Package: comphy (via r-universe)

June 2, 2026

Title Functions Used in the Book ``Computational Physics with R''

Version 1.0.5

Description Provides a collection of functions described and used in the book Foadi (2026, ISBN:9780750326308) ``Computational Physics with R''. These include routines for numerical differentiation, integration, differential equations, eigenvalue problems, Monte Carlo methods, and other algorithms relevant to computational physics.

License GPL (>= 2)

Encoding UTF-8

Depends R (>= 3.6.0)

RoxygenNote 7.3.2

URL https://github.com/jfoadi/comphy

BugReports https://github.com/jfoadi/comphy/issues

Repository https://jfoadi.r-universe.dev

Date/Publication 2026-01-28 15:31:11 UTC

RemoteUrl https://github.com/jfoadi/comphy

RemoteRef HEAD

RemoteSha 92af1a92738dceb3e4098e5281a82601d18ef239

Contents

`backdif` *Backward differences*

Description

Computes backward differences of all orders up to n, based on n+1 tabulated points on a regular grid.

Usage

```
backdif(f)
```

Arguments

`f`	A vector of real numbers. Tabulated (known) values of the function, corresponding to a regular grid.

Details

The backward difference of first order is

$$f(x_i) - f(x_i - h)$$

Backward differences of higher orders follow from this one, where the function f is replaced by the backward difference of previous order. All values are contained in a $(n+1) \times (n+1)$ lower triangular matrix.

Value

A lower triangular matrix with $n+1$ rows and $n+1$ columns. The first column includes the tabulated values of the function. The second column includes a zero and the n backward differences of first order. The third column includes two zeros and the $n-1$ forward differences of second order. And so on.

Examples

```
# Tabulated values: f(x) = x^3+x^2-x-1
x <- c(0,1,2,3)
f <- x^3+x^2-x-1

# Triangular matrix with backward differences
B <- backdif(f)
print(B)
```

`BVPlinshoot2` *Linear shooting method for second-order linear BVPs*

Description

Solves a second-order linear boundary value problem using the linear shooting method and superposition of two initial value problems.

Usage

```
BVPlinshoot2(f, t0, tf, y0, yf, h, ...)
```

Arguments

`f`	A function of the form $f(t, y, y')$ representing the second-order ODE: $y'' = f(t, y, y')$.
`t0`	Initial time.
`tf`	Final time.
`y0`	Boundary value at `t0`, i.e. $y(t_0) = y_0$.
`yf`	Boundary value at `tf`, i.e. $y(t_f) = y_f$.
`h`	Step size.
`...`	Optional parameters passed to the gradient function `f`.

Details

If the solution of the associated homogeneous IVP is very small (close to zero) at the second boundary (tf), the solution becomes unstable and the function stops with a warning. Other methods must be used in those cases.

Value

A list with elements `t` (time points) and `y` (solution matrix). The first column of matrix `y` is the solution, $y(t)$, the second is its first derivative, $y'(t)$.

Examples

```
# Solve: y'' - (3/x)y' + (4/x^2)y = x
# with y(1) = 0, y(2) = 4*(log(2) + 1)
# Exact solution: y(x) = x^2*(log(x) - 1) + x^3

# Gradient
f <- function(x,y,dy,...) {
  (3/x)*dy-(4/x^2)*y+x
}

t0 <- 1
tf <- 2
y0 <- 0
yf <- 4*(log(2)+1)
h <- 0.01

ltmp <- BVPlinshoot2(f,t0,tf,y0,yf,h)

# Checