ and the newtons function is applied.

Value

Complex point with sufficiently small function value.

See Also

newtons
Examples

\begin{verbatim}
fz <- function(z) sin(z)^2 + sqrt(z) - log(z)
fzsolve(fz, 1+1i)
# 0.2555197+0.8948303i
\end{verbatim}

\section*{gammainc
\textit{Incomplete Gamma Function}}

\section*{Description}
Lower and upper incomplete gamma function.

\section*{Usage}
\begin{verbatim}
gammainc(x, a)
\end{verbatim}

\begin{verbatim}
incgam(x, a)
\end{verbatim}

\section*{Arguments}
\begin{itemize}
  \item \textbf{x} positive real number.
  \item \textbf{a} real number.
\end{itemize}

\section*{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 \texttt{pgamma}. The accuracy is thus much higher. It works for \(a \geq -1\), for even smaller values a recursion will give the result.

\section*{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.

\section*{Note}
Directly converting Fortran code is often easier than translating Matlab code generated with f2matlab.


**gammaz**

**References**


**See Also**

`gamma, pgamma`

**Examples**

```r
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.
gaussHermite

References


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 ] - \infty, \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 ]-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


See Also

gaussLegendre, gaussLaguerre

Examples

cc <- gaussLegendre(17)
# Integrate exp(-x^2) from -Inf to Inf
sum(cc$w)       #=> 1.7724538509189552 == sqrt(pi)
# Integrate x^2 exp(-x^2)
sum(cc$w * cc$x^2)     #=> 0.88622692545276 == sqrt(pi) /2
# Integrate cos(x) * exp(-x^2)
sum(cc$w * cos(cc$x))  #=> 1.380388447204314 == 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, 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 \([0, \infty]\), 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


See Also

gaussLegendre, gaussHermite

Examples

```r
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)) #=> 0.4999999999994987 , should be 0.5
```
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


See Also

gaussHermite, gaussLaguerre

Examples

```r
## Quadrature with Gauss-Legendre nodes and weights
f <- function(x) sin(x)*cos(10*exp(x))/3
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:
```
gaussNewton

Gauss-Newton Function Minimization

Description

Gauss-Newton method of minimizing a term \( f_1(x)^2 + \ldots + f_m(x)^2 \) or \( F'F \) where \( F = (f_1, \ldots, f_m) \) is a multivariate function of \( n \) variables, not necessarily \( n = m \).

Usage

```r
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 \( F \); 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**


**See Also**

`newtonsys`, `softline`

**Examples**

```r
gaussNewton(c(4, 4), f1)

f2 <- function(x) c( x[1] + 10*x[2], sqrt(5)x(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)
gaussNewton(c(0, 0), f3)
# $xs 0.5671433 0.5671433

f4 <- function(x)  # Dennis Schnabel
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)
    }
```
\[
\begin{align*}
D & \leftarrow X \odot X \odot X - \text{matrix}(c(1,3,2,4), 2, 2) \\
\text{return}(c(D))
\end{align*}
\]

sol \leftarrow \text{gaussNewton}(\text{ones}(2,2), F2) \\
(X \leftarrow \text{matrix}(\text{sol}$\times$s, 2, 2)) \\
# \begin{bmatrix}
[1,] & [2,] \\
\text{[1,]} & -0.1291489 & 0.8602157 \\
\text{[2,]} & 1.2903236 & 1.1611747
\end{bmatrix}
\]

\[X \odot X \odot X \odot X\]

---

gauss_kronrod

\textit{Gauss-Kronrod Quadrature}

\textbf{Description}

Simple Gaussian-Kronrod quadrature formula.

\textbf{Usage}

\[
\text{gauss_kronrod}(f, a, b, \ldots)
\]

\textbf{Arguments}

- \(f\) function to be integrated.
- \(a, b\) end points of the interval.
- \(\ldots\) variables to be passed to the function.

\textbf{Details}

Gaussian quadrature of degree 7 with Gauss-Kronrod of degree 15 for error estimation, the quadQK15 procedure in the QUADPACK library.

\textbf{Value}

List of value and relative error.

\textbf{Note}

The function needs to be vectorized (though this could easily be changed), but the function does not need to be defined at the end points.

\textbf{References}


\textbf{See Also}

quadgk, romberg
Examples

```
gauss_kronrod(sin, 0, pi)  # 2.00000000000000, rel.error: 1.14e-12
gauss_kronrod(exp, 0, 1)  # 1.718281828459045, rel.error: 0
# 1.718281828459045, i.e. exp(1) - 1
```

**gcd, lcm**  

### GCD and LCM Integer Functions

**Description**

Greatest common divisor and least common multiple

**Usage**

```r
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 = gcd(n, m) * 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

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 \( k = \text{round}(n \times \text{percent}/100/2) \) 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 \( \text{dim}=1 \) the mean of columns, \( \text{dim}=2 \) 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)
geo_mean(A, dim = 1)
harm_mean(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_i-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))

C generate QPP points on the unit sphere in the QPMdimN space
setNseedHQPPQI
p \ M randsHn\]QPPL n\]YI
H sol \M geo_medianHpI I
C Dp
C {Q} MPNPPYTXQSVQ MPNPPWVTSTQP MPNPPQRURYQP PNPPVTSWWPS MPNPQYYXRXXU MPNPTUSSWYXW
C {W} PNPSVRTYUVS PNPPSRSRQWU PNPSUPTPUYR PNPTVWQSPRS
C Dd
C {Q} YYNVVSX
C Dreltol
C {Q} SNPVYPVSeMPX
C Dniter
C {Q} QP

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 * [xk; x1] = [x, θ] where x=sqrt(xk^2+x1^2)

Value

List with two matrices Q and R, Q orthonormal and R upper triangular, such that A=Q*%*R.
References


See Also

householder

Examples

```r
## QR decomposition
A <- matrix(c(1,-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 $A\times x = b$ for $x$.

Usage

```r
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


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 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 sqrt(eps).
... 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

\[
\left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)
\]

numerically using the “central difference formula”.

**Value**

Vector of the same length as \(x_0\).

**References**


**See Also**

`fderiv`

**Examples**

```r
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

```r
x <- seq(0, 1, by=0.2)
y <- c(1, 2, 3)
(M <- meshgrid(x, y))
gradients(M$X^2 + M$Y^2)
gradients(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)
```
gramSchmidt

Description

Modified Gram-Schmidt Process

Usage

gramSchmidt(A, tol = .Machine$double.eps^0.5)

Arguments

A numeric matrix with nrow(A)>=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

See Also

householder, givens

Examples

```r
# 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 $2^e$, $12*2^e$, or $20*2^e$

**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)
```
Description

Finding roots of univariate functions using the Halley method.

Usage

\texttt{halley(fun, x0, maxiter = 500, tol = 1e-08, \ldots)}

Arguments

- \texttt{fun}: function whose root is to be found.
- \texttt{x0}: starting value for the iteration.
- \texttt{maxiter}: maximum number of iterations.
- \texttt{tol}: absolute tolerance; default $\text{eps}^{\ast}(1/2)$
- \texttt{\ldots}: 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 \texttt{root}, \texttt{f.root}, the function value at the found root, \texttt{iter}, the number of iterations done, and the estimated precision \texttt{estim.prec}

References

http://mathworld.wolfram.com/HalleysMethod.html

See Also

\texttt{newtonRaphson}

Examples

\texttt{halley(sin, 3.0)} \quad \# 3.14159265358979 in the 3 iterations
\texttt{halley(function(x) x*exp(x) - 1, 1.0)}
\quad \# 0.567143290409784 Gauss' omega constant

\# Legendre polynomial of degree 5
\texttt{lp5 <- c(63, 0, -70, 0, 15, 0)/8}
\texttt{f <- function(x) polyval(lp5, x)}
\texttt{halley(f, 1.0)} \quad \# 0.906179845938664
**hampel**

**Hampel Filter**

**Description**

Median absolute deviation (MAD) outlier in Time Series

**Usage**

```
hampel(x, k, t0 = 3)
```

**Arguments**

- `x`: numeric vector representing a time series
- `k`: window length \(2*k+1\) in indices
- `t0`: threshold, default is 3 (Pearson’s rule), see below.

**Details**

The ‘median absolute deviation’ computation is done in the \([-k \ldots k]\) 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**


**See Also**

`findpeaks`
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 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:

\[ dhd(P,Q) = \max_{p \in P} \left\{ \min_{q \in Q} \|p-q\| \right\} \]

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 \( \max(dhd(P,Q), dhd(Q,P)) \).

Value

A single scalar, the Hausdorff distance (dimension).

References


See Also

distmat

Examples

```r
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))
```
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

loc1, loc2 Locations on earth; for format see Details.
R 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

Hessenberg Matrix

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 \( A = P \ast H \ast t(P) \).
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

### hessian

**Hessian Matrix**

**Description**

Numerically compute the Hessian matrix.

**Usage**

\[
\text{hessian}(f, x0, h = \text{.Machine}\$\text{double}\_\text{eps}^{(1/4)}, \ldots)
\]

**Arguments**

- \(f\): univariate function of several variables.
- \(x0\): point in \(\mathbb{R}^n\).
- \(h\): step size.
- \(\ldots\): 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

See Also
laplacian

Examples
```r
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)
```

---

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**

```r
histc(x, edges)
```

**Arguments**

- `x` numeric vector or matrix.
- `edges` numeric vector of grid points, must be monotonically non-decreasing.

**Details**

\[
\text{let } n = \text{histc}(x, \text{edges}) \text{ counts the number of values in vector } x \text{ that fall between the elements in the edges vector (which must contain monotonically nondecreasing values). } n \text{ is a length}(\text{edges}) \text{ vector containing these counts.}
\]

If `x` is a matrix then `cnt` and `bin` are matrices too, and

```r
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, codehistss, findInterval
Examples

```r
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 1
# $bin
# [1] 0 1 2 3 4 5 6 7 8 9 0 0

## Not run:
# Compare
findInterval(x, e)
# [1] 0 0 1 1 2 2 3 4 5 5 6 6 7 7 8 8 9 9 9
findInterval(x, e, all.inside = TRUE)
# [1] 1 1 1 1 2 3 4 4 5 5 6 6 7 7 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,] 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,] 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

```r
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


See Also

`hist`, `histc`

Examples

```r
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)
```
Arguments

- `x0`: starting vector.
- `fn`: nonlinear function to be minimized.
- `...`: additional arguments to be passed to the function.
- `lb, ub`: lower and upper bounds.
- `tol`: relative tolerance, to be used as stopping rule.
- `maxfeval`: maximum number of allowed function evaluations.
- `target`: iteration stops when this value is reached.
- `info`: 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.

See Also

- `nelder_mead`
Examples

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

hooke_jeeves(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**

```r
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

The function `horner` utilizes the Horner scheme to evaluate the polynomial and its first derivative at the same time.

The polynomial \( p = p_{-1}x^n + p_{-2}x^{n-1} + \ldots + p_nx + p_{n+1} \) is hereby represented by the vector \( p_{-1}, p_{-2}, \ldots, p_n, p_{n+1} \), 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

\[ p(t) = q(t) \ast (t - x) + r, \ r \ \text{constant} \]

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


See Also

- `polyval`

Examples

```r
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
```
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 * R.

References

See Also
givens

Examples
```r
## 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]) / (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 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 psinc(x, n) = \(\frac{\sin(xn/2)}{n \sin(x/2)}\).

Value

Numeric scalar or vector.

Examples

```r
## Not run:
plot(humps(), type="l"); grid()
x <- seq(0, 10, length=101)
plot(x, sinc(x), type="l"); grid()
## End(Not run)
```
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
  • Hale - 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.

See Also

fractal::hurstSpec, RoverS, hurstBlock and fArma::LrdModelling

Examples

```r
## 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
```
**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. In all other cases, `x` and `y` have to be of the same size.

**Examples**
hypot(3, 4)
hypot(c(0, 0), c(3, 4))
ifft

Inverse Fast Fourier Transformation

Description
Performs the inverse Fast Fourier Transform.

Usage
ifft(x)

ifftshift(x)
fftshift(x)

Arguments
x a real or complex vector

Details
ifft returns the value of the normalized discrete, univariate, inverse Fast Fourier Transform of the values in x.
ifftshift 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
\begin{verbatim}
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)
\end{verbatim}

**inpolygon**                                           **Polygon Region**

**Description**

Points inside polygon region.

**Usage**

\[
inpolygon(x, y, xp, yp, boundary = FALSE)
\]

**Arguments**

- \texttt{x, y} \hspace{1cm} x-, y-coordinates of points to be tested for being inside the polygon region.
- \texttt{xp, yp} \hspace{1cm} coordinates of the vertices specifying the polygon.
- \texttt{boundary} \hspace{1cm} 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 \texttt{x}.
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, ...)

Note

Special care taken for points on the boundary.

References


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
ininpolygon(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)
**Arguments**

- **fun**: integrand, univariate (vectorized) function.
- **xmin, xmax**: endpoints of the integration interval.
- **method**: integration procedure, see below.
- **no_intervals**: number of subdivisions 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**

See Also

quadk, quadgr, quadcc, simpadpt, romberg, quadv, quadinf

Examples

## Very smooth function
fun <- function(x) 1/(x^4+x^2+0.9)
val <- 1.58223963729353
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 3.1974424e-13
  # Rich 1.582233 3.1974424e-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)
  if (x<=0 || x>=pi) 0 else sin(x)
Fun <- Vectorize(fun)
integral(Fun, -100, 100, no_intervals = 1)  # 0
integral(Fun, -100, 100, no_intervals = 1)  # 0
integral(Fun, -100, 100, random=FALSE)    # 2.00000000371071
integral(Fun, -100, 100, random=TRUE)     # 2.00000000340142
integral(Fun, -1000, 1000, random=FALSE)  # 2.00000000655435
integral(Fun, -1000, 1000, random=TRUE)   # 2.00000001157690 (sometimes 0 !)
**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.
reitol 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 ymin and ymax, 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 <= 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

\[ 0 <= a <= \theta <= b - a \text{ and } b \text{ must be constants} \]
\[ c <= r <= d - c \text{ and } d \text{ can be constants or ...} \]
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.

\texttt{integral2} can be applied to functions that are singular on a boundary. With value \texttt{singular=TRUE}, this option causes \texttt{integral2} to use transformations to weaken singularities for better performance.

\texttt{integral3} also accepts functions for the inner interval limits. \( y_{\text{min}}, y_{\text{max}} \) must be constants or functions of one variable \( (x) \), \( z_{\text{min}}, z_{\text{max}} \) 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 \texttt{integral2}, and then integrated over a single variable with \texttt{integral}.

\textbf{Value}

Returns a list with \texttt{int} the integral and \texttt{error} the error term.

\textbf{Note}

To avoid recursion, a possibly large matrix will be used and passed between subprograms. A more efficient implementation may be possible.

\textbf{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.

\textbf{References}


\textbf{See Also}

\texttt{integral, cubature:adaptIntegrate}

\textbf{Examples}

```r
fun <- function(x, y) cos(x) * cos(y)
integral2(fun, 0, 1, 0, 1, reltol = 1e-10)
# $int: 0.708073418273571 # $error: 8.618277e-19

# 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$int # 0.5236076 - pi/6 => 0.800354e-06

# Compute the volume over a sector
I <- integral2(f, 0, pi/2, 0, 1, sector = TRUE)
```


## interp1

**One-dimensional Interpolation**

### Description

One-dimensional interpolation of points.

### Usage

```r
interp1(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.

\( \text{xi} \)  
Numeric vector; points at which to compute the interpolation; all points must lie between \( \min(x) \) and \( \max(x) \).

\( \text{method} \)  
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 \( \text{xi} \).

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 \( \text{xi} \).

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 \text{pchip} here.

See Also

approx, spline

Examples

```r
x <- c(0.8, 0.3, 0.1, 0.6, 0.9, 0.5, 0.2, 0.8, 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(x, y, col="black", lty = 2)
lines(x1, y1, 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)
y <- 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**

```r
## 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**

```r
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:
for (i in 1:n) {
  for (j in 1:n) {
    b[i, j] <- (-1)^(i+j) * det(A[-j, -i]) / detA
  }
}
b

invlap  

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


Examples

```r
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, 10000)
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
Li <- invlap(Fs, 0, 0.01, 5, 200, 6, 40, 20)
plot(Li[[1]], Li[[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()) # FALSE
isempty(matrix(0, 1, 0)) # TRUE
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
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
}
}

}
Description

Iterative solutions of systems of linear equations.

Usage

\texttt{itersolve(A, b, x0 = NULL, nmax = 1000, tol = .Machine$double.eps*(0.5),
\hspace{3cm} \text{method = c("Gauss-Seidel", "Jacobi", "Richardson")})}

Arguments

- \texttt{A} numerical matrix, square and non-singular.
- \texttt{b} numerical vector or column vector.
- \texttt{x0} starting solution for iteration; defaults to null vector.
- \texttt{nmax} maximum number of iterations.
- \texttt{tol} relative tolerance.
- \texttt{method} iterative method, Gauss-Seidel, Jacobi, or Richardson.

Details

Iterative methods are based on splitting the matrix \texttt{A=(P-A)-A} with a so-called ‘preconditioner’ matrix \texttt{P}. The methods differ in how to choose this preconditioner.

Value

Returns a list with components \texttt{x} the solution, \texttt{iter} the number of iterations, and \texttt{method} the name of the method applied.

Note

Richardson’s method allows to specify a ‘preconditioner’; this has not been implemented yet.

References


See Also

\texttt{qrSolve}
Examples

```r
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 $\mathbb{R}^n \rightarrow \mathbb{R}^m$.

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 $x_1, \ldots, x_n$.

Note

Obviously, this function is not vectorized.

References

See Also

gradient

Examples

```r
## Example function from Quarteroni & Saleri
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

```r
kriging(u, v, u0, type = c("ordinary", "simple"))
```

Arguments

- `u` an `nxm`-matrix of `n` points in the `m`-dimensional space.
- `v` an `n`-dim. (column) vector of interpolation values.
- `u0` a `kxm`-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. vector of interpolation values.

Note

In the literature, different versions and extensions are discussed.

References

See Also

akimaInterp, barylag2d, package kriging

Examples

## Interpolate the Saddle Point function

```r
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  Kronecker product (Matlab Style)
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 a11*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 \(A \times x = b\) in an Lp sense, that is minimize the term \(\sum |b - A \times x|^p\). The case \(p=1\) is also called “least absolute deviation” (LAD) regression.

Usage

L1linreg(A, b, p = 1, tol = 1e-07, maxiter = 200)
Arguments

- \( \mathbf{A} \)  
  matrix of independent variables.

- \( \mathbf{b} \)  
  independent variables.

- \( p \)  
  the \( p \) in \( L^p \) norm, \( p \leq 1 \).

- \( \text{tol} \)  
  relative tolerance.

- \( \text{maxiter} \)  
  maximum number of iterations.

Details

L1/Lp 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 \( \mathbf{A} \).

Value

Returns a list with components 

\( \mathbf{x} \) the linear coefficients describing the solution, 

\( \text{reltol} \) the relative tolerance reached, and 

\( \text{niter} \) the number of iterations.

Note

In this case of \( p=1 \), the problem would be better approached by use of linear programming methods.

References


URL: http://mpra.ub.uni-muenchen.de/1781/1/MPRA_paper_1781.pdf

See Also

- \( \text{lm}, \text{lsqnonlin} \), \text{quantreg}::\text{rq} \)

Examples

```r
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
```

# \( \text{reltol} \)

```
[1] 6.712355e-10
```

# \( \text{niter} \)

```
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


See Also

roots
### Description

Principal real branch of the Lambert W function.

### Usage

```r
lambertWp(x)
lambertWn(x)
```

### Arguments

- `x`: Numeric vector of real numbers $\geq -1/e$.

### Details

The Lambert W function is the inverse of $x \rightarrow x \cdot e^x$, 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 are not really vectorized, but at least return a vector of values when presented with a numeric vector of length $\geq 2$.

### Value

Returns the solution $w$ of $w \cdot e^w = x$ 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

See Also

halley

Examples

```r
## 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(-1,6),
     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 Operator

---

laplacian
**Description**

Numerically compute the Laplacian of a function.

**Usage**

\[
\text{laplacian}(f, x0, h = \text{.Machine}\$\text{double}\$.\text{eps}^{(1/4)}, \ldots)
\]

**Arguments**

- \( f \): univariate function of several variables.
- \( x0 \): point in \( R^n \).
- \( h \): step size.
- \( \ldots \): variables to be passed to \( f \).

**Details**

Computes the Laplacian operator \( f_{x_1 x_1} + \ldots + f_{x_n x_n} \) 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**


**See Also**

hessian

**Examples**

\[
\begin{align*}
f & \leftarrow \text{function}(x) \ x[1]^2 + 2 \times x[1] \times x[2] + x[2]^2 \\
laplacian(f, c(1,1))
\end{align*}
\]
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


See Also

barylag

Examples

lebesgue(seq(0, 1, length.out = 6)) #=> 3.100425
Legendre Functions (Matlab Style)

Description

Calculate the values of (associated) Legendre functions.

Usage

\texttt{legendreHnL} \texttt{xI}

Arguments

\begin{itemize}
  \item \texttt{n} \hspace{1cm} \text{degree of the Legendre polynomial involved.}
  \item \texttt{x} \hspace{1cm} \text{real points to evaluate Legendre’s functions at.}
\end{itemize}

Details

\texttt{legendreHnLxI} computes the associated Legendre functions of degree \texttt{n} and order \texttt{m}, evaluated for each element of \texttt{x} where \texttt{x} must contain real values in \([-1, 1]\).

If \texttt{x} is a vector, then \texttt{L} = \texttt{legendreHnL(n, x)} is an \((n+1)\)-by-\(N\) matrix, where \(N = \text{length}(x)\). Each element \(L[m+1,i]\) corresponds to the associated Legendre function of degree \texttt{legendreHnL(n, x)} and order \texttt{m} evaluated at \(x[i]\).

Note that the first row of \texttt{L} is the Legendre polynomial evaluated at \(x\).

Value

Returns a matrix of size \((n+1)\)-by-\(N\) where \(N = \text{length}(x)\).

Note

Legendre functions are solutions to Legendre’s differential equation (it occurs when solving Laplace’s equation in spherical coordinates).

See Also

\texttt{chebPoly}

Examples

\begin{verbatim}
x <- c(0.0, 0.1, 0.2)
legendre(2, x)
# [1] -0.5 -0.4850000 -0.440000
# [2] 0.0 -0.2984962 -0.5878775
# [3] 3.0 2.9700000 2.8800000

## Not run:
\end{verbatim}
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 R^n.
B  Matrix whose columns are to be projected.
\(x_0\)  Point in R^n to be projected onto C \(x = b\).
C, b  Matrix and vector, defining an affine subspace as \(C x = b\)
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 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 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 \(\text{orth}(A)\). If you want to project points onto a subspace that is defined by \(A x = 0\), then generate an orthonormal basis of the nullspace of A with \(\text{null}(A)\).
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 $A \mathbf{x} = \mathbf{b}$ and calculates the distance of $\mathbf{x} \neq \mathbf{0}$ from this subspace. The calculation is based on the following formula:

$$p = (I - A'(AA')^{-1})x0 + A'(AA')^{-1}b$$

Technically, if $a$ is one solution of $C \mathbf{x} = \mathbf{b}$, then the projection onto $C$ can be derived from the projection onto $S = C \mathbf{x} = 0$ 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 $\mathbb{R}^n$.

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


See Also

nullspace, orth

Examples

```r
#-- Linear projection -----------------------------------------------

# Projection onto the line (1,1,1) in $\mathbb{R}^3$
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)
## $S$
## [,1] [,2] [,3]
## [1,] 0.3333333 2 0
```
## Example: Linear and Affine Projection

### Linear Translation
Let $s$ be the linear subspace and $a$ the parallel affine subspace of $x = b$. The solution of the linear system then is 

$$a = \text{qr.solve}(A, b)$$

$$A0 = \text{nullspace}(A)$$

$$xp = c(a + \text{linearproj}(A0, x0 - a))$$

### Affine Projection
The projection onto the (hyper-)surface $x+y+z = 1$ in $\mathbb{R}^3$ is:

$$A <- t(A); b <- 1$$

$$x0 <- c(1,2,3)$$

$$\text{affineproj}(x0, A, b)$$

### Projection with Positivity
The projection with positivity is:

$$s <- \text{affineproj}(x0, A, b, \text{unbound} = \text{FALSE})$$

### Extended Example

```r
set.seed(65537)

n = 1000; m = 100  # dimension, codimension
x0 <- rep(0, n)  # project \( \{ 0, \ldots, 0 \} \)
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 <- c(a + drop(A0 %*% dot(x0-a, A0)))  # projection

Norm(xp - x0)  # [1] 0.06597077
```

### Solution with quadprog

```r
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

```r
library(CVXR)
x = Variable(n)  # n decision variables

objective = Minimize(p_norm(x0 - x))  # min! || p0 - x ||
**Description**

Provides complex line integrals.

**Usage**

```r
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

```r
## 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 $A \times \leq b$
- `b` vector, right hand side of the inequalities.
- `Aeq` matrix representing the equality constraints $Aeq \times \leq beq$
- `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 $\min c^T x$ 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


See Also

linprog::solveLP, lpSolve::lp

Examples

```r
## 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, Ain, b = bin)

#-- employee scheduling
cc <- c(1, 1, 1, 1, 1)
A <- (-1)*matrix(c(1, 0, 0, 0, 1,
1, 1, 0, 0, 0,
0, 1, 0, 0, 0,
0, 0, 1, 1, 0,
0, 0, 0, 1, 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, -1, 0, 0,
0, 1, 0, 1, -1, 0,
0, 0, 1, 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, 0, 0, 0,
0, 0, 0, 0, 0,
0, 0, 0, 3.5, 0,
0, 0, 0, 0, 5.5,
-5, -4.5, -5.5, -5.5, -5.5,
10, 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)
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,
linspace

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)

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)
Description

Generate log-linearly spaced sequences.

Usage

\begin{verbatim}
logspace(x1, x2, n = 50)
logseq(x1, x2, n = 100)
\end{verbatim}

Arguments

- \textit{x1}: numeric scalar specifying starting point
- \textit{x2}: numeric scalar specifying ending point
- \textit{n}: numeric scalar specifying number of points to be generated

Details

These functions will generate logarithmically resp. exponentially spaced points between \textit{x1} and \textit{x2} resp. $10^{x1}$ and $10^{x2}$.

If \textit{n} < 2, the result will be the ending point \textit{x2}. For \texttt{logspace()}, if \textit{x2} = \texttt{pi}, the endpoint will be \texttt{pi} and not $10^{\texttt{pi}}$!

Value

vector containing \textit{n} points between \textit{x1} and \textit{x2} inclusive.

See Also

\texttt{logspace}, \texttt{seq}

Examples

\begin{verbatim}
logspace(1, pi, 36)
logseq(0.05, 1, 20)
\end{verbatim}
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 lsqin solves a more general problem, allowing additional linear inequalities and bound constraints. In pracma this task is solved applying function lsqincon.

Author(s)
HwB email: <hwborchers@googlemail.com>

References

See Also
nullspace, pinv, lsqincon
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)

C <- str2num(c('[1 1; 1 1]')

(L <- lsqin(A, b, C, d))
L <- str2num(c('[1; 1]')
(L <- lsqin(A, b[, 1], C, d))
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 <- lsqin(A2, b[, 2], C, d))
L <- as.matrix(c(1, 0.5))
```
Description
Solves linearly constrained linear least-squares problems.

Usage
\[ lsqlincon(C, d, A = \text{NULL}, b = \text{NULL}, \]
\[ \text{Aeq = NULL, beq = NULL, lb = NULL, ub = NULL}) \]

Arguments
- \( C \) : \( mxn \)-matrix defining the least-squares problem.
- \( d \) : vector or a one columna matrix with \( m \) rows.
- \( A \) : \( pxn \)-matrix for the linear inequality constraints.
- \( b \) : vector or \( px1 \)-matrix, right hand side for the constraints.
- \( \text{Aeq} \) : \( qxn \)-matrix for the linear equality constraints.
- \( \text{beq} \) : vector or \( qx1 \)-matrix, right hand side for the constraints.
- \( \text{lb} \) : lower bounds, a scalar will be extended to length \( n \).
- \( \text{ub} \) : upper bounds, a scalar will be extended to length \( n \).

Details
\[ lsqlincon(C, d, A, b, \text{Aeq}, \text{beq}, \text{lb}, \text{ub}) \] minimizes \( ||C \cdot x - d|| \) (i.e., in the least-squares sense) subject to the following constraints: \( A \cdot x \leq b, \text{Aeq} \cdot x = \text{beq}, \text{and lb} \leq x \leq \text{ub} \).

It applies the quadratic solver in \texttt{quadprog} with an active-set method for solving quadratic programming problems.

If some constraints are \texttt{NULL} (the default), they will not be taken into account. In case no constraints are given at all, it simply uses \texttt{qr.solve}.

Value
Returns the least-squares solution as a vector.

Note
Function \texttt{lsqlin} in \texttt{pracma} solves this for equality constraints only, by computing a base for the nullspace of \texttt{Aeq}. But for linear inequality constraints there is no simple linear algebra ‘trick’, thus a real optimization solver is needed.
Description

`lsqnonlin` solves nonlinear least-squares problems, including nonlinear data-fitting problems, through the Levenberg-Marquardt approach.

`lsqnonneg` solves nonnegative least-squares constraints problem.
Usage

\texttt{lsqnonlin(fun, x0, options = list(), ...)}
\texttt{lsqnonneg(C, d)}

\texttt{lsqsep(flist, p0, xdata, ydata, const = TRUE)}
\texttt{lsqcurvefit(fun, p0, xdata, ydata)}

Arguments

- \texttt{fun}: User-defined, vector-valued function.
- \texttt{x0}: starting point.
- \ldots: additional parameters passed to the function.
- \texttt{options}: list of options, for details see below.
- \texttt{C, d}: matrix and vector such that \( C x \approx d \) will be minimized with \( x \geq 0 \).
- \texttt{flist}: list of (nonlinear) functions, depending on one extra parameter.
- \texttt{p0}: starting parameters.
- \texttt{xdata, ydata}: data points to be fitted.
- \texttt{const}: logical; shall a constant term be included.

Details

\texttt{lsqnonlin} computes the sum-of-squares of the vector-valued function \texttt{fun}, that is if \( f(x) = (f_1(x), \ldots, f_n(x)) \) then

\[
\text{min} ||f(x)||^2_2 = \text{min}(f_1(x)^2 + \ldots + f_n(x)^2)
\]

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 \texttt{fun}. \texttt{fun} shall return a vector of values and not the sum of squares of the values. (The algorithm implicitly sums and squares \texttt{fun}(x).)

\texttt{options} is a list with the following components and defaults:

- \texttt{tau}: used as starting value for Marquardt parameter.
- \texttt{tolx}: stopping parameter for step length.
- \texttt{tolg}: stopping parameter for gradient.
- \texttt{maxeval}: the maximum number of function evaluations.

Typical values for \texttt{tau} are from \( 1e{-6} \ldots 1e{-3} \ldots 1 \) with small values for good starting points and larger values for not so good or known bad starting points.

\texttt{lsqnonneg} solves the linear least-squares problem \( C x \approx d, x \) nonnegative, treating it through an active-set approach.

\texttt{lsqsep} solves the separable least-squares fitting problem

\[
y = a0 + a1*f1(b1, x) + \ldots + aN*fn(bN, x)
\]

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. 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.
- errmess: error message, i.e. reason for stopping.

lsqnonneg returns a list of x the non-negative solution, and residNorm the norm of the residual.

lsqsep will return the coefficients sparately, a0 for the constant term (being 0 if const=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


See Also

nlm, nls

Examples

```r
## Rosenberg 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,
```
lsqnonlin

```r
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)
x0 <- c(21.16322, 0.83669, 2.957765)
lsqnonlin(f, x0)  # sq 50.50144 at c(36.133144, 2.572373, 1.079811)

# nls() will break down
# nls(Y - a*(exp((b-X)/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: Hougun function
x1 <- c(470, 285, 470, 470, 100, 100, 100, 470, 100, 100, 100, 100, 285, 285)
x2 <- c(300, 80, 300, 80, 80, 190, 80, 190, 300, 80, 300, 80, 190, 190)
x3 <- c(10, 10, 120, 120, 10, 10, 65, 65, 54, 120, 120, 10, 120)
rate <- c(5.55, 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] 1.25258502 0.06277577 0.04004772 0.11241472 1.19137819
# $sq 0.290901

## 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, 0.1009, 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, 0.9942, 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) = 0.9951*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, 1.12322349, 0.92688972,
      0.76793386, 0.63887755, 0.53378353, 0.44793636, 0.37758479, 0.31973932,
      0.27261388, 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, 0.4, 1.5, 0.5, 0.5, 0.5)
fp <- function(p, x) p[1]*exp(-p[2]*x) + p[3]*exp(-p[4]*x) + p[5]*exp(-p[6]*x)
```
lu

LU Matrix Factorization

Description

LU decomposition of a positive definite matrix as Gaussian factorization.

Usage

lu(A, scheme = c("kji", "jki", "ijk"))

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, ..., 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 oper-
ations are processed. The “ijk” scheme is one of the two compact forms, here the Doolite factoriza-
tion (the Crout factorization would be similar).

lufact applies partial pivoting (along the rows). lusys uses LU factorization to solve the linear
system A*x=b.

Value

lu returns a list with components L and U, the two 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

This function is not meant to process huge matrices or linear systems of equations. Without pivoting
it may also be harmed by considerable inaccuracies.

References

Verlag, Berlin Heidelberg.

(Prentice-Hall), updated 2006.

See Also

qr

Examples

A <- magic(5)
D <- lu(A, scheme = "ijk")  # Doolittle scheme
D$L %*% D$U
## [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)
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 \( \text{sum}(1 : n^2)/n \).

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**  
*M matlab Compatibility*

**Description**

Matlab compatibility.

**Usage**

`matlab()`

**Details**

Lists all the functions and function names that emulate Matlab functions.

**Value**

Invisible NULL value.

---

**meshgrid**  
*Generate a Mesh Grid*

**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)
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 lsqnonlin, 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,

\[ y = a_0 + a_1 e^{b_1 x} + \ldots + a_n e^{b_n x} \]

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.
**mexpfit**

**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**


**See Also**

*lsqsep, lsqnonlin*

**Examples**

```r
# Lanczos1 data (artificial data)
# f(x) = 0.0951*exp(-x) + 0.8607*exp(-5*x) + 1.5576*exp(-3*x) + 1.12323249
x <- linspace(0, 1.15, 24)
y <- c(2.51340000, 2.04433337, 1.66840444, 1.36641802, 1.12323249, 0.92688972, 0.76793386, 0.63887755, 0.53373535, 0.44793636, 0.37758479, 0.31973932, 0.27201308, 0.23249655, 0.19965895, 0.17227041, 0.14934057, 0.13007002, 0.11381193, 0.10004156, 0.08835209, 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
```
## 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 \( n \mod m \). 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 \( n \mod m \). mod(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 \(!=\) 0):

mod(n, m) = n - m * floor(n/m)
**Mode**

See Also

Binary R operators `%/%` and `%%`.

Examples

```r
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 function (Matlab style)

Description

Most frequent value in vector or matrix

Usage

```r
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
moler

Examples

x <- round(rnorm(1000), 2)
Mode(x)

moler matrix

Moler Matrix

Description

Generate the Moler matrix of size n x 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 x n

See Also

wilkinson

Examples

(a <- moler(10))
min(eig(a))
Description

Different types of moving average of a time series.

Usage

\[
\text{movavg}(x, n, \text{type}=c("s", "t", "w", "m", "e", "r"))
\]

Arguments

- \(x\) time series as numeric vector.
- \(n\) backward window length.
- \(\text{type}\) one of “s”, “t”, “w”, “m”, “e”, or “r”.

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 \(\text{cei}l(n+1)/2\); 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 \((1 - \text{reduction factor})\); 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
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
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
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
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).

A vector is seen as a matrix with one row, i.e. for a vector the number of dimensions is 2.

Value

the number of dimensions in the vector or array x.

Note

The result will differ from Matlab when x is a character vector.

See Also

size

Examples

ndims(1:8)
ndims(array(1:8, c(2,2,2)))
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

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 Function Minimization Method**

**Description**

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

**Usage**

```r
nelder_mead(fn, x0, ..., adapt = TRUE,
             tol = 1e-08, maxfeval = 5000,
             step = rep(1.0, length(x0)))
```

**Arguments**

- `fn`: nonlinear function to be minimized.
- `x0`: starting point for the iteration.
- `...`: additional arguments to be passed to the function.
- `adapt`: logical; adapt to parameter dimension.
- `tol`: terminating limit for the variance of function values; can be made *very* small, like `tol=1e-50`.
- `maxfeval`: maximum number of function evaluations.
- `step`: 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:

- \( \text{xmin} \): minimum solution found.
- \( \text{fmin} \): value of \( f \) at minimum.
- \( \text{fcount} \): number of iterations performed.
- \( \text{restarts} \): number of restarts.
- \( \text{errmess} \): error message

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


See Also

- hooke_jeeves

Examples

```r
# 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 +
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 +
x0 <- c(3, -1, 0, 1)
```
# needs maximum number of function calls
nelder_mead(psflx0, maxfeval=30000) # 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] 1.0000000 0.9999997 0.9999999 0.9999999 0.9999999 1.0000000
# [13] 0.9999999 0.9999994 0.9999998 0.9999999 0.9999999 0.9999999
# [19] 0.9999999 1.0000000 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**

```r
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

```r
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])
```

```r
## Not run:
ezplot(fp, 0, 4)
points(xx, yy)
## End(Not run)
```

newmark

<table>
<thead>
<tr>
<th>Newmark Method</th>
</tr>
</thead>
</table>

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

```r
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 \( y(t0) \), the second \( dy/dt(t0) \).
- **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 
\[
\text{zeta} = (\text{theta}+1/2)^*(1/2).
\]

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


See Also

ode23, cranknic

Examples

```r
# Mathematical pendulum  m 1 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

```r
newtonHorner(p, x0, maxiter = 50, tol = .Machine$double.eps^0.5)
```
newtonHorner

Arguments

- **p**: Numeric vector representing a polynomial.
- **x0**: Starting value for `newtonHorner()`.  
- **maxiter**: Maximum number of iterations; default 100.  
- **tol**: Absolute tolerance; default $\text{eps}^{(1/2)}$.

Details

Similar to `newtonRaphson`, 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


See Also

`newtonRaphson`

Examples

```r
## 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))
```
Description

Lagrange’s and Newton’s method of polynomial interpolation.

Usage

\[
\text{newtonInterp}(x, y, xs = \text{c}())
\]

\[
\text{lagrangeInterp}(x, y, xs)
\]

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

\texttt{neville, barylag}

Examples

\[
p <- \text{Poly(c(1, 2, 3))}
fp <- \text{function(x) polyval(p, x)}
\]

\[
x <- 0:4; y <- fp(x)
xx <- \text{linspace(0, 4, 51)}
yy <- \text{lagrangeInterp}(x, y, xx)
yy <- \text{newtonInterp}(x, y, xx)
## Not run:
\text{ezplot(fp, 0, 4)}
\text{points(xx, yy)}
## End(Not run)\]
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
fun Function or its name as a string.
x0 starting value for newtonRaphson().
dfun A function to compute the derivative of f. If NULL, a numeric derivative will be computed.
maxiter maximum number of iterations; default 100.
tol absolute tolerance; default eps^(1/2)
... Additional arguments to be passed to f.

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 root, 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

See Also
newtonHorner
Examples

```r
# 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
```

Description

Newton’s method applied to multivariate nonlinear functions.

Usage

```r
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

See Also

`newtonRaphson`, `broyden`
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

\[
\begin{align*}
\text{nextpow2}(10) & \quad \Rightarrow \ 4 \\
\text{nextpow2}(1:10) & \quad \Rightarrow \ 0 \ 1 \ 2 \ 2 \ 3 \ 3 \ 3 \ 4 \ 4 \\
\text{nextpow2}(-2^10) & \quad \Rightarrow \ 10 \\
\text{nextpow2}(\text{Machine}\$\text{double}.) & \quad \Rightarrow \ -52
\end{align*}
\]

---

nile

NILE overflow data

Description

Nile overflow data 1871–1984, gathered mostly by H. E. Hurst.

Usage

data("nile")

Format

A data frame with 114 years of observations during the months Jan to Dec.

Details

Monthly flow data taken at the Dongola measurement station just upstream from the high dam at Aswan.

References

Examples

data(nile)  # loads "nile" data frame

## Not run:
nile_dt <- nile[, 2:13]  # erase the "years" column

# plot all years in one figure
plot(ts(nile_dt), plot.type="single")

# merge all years in one time series
nile_ts <- ts(t(nile[, 2:13]), frequency = 12, start = c(1871, 1))

# aggregated flow per year
nile_flow <- apply(nile_dt, 1, sum)

plot(ts(nile_flow, frequency = 1, start = 1871) / 1000, 
     col = "darkblue", lwd = 2.0, 
     main = "Nile flows 1871 - 1984", ylab = "Flow / 1000")
grid()

# Hurst exponent of yearly Nile flow
hurstexp(nile_flow)

## Simple R/S Hurst estimation: 0.7348662
## Corrected R over S Hurst exponent: 1.041862
## Empirical Hurst exponent: 0.6975531
## Corrected empirical Hurst exponent: 0.7136607
## Theoretical Hurst exponent: 0.5244148

## End(Not run)

---

**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`. 
Norm

See Also

find

Examples

nnz(diag(10))

<table>
<thead>
<tr>
<th>Norm</th>
<th>Vector Norm</th>
</tr>
</thead>
</table>

Description

The `norm` function calculates several different types of vector norms for \( x \), depending on the argument \( p \).

Usage

\[ \text{Norm}(x, p = 2) \]

Arguments

\( x \) Numeric vector; matrices not allowed.
\( p \) Numeric scalar or Inf, -Inf; default is 2

Details

\( \text{Norm} \) returns a scalar that gives some measure of the magnitude of the elements of \( x \). It is called the \( p \)-norm for values \(-Inf \leq p \leq Inf\), defining Hilbert spaces on \( R^n \).

\( \text{Norm}(x) \) is the Euclidean length of a vector \( x \); same as \( \text{Norm}(x, 2) \).
\( \text{Norm}(x, p) \) for finite \( p \) is defined as \( \text{sum}(|\text{abs}(A)|^p)^{(1/p)} \).
\( \text{Norm}(x, \text{Inf}) \) returns \( \text{max}(|\text{abs}(x)|) \), while \( \text{Norm}(x, -\text{Inf}) \) 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 \( \text{norm} \); R’s \( \text{norm} \) function \( \text{norm}(x, "F") \) (‘Frobenius Norm’) is the same as \( \text{Norm}(x) \).

See Also

\( \text{norm} \) of a matrix
Examples

\begin{verbatim}
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)
\end{verbatim}

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

- \( M \) Numeric matrix; vectors will be considered as column vectors.
- \( maxiter \) Maximum number of iterations allowed; default: 100.
- \( tol \) 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

\[
\| M \|_2 = \max_{\| x \|_2 \neq 0} \frac{\| M x \|_2}{\| x \|_2}
\]

where \( \| . \|_2 \) 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):

\[
\max(\text{svd}(M)\,d)
\]
throot

References


See Also

cond, svd

Examples

\[
\text{normest(magic(5)) == max(svd(magic(5)))d} \quad \# \quad \text{TRUE}
\]
\[
\text{normest(magic(100))} \quad \# \quad 500050
\]

\begin{tabular}{ll}
throot & Real nth Root \\
\hline
\end{tabular}

Description

Compute the real n-th root of real numbers.

Usage

\text{nthroot}(x, n)

Arguments

\begin{itemize}
  \item \text{x} numeric vector or matrix
  \item \text{n} positive integer specifying the exponent \(1/n\).
\end{itemize}

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 \textit{Octave})

Value

Returns a numeric vector of solutions to \(x^{1/n}\).

See Also

sqrt

Examples

\[
\text{nthroot(c(1, -2, 3), 3)} \quad \#=> \quad 1.000000 \quad -1.259921 \quad 1.442250
\]
\[
(-2)^{(1/3)} \quad \#=> \quad \text{NaN}
\]
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 ... 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(nullspace(M)) + rank(M)

**Note**

The image of M can be retrieved from orth().

**References**


**See Also**

Rank, orth, MASS::Null
numderiv

Examples

\[ M <- \text{matrix}(1:12, 3, 4) \]
\[ \text{Rank}(M) \]
\[ N <- \text{nullspace}(M) \]
\[ \text{nullspace}(M) \]
\[ M1 <- \text{matrix}(1:6, 2, 3) \]
\[ M2 <- \text{t}(M1) \]
\[ \text{nullspace}(M1) \]
\[ \text{nullspace}(M2) \]
\[ M <- \text{magic}(5) \]
\[ \text{Rank}(M) \]
\[ \text{nullspace}(M) \]

numderiv

Richardson’s Numerical Derivative

Description

Richardson’s method applied to the computation of the numerical derivative.

Usage

\[ \text{numderiv}(f, x0, \text{maxiter} = 16, h = 1/2, \ldots, \text{tol} = \text{.Machine}$\text{double.eps}) \]
\[ \text{numdiff}(f, x, \text{maxiter} = 16, h = 1/2, \ldots, \text{tol} = \text{.Machine}$\text{double.eps}) \]

Arguments

\[ f \]
function to be differentiated.
\[ x0, x \]
point(s) at which the derivative is to be computed.
\[ \text{maxiter} \]
maximum number of iterations.
\[ h \]
starting step size, should be the default \( h=0.5 \).
\[ \text{tol} \]
relative tolerance.
\[ \ldots \]
variables to be passed to function \( f \).

Details

\text{numderiv} returns the derivative of \( f \) at \( x0 \), where \( x0 \) must be a single scalar in the domain of the function.
\text{numdiff} is a vectorized form of \text{numderiv} such that the derivatives will be returned at all points of the vector \( x \).
**numel**

**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**


**See Also**

fderiv, complexstep

**Examples**

```r
# 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 x.

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)
node23s(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)
node78(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 u0 needs to be a vector of length m.
ode23

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 \((t_{\text{final}} - t_0)/10\).

... Additional parameters to be passed to the function.

Details


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 \( x_{\text{out}} \) (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 \( t_0 \) and \( t_{\text{final}} \), 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


https://sites.google.com/site/comperem/home/ode_solvers

See Also

rk4sys, deval
Examples

```r
## 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[, 1], 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]/0, 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], l/(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)
```
odregress

## 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


See ODRPACK or ODRPACK95 (TOMS Algorithm 676).
URL: http://semi.vt.edu/presentations/SEMI-March05_Watson.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
orth

# $s = q$
# [1] 0.02168174

lm(y ~ x)
# Coefficients:
# (Intercept) x1 x2
# 0.9153 0.5671 1.6209

---

<table>
<thead>
<tr>
<th>orth</th>
<th>Range Space</th>
</tr>
</thead>
</table>

**Description**

Range space or image of a matrix.

**Usage**

orth(M)

**Arguments**

M

**Arguments**

Numeric matrix; vectors will be considered as column vectors.

**Details**

\( \text{B} = \text{orth}(A) \) 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**


**See Also**

nullspace

**Examples**

```r
M <- matrix(1:12, 3, 4)
Rank(M) #=> 2
orth(M)
```
**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**


**See Also**

taylor, ratInterp
Examples

```r
## 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)
```

### Description

Pascal triangle in matrix format

### Usage

```r
pascal(n, k = 0)
```

### Arguments

- `n` natural number
- `k` natural number, `k <= n`

### Details

Pascal triangle with `k` variations.

### Value

matrix representing the Pascal triangle

### See Also

`nchoosek`

### Examples

```r
pascal(5)
pascal(5, 1)
pascal(5, 2)
```
**pchip**

**Hermitean Interpolation Polynomials**

**Description**

Piecewise Cubic Hermitean Interpolation Polynomials.

**Usage**

```matlab
pchip(x, y, x)
pchipfun(x, y)
```

**Arguments**

- `x`, `y` x- and y-coordinates of supporting nodes.
- `x` 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.

`x` and `y` must be vectors of the same length greater or equal 3 (for cubic interpolation to be possible), and `x` must be sorted. `pchip` can be applied to points outside `[\min(x), \max(x)]`, but the result does not make much sense outside this interval.

**Value**

Values of interpolated data at points `x`.

**Author(s)**


**References**


**See Also**

`interp1`
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:

\[ z <- 3 \times (1-x)^2 \times \exp(-(x^2) - (y+1)^2) - 10 \times (x/5 - x^3 - y^5) \times \exp(-x^2 - y^2) - \]

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)
Examples

perms(6)
perms(1:6)
perms(c(1, exp(1), pi))

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**

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

**Arguments**

- `A`: 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:

\[ A B = A \]

The pseudoinverse `B` solves the problem to minimize \( |Ax - b| \) by setting \( x = Bb \)

```r
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$v %*% diag(1/s$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**


**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

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


Examples

```r
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

```r
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

```r
## Not run:
t <- deg2rad(seq(0, 360, by = 2))
polar(t, cos(2*pi*t), bxcol = "white", main = "Sine and Cosine")
polar(t, sin(2*pi*t), col = "red", add = TRUE)

## End(Not run)
```

---

**Poly**

*Define Polynomial by Roots*

---

**Description**

Define a polynomial by its roots.

**Usage**

```r
Poly(x)
```

**Arguments**

```r
  x         vector or square matrix
```

**Details**

Computes the characteristic polynomial of an (n x 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().
poly2str

See Also
	polyval, roots

Examples

Poly(c(1, -1, 11, -11))  # 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.coef: 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

See Also
chebApprox, polyfit
Examples

```r
## 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
```

```r
## 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

```r
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.
polyarea

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 parametrizised 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)
polyder

Derivative of Polynomial

Description

Differentiate polynomials.

Usage

polyder(p, q)

Arguments

p polynomial p given as a vector
q polynomial p 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

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 is 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

Anti-derivative of Polynomial

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

Polylogarithm Function

Description
Computes the n-based polylogarithm of z: \( \text{Li}_n(z) \).

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

\[
\sum_{k=1}^{\infty} \frac{z^k}{k^n} = z + \frac{z^2}{2^n} + ...
\]

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


Examples

- \text{polylog}(0.5, 1)  \quad \# \text{polylog}(z, 1) = -\log(1-z)
- \text{polylog}(0.5, 2)  \quad \# (\pi^2 - 6*\log(2)^2) / 12
- \text{polylog}(0.5, 3)  \quad \# (4*\log(2)^3 - 2*\pi^2*\log(2) + 21*zeta(3)) / 24
- \text{polylog}(0.5, 0)  \quad \# \text{polylog}(z, 0) = z/(1-z)
- \text{polylog}(0.5, -1) \quad \# \text{polylog}(z, -1) = z/(1-z)^2
Description

Multiply two polynomials given as vectors.

Usage

polymul(p, q)

Arguments

p, q  Vectors representing two polynomials.

Details

Polynomial multiplication realized simply by multiplying and summing up all the coefficients.

Value

Vector representing a polynomial.

Note

conv also realizes polynomial multiplication, through Fast Fourier Transformation, with the drawback that small imaginary parts may evolve.

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
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

Polynomial Transformation

Description

Transform a polynomial.

Usage

polytrans(p, q)

Arguments

p, q vectors representing two polynomials.

Details

Transforms polynomial p replacing occurrences of x with another polynomial q in x.

Value

Vector representing a polynomial.

Note

There is no such function in Matlab or Octave.

See Also

polypow

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
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**

```r
# 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 $f \times 2^e$, 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 $f \times 2^e$.

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

x, y  
-x-, y-coordinates of given points.

xi

x-coordinates of the choosen support nodes.

method

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

```r
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, 228.1925, 199.5875,
       197.3925, 175.3050, 218.6325, 163.0775, 178.6625, 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**

```markdown
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 \( L \times k \) 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

```r
## Example: Linear interpolation of the sine function
xs <- linspace(0, pi, 10)
y <- sin(xs)
P <- matrix(NA, nrow = 9, ncol = 2)
for (i in 1:9) {
P[i,1] <- (ys[i+1]-ys[i])/(xs[i+1]-xs[i])
     ypsin <- 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 to \( n \), resp. between \( n_1 \) and \( n_2 \).

**Usage**

```r
primes(n)
```

**Arguments**

- **n**: nonnegative integer greater than 1.

**Details**

The list of prime numbers up to \( n \) is generated using the "sieve of Eratosthenes". 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`
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*Q = B 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


See Also

`svd`

Examples

```r
## Procrustes
U <- randortho(5)       # random orthogonal matrix
P <- procrustes(U, eye(5))

## Kabsch
P <- matrix(c(0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1), nrow = 3, ncol = 8, byrow = TRUE)
R <- c(1, 1, 1)
```
\[ \psi(n, z) = \frac{d^{n+1}}{dz^{n+1}} \log(\Gamma(z)) \]

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

\[
\begin{align*}
\psi(2) - \psi(1) & \quad \# 1 \\
-\psi(1) & \quad \# Eulers constant: 0.57721566490153 \ [or, -\psi(0, 1)] \\
\psi(1, 2) & \quad \# \pi^2/6 - 1 : 0.64493406684823 \\
\psi(10, -11.5-0.5770078135681421i) & \quad \# is near a root of the decagamma function
\end{align*}
\]
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:
\[ \min q(x) = ||Gx||_2^2 = x'(G'G)x \]
\[ \text{s.t. } \sum x = 1 \]
\[ \text{and } x \geq 0 \]

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:
\[ \min q(x) = 0.5 \cdot t(x)x - d x \]
\[ \text{s.t. } A x \geq b \]

with A x = b for the first meq rows.

Value

Returns a list with the following components:

- x – optimal point attaining optimal value;
- d = Gx – smallest vector in the convex hull;
- q – optimal value found, = t(d) %%% d;
\[ qpspecial, qpsolve \]

- 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, 0.66, 0.44, 0.80), 3, 4, byrow = TRUE)
qpspecial(G)
# $x
# [1] 1.383697e-07
# $d
# [1] 4.940377
# $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
qrSolve

Description

Systems of linear equations via QR decomposition.

Usage

```
qrSolve(A, b)
```

Arguments

- `A`: numerical matrix with `nrow(A) >= 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


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

See Also

integrate, quadl

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.
Quadcc

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", \texttt{http://people.maths.ox.ac.uk/trefethen/ten_digit_algs.htm} and Matlab code at \texttt{http://people.maths.ox.ac.uk/trefethen/tda.html}.

References


See Also

\texttt{quad}, \texttt{cubature::adaptIntegrate}

Examples

```r
## 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
```

Adaptive Clenshaw-Curtis Quadrature

Description

Adaptive Clenshaw-Curtis Quadrature.

Usage

\texttt{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

```r
## Not run:  
## Dilogarithm function  
flog <- function(t) log(1-t)/t  
quadcc(flog, 1, 0, tol = 1e-12)  
# 1.64493406688128 - pi^2/6 < 1e-13

## End(Not run)
```

---

**quadgk**  
*Adaptive Gauss-Kronrod Quadrature*

**Description**

Adaptive Gauss-Kronrod Quadrature.

**Usage**

```r
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 $F = \text{Vectorize}(f)$.

**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**

```r
## 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**

```r
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
```r
## 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
# 2.07167616395854e-13
```

Infinite Integrals
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.
quadinf

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 \( g(t) = \tanh(\pi t/2 \sinh(t)) \).

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


See Also

`integrate`, `quadgk`

Examples

```r
## 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 = π
quadinf(f, -Inf, Inf)      # 3.141592653589784
```
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?
- \( \ldots \): 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

quadprog

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)
# 1.2821290743501 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)

---

**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:

\[ \min \frac{1}{2} x' C x + d' x \]

such that the following constraints are satisfied:

\[ A x \leq b \]
\[ Aeq x = beq \]
\[ lb \leq x \leq ub \]

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


See Also

- lsqlincon, quadprog::solve.QP

Examples

```markdown
## Example in ?solve.QP
# Assume we want to minimize: 1/2 x^T x - (0 5 0) \times x
# under the constraints: A x <= b
# with b = (8,-2, 0)
# and
# 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)
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] -8.2222222
# $eflag
# [1] 1

---

**quadv**

**Vectorized Integration**

---

**Description**

Vectorized adaptive Simpson integration.
Usage

```r
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

```r
## 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] 0.5000000 0.3333333 0.2500000 0.2000000 0.1666667 0.1428571 0.1250000 0.1111111 0.1000000 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**

\[
quiver(x, y, u, v, \\
\text{scale} = 0.05, \text{angle} = 10, \text{length} = 0.1, \ldots)
\]

**Arguments**

- \(x, y\) \(x,y\)-coordinates of start points of the arrows.
- \(u, v\) \(x,y\)-coordinates of start points.
- \(\text{scale}\) scales the length of the arrows.
- \(\text{angle}\) angle between shaft and edge of the arrows.
- \(\text{length}\) length of the arrow edges.
- \(\ldots\) 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`
Create Random Matrices

Description

Create random matrices or random points in a unit circle (Matlab style).

Usage

\begin{verbatim}
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)
\end{verbatim}

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

\(\text{rand()}\), \(\text{randn()}\), \(\text{randi()}\) create random matrices of size \(n \times m\), where the default is square matrices if \(m\) is missing.

\(\text{rand()}\) uses the uniform distribution on \([0, 1]\), while \(\text{randn()}\) uses the normal distribution with mean 0 and standard deviation 1.

\(\text{randi()}\) generates integers between \(\text{imax}[1]\) and \(\text{imax}[2]\) resp. 1 and \(\text{imax}\), if \(\text{imax}\) is a scalar.

\(\text{randsample()}\) samples \(k\) elements from \(1:n\), with or without replacement, or returns a weighted sample (with replacement), using the weight vector \(w\) for probabilities.

\(\text{rands()}\) 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 \(S^{\langle N\rangle}\) and multiply with \(\text{unif}(n)^{\langle 1/(N-1)\rangle}\).

\(\text{randp()}\) generates uniformly random points in the unit circle (or in a circle of radius \(r\)).
**Value**

Matrices of size \( nxm \) resp. a vector of length \( n \).

- \( \text{rand}() \) returns a pair of values representing a point in the circle, or a matrix of size \((n,2)\).
- \( \text{rands}() \) 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**


**See Also**

`set.seed`

**Examples**

```r
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 <- \text{rands}(1000, N = 1, r = 2)
U <- \text{randp}(1000, 2)
## Not run:
plot(U[, 1], U[, 2], pch = "+", asp = 1)
points(P, pch = ".")
## End(Not run)

```

```r
#-- 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

\texttt{randcomb(a, m)}

Arguments

\begin{itemize}
  \item \texttt{a} numeric vector of some length \( n \)
  \item \texttt{m} integer with \( 0 \leq m \leq n \)
\end{itemize}

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 \texttt{perms()} function.

See Also

\texttt{combs}, \texttt{randperm}

Examples

\texttt{randcomb(seq(2, 10, by=2), m = 3)}
**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**

```r
randortho(n, type = c("orthonormal", "unitary"))
```

**Arguments**

- \( n \) : positive integer.
- \( \text{type} \) : orthonormal (i.e., real) or unitary (i.e., complex) matrix.

**Details**

Generates orthonormal or unitary matrices \( Q \), that is \( t(Q) \) resp. \( \text{conj}(Q) \) 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. \( Q \ %*% \text{conj}(Q) \) is the unit matrix of size \( n \).

**Note**

`rortho` was deprecated and eventually removed in version 2.1.7.

**References**


**Examples**

```r
Q <- randortho(5)
zapsmall(Q %*% t(Q))
zapsmall(t(Q) %*% Q)
```
**randperm**

*Random Permutation*

**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 $\min(ncol(M), nrow(M))$.

**Note**

The corresponding function in Matlab is called `rank`, but that term has a different meaning in R.

**References**


**See Also**

`nullspace`

**Examples**

```r
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 corresponding rational representation and returns the numeric values.

Value

rat returns a character vector of string representations of continuous fractions in the format \([b_0; b_1, \ldots, b_{(n-1)}]\).
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)
**ratiinterp**  
*Rational Interpolation*

**Description**

Burlisch-Stoer rational interpolation.

**Usage**

\[
\text{ratiinterp}(x, y, xs = x)
\]

**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**


**See Also**

`rationalfit`, `pade`
Examples

```r
## 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**

```r
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 (`d2`) 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 $d1$ and $d2$) such that $p_1(x)/p_2(x)$ approximates $y$ in a least squares sense.

$d1$ and $d2$ must be large enough to get a good fit and usually $d1=d2$ gives good results.

**Value**

List with components $p1$ and $p2$ 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. Re-implementation by Hans W Borchers.

References


See Also

ratinterp

Examples

```r
## 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")
g```
Description

Calculates the area of intersection of rectangles, specified by position vectors \( x \) and \( y \).

Usage

\[ \text{rectint}(x, y) \]

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 \( \text{rectint}(x[i,], y[j,]) \).

See Also

\( \text{polyarea} \)

Examples

\( x <- c(0.5, 0.5, 0.25, 1.00) \)
\( y <- c(0.3, 0.3, 0.35, 0.75) \)
\( \text{rectint}(x, y) \)
\( \# [1] 0.0825 \)
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("abababa", 'aba')
gregexpr('a(?=ba)', "abababa", perl=TRUE)

refindall("AbababaBa", 'aba')
refindall("AbababaBa", 'aba', ignorecase = TRUE)
**regexp**

*Match regular expression*

**Description**

Returns the positions of substrings that match the regular expression.

**Usage**

\[
\text{regexp}(s, \text{pat}, \text{ignorecase} = \text{FALSE}, \text{once} = \text{FALSE}, \text{split} = \text{FALSE})
\]

\[
\text{regexpi}(s, \text{pat}, \text{once} = \text{FALSE}, \text{split} = \text{FALSE})
\]

**Arguments**

- **s**: Character string, i.e. of length 1.
- **pat**: Matching pattern as character string.
- **ignorecase**: Logical: whether case should be ignored; default: FALSE.
- **once**: Logical: whether the first are all occurrences should be found; default: all.
- **split**: 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 **regexp**.

**See Also**

regexp
Examples

```r
s <- "bat cat can car COAT court cut ct CAT-scan"
pat <- "[aeiou]+t"
regexp(s, pat)
regexpi(s, pat)
```

Description

Replace string using regular expression.

Usage

```r
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

\begin{verbatim}
s <- "bat cat can car COAT court cut ct CAT-scan"
p = 'c[aeiou]+t'
regexprep(s, pat, '----')
regexprep(s, pat, '----', once = TRUE)
regexprep(s, pat, '----', ignorecase = TRUE)
\end{verbatim}

Description

Replicate and tile matrix.

Usage

\begin{verbatim}
repmat(a, n, m = n)
\end{verbatim}

Arguments

- **a**: vector or matrix to be replicated.
- **n, m**: number of times to replicate in each dimension.

Details

\texttt{repmat(a,m,n)} creates a large matrix consisting of an m-by-n tiling of copies of \texttt{a}.

Value

Returns matrix with value \texttt{a} replicated to the number of times in each dimension specified. Defaults to square if dimension argument resolves to a single value.

See Also

\texttt{reshape}

Examples

\begin{verbatim}
repmat(1, 3)         # same as ones(3)
repmat(1, 3, 3)
repmat(matrix(1:4, 2), 3)
\end{verbatim}
Reshape

**Reshape Matrix**

**Description**

Reshape matrix or vector.

**Usage**

\[ \text{Reshape}(a, n, m) \]

**Arguments**

- **a**: matrix or vector
- **n, m**: size of the result

**Details**

\[ \text{Reshape}(a, n, m) \]

returns the \( n \)-by-\( m \) matrix whose elements are taken column-wise from \( a \).

An error results if \( a \) does not have \( n \times m \) 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**

\[
a \leftarrow \text{matrix}(1:12, \text{nrow}=4, \text{ncol}=3) \\
\text{Reshape}(a, 6, 2) \\
\text{Reshape}(a, 6) \quad \# \text{the same} \\
\text{Reshape}(a, 3, 4)
\]

---

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**

\[ \text{ridders}(\text{fun}, a, b, \text{maxiter} = 500, \text{tol} = 1e-12, \ldots) \]
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 the 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 bracket.

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


See Also

brent
Examples

```r
## 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) [1]*x^11 - 1 # [0.4, 1.6], 0.884 133 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], 0.679 008 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, 3.4); r$root; r$niter # 30, err = 1e-5

## Not run:
## Use ridders() with Rmpfr
options(digits=16)
library("Rmpfr")  # unrootR
prec <- 256
f12 <- function(x) {
  e1 <- exp(.N(1))
  log(x) + x^2/(2*e1) - 2*x/sqrt(e1) + 1
}
sqrtex <- sqrt(exp(.N(1))) # 1.648721270700128...
f12(sqrtex) # @
unrootR(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)
```
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 \( \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m \), 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


See Also

ode23, deval
Examples

```r
## 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]
  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)
```

Description

Runge-Kutta-Fehlberg with adaptive step size.

Usage

```r
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


Mathematica code associated with the book:

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


Examples

```r
x <- rep(1, 10)
y <- rnorm(10, 1, 0.1)
rmse(x, y, summary = TRUE)
```

---

### Description

Romberg Integration

### Usage

```r
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 or 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

See Also
integrate, quadgr

Examples
\begin{verbatim}
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: -0.66666667, iter: 7, rel.error: 1e-12
\end{verbatim}

\begin{table}
\centering
\begin{tabular}{l|l}
\hline
roots & \textit{Polynomial Roots} \\
\hline
\end{tabular}
\end{table}

Description
Computes the roots of the polynomial \(p\).

Usage
\[ \text{roots}(p) \]

Arguments
\begin{itemize}
\item \texttt{p} vector of real numbers representing the polynomial
\end{itemize}

Details
For solution the function computes the eigenvalues of the companion matrix.
At the moment, only real coefficients are allowed.

Value
a vector holding the roots of the polynomial


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 eigen-
value.

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 \text{rank}, \text{orth} or \text{null}.

Value
A matrix the same size as \( m \).

Note
This serves demonstration purposes only; don’t use for large matrices.

References
http://mathworld.wolfram.com/EchelonForm.html

See Also
\text{qr.solve}

Examples
\begin{verbatim}
A <- matrix(c(1, 2, 3, 1, 3, 2, 3, 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, 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
\end{verbatim}
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, f1, forder = 4, dorder = 0)

Arguments

T  Vector of signals to be filtered.
f1  Filter length (for instance f1 = 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)


References

See Numerical Recipes, 1992, Chapter 14.8, for details.

See Also

RTisean::sav_gol, signal::sgolayfilt, whittaker.
segm_distance

Examples

```r
# *** 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

```r
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 `l` 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

```r
## 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


See Also

segm_distance

Examples

```r
## 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**

```r
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
```r
## 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
```r
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 \texttt{ode45} with some other solver.

References


See Also

\texttt{bvp}

Examples

```r
#-- 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=range(sol$y))
xs <- seq(0, 1, length.out=100);
ys <- linspace(0, 1);
lines(xs, ys, col="red")
lines(sol$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-Piyavskii Method**

### Description
Shubert-Piyavskii Univariate Function Maximization

### Usage
```matlab
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:
- `xopt`: the x-coordinate of the minimum found.
- `fopt`: the function value at the minimum.
- `nopt`: number of steps.

### References

### 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
## $fopt
## [1] 3.216209
## $nopt
## [1] 481

---

**Si, Ci**

**Sine and Cosine Integral Functions**

**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 γ 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*1, 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**

See Also

sinc, expint

Examples

```r
x <- c(-3:3) * pi
Si(x); Ci(x)

## Not run:
x <- linspace(0, 10*pi, 200)
y <- Si(x); yci <- Ci(x)
plot(c(0, 35), c(-1.5, 2.0), type = 'n', xlab = '', ylab = '',
     main = "Sine and cosine integral functions")
lines(x, y, col = "darkred", lwd = 2)
lines(x, yci, col = "darkblue", lwd = 2)
lines(c(0, 10*pi), c(pi/2, pi/2), col = "gray")
lines(x, cos(x), col = "gray")
grid()
## End(Not run)
```

sigmoid  Sigmoid Function

Description

Sigmoid function (aka sigmoidal curve or logistic function).

Usage

```r
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

\[
y = \frac{1}{1 + e^{-a(x-b)}}
\]

The sigmoid function is also the solution of the ordinary differential equation

\[
y' = y(1 - y)
\]

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

\[
y = b + \frac{1}{a}\log(x/(1 - x))
\]
**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


See Also

quad, simpson2d

Examples

```r
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

```r
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

```r
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

\[
\begin{align*}
\text{sind}(x) \\
\text{cosd}(x) \\
\text{tand}(x) \\
\text{cotd}(x) \\
\text{asind}(x) \\
\text{acosd}(x) \\
\text{atand}(x) \\
\text{acotd}(x) \\
\text{secd}(x) \\
\text{cscd}(x) \\
\text{asecd}(x) \\
\text{acscd}(x) \\
\text{atan2d}(x_1, x_2)
\end{align*}
\]

Arguments

\[x, \ x_1, \ x_2 \quad \text{numeric or complex scalars or vectors}\]

Details

The usual trigonometric functions with input values as scalar or vector in degrees. Note that \(\text{tand}(x)\) with fractional part does not return NaN as \(\text{tanpi}(x)\), but is computed as \(\text{sind}(x)/\text{cosd}(x)\).

For \(\text{atan2d}\) the inputs \(x_1, x_2\) 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

\[
\begin{align*}
\text{sind}(x) \text{ and } \text{cosd}(x) \text{ are accurate for } x \text{ which are multiples} \\
\text{# of 90 and 180 degrees, while tand}(x) \text{ is problematic.}
\end{align*}
\]

\[x \leftarrow \text{seq}(0, 720, \text{by} = 90)\]

\[
\begin{align*}
\text{sind}(x) & \quad \# 0 \ 1 \ 0 \ -1 \ 0 \ 1 \ 0 \ -1 \ 0 \\
\text{cosd}(x) & \quad \# 1 \ 0 \ -1 \ 0 \ 1 \ 0 \ -1 \ 0 \ 1 \\
\text{tand}(x) & \quad \# 0 \ \text{Inf} \ 0 \ -\text{Inf} \ 0 \ \text{Inf} \ 0 \ -\text{Inf} \ 0 \\
\text{cotd}(x) & \quad \# \text{Inf} \ 0 \ -\text{Inf} \ 0 \ \text{Inf} \ 0 \ -\text{Inf} \ 0 \ \text{Inf}
\end{align*}
\]
**size**

Size of Matrix

**Description**

Provides the dimensions of \( x \).

**Usage**

\[
\text{size}(x, k)
\]

**Arguments**

- \( x \) vector, matrix, or array
- \( k \) integer specifying a particular dimension

**Details**

Returns the number of dimensions as \( \text{length}(x) \).

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**

\[
\begin{align*}
\text{size}(1:8) & \quad \# 1 \ 8 \\
\text{size}(\text{matrix}(1:8, 2, 4)) & \quad \# 2 \ 4 \\
\text{size}(\text{matrix}(1:8, 2, 4), 2) & \quad \# 4 \\
\text{size}(\text{matrix}(1:8, 2, 4), 3) & \quad \# 1 \\
\end{align*}
\]
Fletcher’s inexact line search algorithm.

### Usage

```r
softline(x0, d0, f, g = NULL)
```

#### Arguments

- `x0`: initial point for linesearch.
- `d0`: search direction from `x0`.
- `f`: real function of several variables that is to be minimized.
- `g`: gradient of objective function `f`; 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


#### See Also

`gaussNewton`
Examples

```r
g_himm <- function(x) {
  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 Routines

Description

R implementations of several sorting routines. These implementations are meant for demonstration and lecturing purposes.

Usage

```r
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 between3 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


See Also

sort, the internal C-based sorting routine.
Examples

```r
## 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)
```

**Description**

Sort rows of a matrix according to values in a column.

**Usage**

```r
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**

```r
A <- magic(5)
sortrows(A)
sortrows(A, k = 2)
```
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

\texttt{spinterp(x, y, xp)}

Arguments

- \texttt{x, y} \hspace{1cm} x- and y-coordinates of the points that shall be interpolated.
- \texttt{xp} \hspace{1cm} 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

spinterp

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, 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, 0.3, 0.96, -0.88, -0.62, 0.13, 1),
             y = c(-0.9, 0.3, 0.4, 0.5, 0.6, -0.88, -0.62, 0.13, 1))
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, (1/3)^3, 1))

## Not run:
opar <- 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)
Matrix Square and p-th Roots

Description

Computes the matrix square root and matrix p-th root of a nonsingular real matrix.

Usage

\[
\text{sqrtm}(A, \text{kmax} = 20, \text{tol} = \text{.Machine}\$\text{double}\$.\text{eps}^*(1/2))
\]

\[
\text{signm}(A, \text{kmax} = 20, \text{tol} = \text{.Machine}\$\text{double}\$.\text{eps}^*(1/2))
\]

\[
\text{rootm}(A, p, \text{kmax} = 20, \text{tol} = \text{.Machine}\$\text{double}\$.\text{eps}^*(1/2))
\]

Arguments

- \(A\) numeric, i.e. real, matrix.
- \(p\) p-th root to be taken.
- \(\text{kmax}\) maximum number of iterations.
- \(\text{tol}\) 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 $\text{sectm}(A, m) = (A^m)^{(-1/p)}$; for $p=2$ this is the matrix sign function.

$S = \text{signm}(A)$ is real if $A$ is and has the following properties:

$S^2 = 1$; $S A = A S$

$\text{signm}([0 \ A; B \ 0]) = [0 \ C; C^{-1} \ 0]$ where $C = A(BA)^{-1}$

These functions arise in control theory.

Value

$sqrtm(A)$ returns a list with components

- $B$ square root matrix of $A$.
- $\text{Binv}$ inverse of the square root matrix.
- $k$ number of iterations.
- $\text{acc}$ accuracy or absolute error.

$\text{rootm}(A)$ returns a list with components

- $B$ square root matrix of $A$.
- $k$ number of iterations.
- $\text{acc}$ accuracy or absolute error.

If $k$ is negative the iteration has not converged.

$\text{signm}$ 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 $\text{expm}$, $\text{logm}$ 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


See Also

`expm`, `expm::sqrtm`

Examples

```r
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, 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

\[
\begin{align*}
\text{\textbf{R}} & : \ x \leftarrow 1:6 \\
& \ y \leftarrow \text{squareform}(x) \\
& \ \begin{bmatrix}
0 & 1 & 2 & 3 \\
1 & 0 & 4 & 5 \\
2 & 4 & 0 & 6 \\
3 & 5 & 6 & 0
\end{bmatrix} \\
& \text{all(squareform(y) == x)} \\
& \ \begin{bmatrix}
\text{TRUE} & \text{TRUE} & \text{TRUE} & \text{TRUE} \\
\text{TRUE} & \text{TRUE} & \text{TRUE} & \text{TRUE} \\
\text{TRUE} & \text{TRUE} & \text{TRUE} & \text{TRUE} \\
\text{TRUE} & \text{TRUE} & \text{TRUE} & \text{TRUE}
\end{bmatrix}
\end{align*}
\]

---

**std**

*Standard Deviation (Matlab Style)*

Description

Standard deviation of the values of \( x \).

Usage

\[
\text{std}(x, \text{flag}=0)
\]

Arguments

- \( x \) : numeric vector or matrix
- \( \text{flag} \) : numeric scalar. If 0, selects unbiased algorithm; and if 1, selects the biased version.

Details

If \( \text{flag} = 0 \) the result is the square root of an unbiased estimator of the variance. \( \text{std}(X, 1) \) 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

\( \text{flag} = 0 \) produces the same result as R’s sd().

Examples

\[
\begin{align*}
\text{std}(1:10) & \quad \# 3.027650 \\
\text{std}(1:10, \text{flag}=1) & \quad \# 2.872281
\end{align*}
\]
Steep Descent Minimization

Description

Function minimization by steepest descent.

Usage

\[
\text{steep\_descent}(x_0, f, g = \text{NULL}, \text{info} = \text{FALSE}, \\
\quad \text{maxiter} = 100, \text{tol} = .\text{Machine}\_\text{double}\_\text{eps}\_\text{eps}(1/2))
\]

Arguments

\[x_0\] start value.
\[f\] function to be minimized.
\[g\] gradient function of \(f\); if \(\text{NULL}\), a numerical gradient will be calculated.
\[\text{info}\] logical; shall information be printed on every iteration?
\[\text{maxiter}\] max. number of iterations.
\[\text{tol}\] 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:

\[\text{xmin}\] minimum solution found.
\[f_{\text{min}}\] value of \(f\) at minimum.
\[\text{niter}\] number of iterations performed.

Note


References


See Also

fletcher_powell
Examples

```r
# Rosenbrock function: The flat valley of the Rosenbruck 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.
```

```r
# 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
```

---

**str2num**

Converting string to number (Matlab style)

---

**Description**

Functions for converting strings to numbers and numbers to strings.

**Usage**

```r
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

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'.
strcmp

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` they 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**  

*Find Substrings*

**Description**

Find substrings within strings of a character vector.

**Usage**

\[
\text{strfind(s1, s2, overlap = TRUE)} \\
\text{strfindi(s1, s2, overlap = TRUE)} \\
\text{findstr(s1, s2, overlap = TRUE)}
\]

**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

```r
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

```r
S <- c("abc", "letters", "1", "2 2")
strjust(S, "left")
```
Find and replace substring

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

Remove leading and trailing white space.

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.

debblank 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)  
debblank(s)
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 or 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


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..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**


**See Also**

`aitken`
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 <= 8`.
- `...`: more variables to be passed to function `f`. 

Examples

```r
# 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) # 0.82246703394433 = pi^2/12 + eps()

## Not run:
## Dirichlet eta() function: eta(s) = 1/1^s - 1/2^s + 1/3^s - 1/4^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)
```
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) #=> 0.04166667 0.16666667 0.50000000 1.0000000 1.0000000

f <- function(x) log(1+x)
p <- taylor(f, 0, 4)

# log(1+x) = 0 + x - 1/2 x^2 + 1/3 x^3 - 1/4 x^4 + ...
# [1] -0.250000 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

```r
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

Examples

```r
## 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**

```r
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

**Matrix trace**

**Description**
Sum of the main diagonal elements.

**Usage**
```
Trace(a)
```

**Arguments**
- `a`: a square matrix

**Details**
Sums the elements of the main diagonal of a real or complex 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

**Trapezoidal Integration**

**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

- **x**: x-coordinates of points on the x-axis
- **y**: y-coordinates of function values
- **f**: function to be integrated.
- **a, b**: lower and upper border of the integration domain.
- **maxit**: maximum number of steps.
- **tol**: tolerance; stops when improvements are smaller.
- **...**: 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 \texttt{polyarea}. This approach matches exactly the approximation for integrating the function using the trapezoidal rule with basepoints \(x\).

\texttt{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 \(\text{length}(x)\).

Inputs \(x\) and \(y\) can be complex.

\texttt{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 \(\text{min}(x)\) to \(\text{max}(x)\). Or a matrix of the same size as \(y\).

\texttt{trapzfun} returns a list with components \(\text{value}\) the value of the integral, \(\text{iter}\) the number of iterations, and \(\text{rel.err}\) the relative error.

See Also

\texttt{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)  #=> 1.999835584

# 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)
c <- cumtrapz(z, 1/z)
c[101] #=> 0+3.14107591i
f <- function(x) x^(3/2)  #
trapzfun(f, 0, 1) #=> 0.4 with 11 iterations

---

### Triangular Matrices (Matlab Style)

**Description**

Extract lower or upper triangular part of a matrix.

**Usage**

```r
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

```r
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**

```r
trigApprox(t, x, m)
```

**Arguments**

- `t` : vector of points at which to compute the values of the trigonometric approximation.
- `x` : data from t=0 to t=2*(n-1)*pi/n.
- `m` : 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**

```r
## 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(1, 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**

```r
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,

\[ a_0 + \sum_k \left( a_k \cos(kt) + b_k \sin(kt) \right) \]

by applying orthogonality relations.

**Value**

Coefficients as a list with components a0, a, and b.
Note

For irregular spaced data or data not covering the whole period, use standard regression techniques, see examples.

References


See Also

trigapprox

Examples

```r
# 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**

```r
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.
- \( \text{tol} \)  
  relative tolerance to be achieved.
- \( \ldots \)  
  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


See Also

quad2d, simpson2d

Examples

```r
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)
```
trisolve

Tridiagonal Linear System Solver

Description

Solves tridiagonal linear systems $A \times x = \text{rhs}$ 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 $A \times x = \text{rhs}$.

Details

Solves tridiagonal linear systems $A \times x = \text{rhs}$ by applying Givens transformations.

By only storing the three diagonals, trisolve has memory requirements of $3 \times n$ 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


See Also

qrSolve
vander

** Examples**

```r
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**

```r
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**

```r
vander(c(1:10))
```
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)
```
Description

Smoothing of time series using the Whittaker-Henderson approach.

Usage

whittaker(y, lambda = 1600, d = 2)

Arguments

y 
  signal to be smoothed.

lambda 
  smoothing parameter (rough 50..1e4 smooth); the default value of 1600 has been recommended in the literature.

d 
  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


wilkinson

wilkinson Matrix

Description

Generate the Wilkinson matrix of size n x 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 x 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)

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 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


See Also

gammaz, eta
Examples

```r
## 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

*Topic arith
  agmean, 15
  angle, 21
  bits, 37
  ceil, 49
  combs, 58
  eps, 89
  gcd, lcm, 140
  mod, rem, 221
  nchoosek, 227
  nextpow2, 239
  nthroot, 244
  perms, 258
  pow2, 279
  primes, 282
  randcomb, 304
  randperm, 306
  kron, 189
  lu, 214
  magic, 216
  meshgrid, 217
  ndims, 228
  nearest_spd, 229
  nnz, 241
  Norm, 242
  normest, 243
  nullspace, 245
  numel, 247
  orth, 253
  pinv, 260
  poly2str, 265
  procrustes, 283
  qrSolve, 288
  Rank, 307
  repmat, 316
  Reshape, 317
  size, 343
  sorting, 345
  sortrows, 347
  squareform, 352
  subspace, 363
  Toeplitz, 368
  Trace, 369
  tri, 371
  trisolve, 376

*Topic array
  accumarray, 13
  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, 165
  hypot, 169
  ifft, 170
  inv, 180
  isposdef, 184
  itersolve, 186
  kron, 189
  lu, 214
  magic, 216
  meshgrid, 217
  ndims, 228
  nearest_spd, 229
  nnz, 241
  Norm, 242
  normest, 243
  nullspace, 245
  numel, 247
  orth, 253
  pinv, 260
  poly2str, 265
  procrustes, 283
  qrSolve, 288
  Rank, 307
  repmat, 316
  Reshape, 317
  size, 343
  sorting, 345
  sortrows, 347
  squareform, 352
  subspace, 363
  Toeplitz, 368
  Trace, 369
  tri, 371
  trisolve, 376

*Topic datasets
  brown72, 40
  nile, 240
  titanium, 367

*Topic fitting
  akimaInterp, 18
  circlefit, 54
  cubicspline, 70
  curvefit, 71
  fornberg, 122
  kriging, 188
INDEX

mldivide, 220
muller, 225
neville, 232
newtonHorner, 234
newtonInterp, 236
newtonRaphson, 237
newtonsys, 238
numderiv, 246
pade, 254
pchip, 256
piecewise, 259
Poly, 264
polyadd, 266
polyApprox, 267
polyarea, 268
polyder, 270
polyfit,polyfix, 271
polyint, 273
polylog, 273
polymul, 275
polypow, 276
polytrans, 277
polyval, polyvalm, 278
quad, 289
quad2d, 290
quadcc, 291
quadgk, 292
quadgr, 293
quadinf, 294
quadl, 296
quadv, 299
randortho, 305
rat, 308
rectint, 312
ridders, 317
romberg, 324
roots, 325
rref, 327
runge, 329
simpadpt, 339
simpson2d, 340
sind,cosd,tand, etc., 341
spinterp, 348
sqrtm,rootm, 350
sumalt, 364
taylor, 365
trapz, 369
trigApprox, 372

trigPoly, 373
triquad, 374

*Topic ode
abm3pc, 11
bulirsch-stoer, 44
bvp, 46
cranknic, 66
deeve, 77
deval, 79
euler_heun, 93
newmark, 233
ode23, 248
rk4, rk4sys, 320
rkf54, 321
shooting, 334

*Topic 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, 161
L1linreg, 190
linprog, 202
lsqlincon, 209
nelder_mead, 230
quadprog, 297
shubert, 336
softline, 344
steep_descent, 355

*Topic package
pracma-package, 8

*Topic specfun
bernoulli, 32
elipke,ellipj, 87
eta, 92
expint, 94
fresne1S/C, 126
gammainc, 131
gammaz, 132
lambertWp, 193
Si, Ci, 337
zeta, 381
+Topic **specmat**
    hadamard, 150
    hankel, 153
    hilb, 158
    moler, 223
    pascal, 255
    rosser, 326
    vander, 377
    wilkinson, 380
+Topic **stat**
    erf, 90
    geomean, harmmean, 141
    poisson2disk, 262
    rand, 302
    rmserr, 323
    std, 353
    std_err, 354
+Topic **string**
    blanks, 38
    refindall, 313
    regexp, 314
    regexprep, 315
    strcat, 357
    strcmp, 358
    strfind, 359
    strjust, 360
    strRep, 361
    strTrim, 362
+Topic **timeseries**
    approx_entropy, 24
    conv, 62
    cutpoints, 73
    deconv, 76
    findpeaks, 109
    hampel, 152
    histss, 160
    hurstexp, 167
    invlap, 181
    movavg, 224
    savgol, 330
    whittaker, 379
+Topic **utilities**
    cd, pwd, what, 48
    clear, who(s), ver, 56
    disp, beep, 81
    fprintf, 123
    tic, toc, 366
    abm3pc, 11
    abs, 22
    accumarray, 13
    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
    acsch(sind, cosd, tand, etc.), 341
    acsch(coth, csch, sech, etc.), 65
    affineproj(linearproj, affineproj), 198
    agmean, 15
    aitken, 16, 364
    akimaInterp, 18, 189
    and (and, or), 19
    and, or, 19
    andrewsplot, 20
    angle, 21
    anms, 22
    approx, 178
    approx_entropy, 24
    arclength, 26, 269
    arnoldi, 28
    arrayfun(bsxfun), 43
    arrows, 301, 378
    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, 196, 232, 236
    barylag2d, 31, 189
    beep(disp, beep), 81
    bernoulli, 32
    bernstein, 34
    bernsteinb(bernstein), 34
    bisect, 35
    bits, 37
    blanks, 38
    blkdiag, 38
    brent, 130, 318
    brent(brentDekker), 39
    brentDekker, 39
    brown72, 40
    broyden, 41, 128, 239
    bsxfun, 43
    bubbleSort(sorting), 345
INDEX

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, 197
choose, 227
Ci (Si, Ci), 337
circlefit, 54, 72
circshift (flipdim), 114
clear (clear, who(s), ver), 56
clear, who(s), ver, 56
clenshaw_curtis, 57
combs, 58, 304
compan, 58, 84
complexstep, 59, 247
cond, 61, 244
contour, 99
cov, 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, 234
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), 369
curve, 101
curvefit, 71
cut, 74
cutpoints, 73
dblquad, 74, 341
deblank, 38
deblank (strTrim), 362
deconv, 63, 76
deeve, 77, 80
deg2rad, 78
detrend, 78
deval, 77, 79, 249, 320
Diag, 39, 80, 98, 369, 371
diag, 81, 369
dim, 343
disp (disp, beep), 81
disp, beep, 81
dist, 83, 353
distmat, 82, 154
dot, 68, 69, 83
eig, 84, 85
eigjacobi, 85
einsteinF, 86
ellipj (ellipke, ellipj), 87
ellipke (ellipke, ellipj), 87
eillipke, 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, 381
euler_heun, 93
expint, 94, 338
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, 102
ezpolar, 101
ezsurf (ezcontour, ezsurf, ezmesh), 98
fact, 102
factorial, 102
factorial2 (fact), 102
factors, 103, 185, 282
fderiv, 104, 147, 148, 247, 366
fft, 170
fftshift (ifft), 170
fibsearch, 105, 116
figure, 106
find, 242
findInterval, 159
findintervals, 107
findmins, 108, 111, 336
findpeaks, 109, 152
finds, 110
findstr (strstr), 359
findzeros, 111
Fix (ceil), 49
fletcher_powell, 112, 355
flipdim, 114
fliplr (flipdim), 114
flipud (flipdim), 114
fminbnd, 115
fminsearch, 117, 120
fminunc, 117, 119
fnorm, 121, 329
fornberg, 122
fplot (ezplot), 99
fprintf, 123
fractalcurve, 124
fresnelC (fresnelsC), 126
fresnels (fresnelsS/C), 126
fresnelsS (fresnelsS/C), 126
fsolve, 42, 127
fzero, 129
fzsolve, 130
gamma, 132, 133
gammaman, 131
gammaz, 93, 132, 381
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
gamma_median, 142
geomean (geomean, harmmean), 141
gamma, harmmean, 141
getwd, 49
givens, 143, 150, 165
gmres, 144
golden_ratio, 116, 145
grad, 146
grad_csd (complexstep), 59
gradient, 147
gramSchmidt, 149
gsub, 315, 361
hadamard, 150, 153
halley, 151, 194
hampel, 110, 152
hankel, 150, 153, 368
harmmean (geomean, harmmean), 141
hausdorff_dist, 154
haversine, 155
hepsort (sorting), 345
hessenberg, 29, 156
hessian, 157, 195
hessian_csd (complexstep), 59
hilb, 158
hist, 159, 161
histc, 159, 161
histss, 159, 160
hooke_jeeves, 119, 161, 231
horner, 163
hornerdefl (horner), 163
householder, 144, 150, 157, 165, 288
humps, 166
hurstep, 167
hypot, 169
idivide (mod, rem), 221
ifft, 170
ifftshift (ifft), 170
Imag (angle), 21
image, 99
incgam (gammaman), 131
inpolygon, 171
insertionSort (sorting), 345
integral, 172, 176, 201
integral2, 174
integrate, 75, 290, 295, 325
interp1, 177, 180, 256
interp2, 31, 179
inv, 180
invlap, 181
isNsorted (sorting), 345
isempty, 183
isposdef, 184
isprime, 103, 185, 282
itersolve, 186
jacobian, 187
jacobian_csd (complexstep), 59
kabsch (procrustes), 283
kriging, 19, 188
kron, 189
Lilinreg, 143, 190
lagrangeInterp (newtonInterp), 236
laguerre, 192
lambertWn (lambertWp), 193
lambertWp, 17, 193
laplacian, 158, 194
laplacian_csd (complexstep), 59
Lcm (gcd, lcm), 140
lebesgue, 196
legendre, 197
li (expint), 94
line_integral, 201
linearproj (linearproj, affineproj), 198
linearproj, affineproj, 198
linprog, 202
linspace, 205
list.files, 49
lml, 191, 252
logit (sigmoid), 338
loglog (semilogx, semilogy), 333
logm (expm), 96
logseq (logspace), 206
logspace, 205, 206, 206
ls, 56
lsqcurvefit (lsqnonlin), 210
lsqlin, 72, 207, 210
lsqlincon, 207, 209, 298
lsqnonlin, 191, 210, 219
lsqnonneg (lsqnonlin), 210
lsqneg (lsqnonlin), 210
lu, 214
luact (lu), 214
lusys (lu), 214
magic, 216
matlab, 217
mean, 141
median, 222
mergeOrdered (sorting), 345
mergeSort (sorting), 345
meshgrid, 217, 258
mexpfit, 218
midpoint (bulirsch-stoer), 44
mkpp, 280
mkpp (ppval), 281
mldivide, 220
Mod, 22
mod (mod, rem), 221
mode, rem, 221
Mode, 222
moler, 223
movavg, 224
mround (mldivide, 220
muller, 225
nchoosek, 227
ndims, 228
nearest_spd, 229
nelder mead, 119, 162, 230
neville, 123, 232, 236
newmark, 67, 233
newton (newtonRaphson), 237
newtonHorner, 234, 237
newtonInterp, 123, 232, 236
newtonRaphson, 40, 151, 226, 235, 237, 239
newtons, 42, 130, 138, 226, 238
nextpow2, 37, 239, 279
nile, 240
nlm, 212
nls, 212
nnz, 241
Norm, 121, 242
norm, 242
normest, 62, 243
nthroot, 244
null (nullspace), 245
nullspace, 199, 207, 245, 253, 307
num2str (str2num), 356
numderiv, 60, 105, 246
numdiff (numderiv), 246
numel, 247
ode23, 12, 45, 67, 234, 248, 320, 322
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, 199, 245, 253, 363
outer, 218
pade, 254, 309
pascal, 255
paste, 358
pchip, 256
pchipfun (pchip), 256
pdist (distmat), 82
pdist2 (distmat), 82
peaks, 257
perms, 58, 258, 306
persp, 99
pgamma, 132
piecewise, 259
pinv, 207, 260
plot, 334
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, 27
poly_length (polyarea), 268
polyadd, 63, 266
polyApprox, 51, 267
polyarea, 268, 312, 370
polyder, 270, 273
polyfit, 79, 267
polyfit (polyfit, polyfix), 271
polyfit, polyfix, 271
polyfix (polyfit, polyfix), 271
polygon, 172
polyint, 271, 273
polylog, 273
polymul, 76, 275, 276
polypow, 276, 277
polyroot, 326
polytrans, 277
polyval, 164, 265, 271–273
polyval (polyval, polyvalm), 278
polyval, polyvalm, 278
polyvalm (polyval, polyvalm), 278
pow2, 240, 279
ppfit, 279
ppval, 280, 281
pracma (pracma-package), 8
pracma-package, 8
primes, 95, 103, 185, 282
procrustes, 229, 283
psi, 285
psinc (humps), 166
pwd (cd, pwd, what), 48
qpsolve (qpspecial, qpsolve), 286
qpspecial (qpspecial, qpsolve), 286
qpspecial, qpsolve, 286
qr, 215
qr.solve, 328
qrSolve, 186, 288, 376
quad, 289, 291, 297, 300, 340
quad2d, 75, 290, 341, 375
quadcc, 291
quadgk, 139, 174, 292, 295
quadgr, 174, 293, 325
quadinf, 174, 294
quadl, 290, 296
quadprog, 297
quadv, 174, 299
quickSort (sorting), 345
quickSortx (sorting), 345
quiver, 301, 378
rad2deg (deg2rad), 78
rand, 302
randcomb, 58, 304
randi (rand), 302
randn (rand), 302
randortho, 229, 305
randp (rand), 302
randperm, 258, 304, 306
rands (rand), 302
randsample (rand), 302
Rank, 245, 307
rat, 308
ratinterp, 309, 311
rationalfit, 309, 310
rats (rat), 308
Real (angle), 21
INDEX

rectint, 312
refindall, 313
regexp, 313, 314
regexp1 (regexp), 314
regexp2, 314
regexpexpr, 315
regulaFalsi (bisection), 35
rem (mod, rem), 221
remat, 316
Reshape, 316, 317
ridders, 36, 40, 317
rk4, 12, 322
rk4 (rk4, rk4sys), 320
rk4, rk4sys, 320
rk4sys, 249
rk4sys (rk4, rk4sys), 320
rkf54, 321
rm, 56
rmserr, 323
romberg, 139, 174, 324
rootm (sqrtm, rootm), 350
roots, 59, 192, 265, 278, 325
rosser, 326
rot90, 327
rref, 327
runge, 329

sample_entropy (approx_entropy), 24
sav gol, 330, 380
sec (cot, csc, sec, etc.), 63
secant, 226
secant (bisection), 35
secd (sind, cosd, tand, etc.), 341
sech (coth, csch, sech, etc.), 65
segm_distance, 331, 333
segm_intersect, 331, 332
selectionSort (sorting), 345
semilogx (semilogx, semilogy), 333
semilogx, semilogy, 333
semilogy (semilogx, semilogy), 333
seq, 205, 206
sessionInfo, 56
set.seed, 303
setwd, 49
shellSort (sorting), 345
shooting, 46, 334
shubert, 336
Si (Si, Ci), 337
Si, Ci, 337

sigmoid, 338
signm (sqrtm, rootm), 350
simpadpt, 65, 174, 339
simpson2d, 75, 340, 340, 375
sinc, 338
sinc (humps), 166
sind (sind, cosd, tand, etc.), 341
sind, cosd, tand, etc., 341
size, 228, 248, 343
sofitline, 138, 344
solve, 145, 181
sort, 346, 347
sorting, 345
sortrows, 347
sp2cart (cart2sp), 47
sinterp, 348
spline, 70, 178
sprintf, 124, 357
sqrt, 244
sqrtm (sqrtm, rootm), 350
sqrtm, rootm, 350
squareform, 352
std, 353
std_err, 354
steep_descent, 113, 355
str2num, 356
strcat, 357, 358
strcmpi, 358, 359
strcmp (strcmp), 358
strfind, 359
strfindi (strfind), 359
strjust, 360, 362
strRep, 361
strTrim, 360, 362
subspace, 363
sumalt, 364
supsmu, 380
svd, 62, 244, 284
system.time, 367
tand (sind, cosd, tand, etc.), 341
taylor, 105, 254, 365
testSort (sorting), 345
tic (tic, toc), 366
tic, toc, 366
titanium, 367
toc (tic, toc), 366
Toeplitz, 150, 153, 368, 380
Trace, 81, 369
trapz, 65, 259, 269, 369
trapzfun (trapz), 369
tri, 371
trigApprox, 372, 374
trigPoly, 372, 373
tril (tri), 371
trimmean (geomean, harmmean), 141
triplequad (dblquad), 74
triquad, 374
trisolve, 376
triu (tri), 371
uniq (accumarray), 13
unique, 14
unirroot, 106, 130, 146
vander, 159, 377
vectorfield, 301, 378
Vectorize, 43
ver (clear, who(s), ver), 56
what (cd, pwd, what), 48
whittaker, 330, 379
who (clear, who(s), ver), 56
whos (clear, who(s), ver), 56
wilkinson, 223, 326, 380
zeros (eye), 97
zeta, 33, 93, 381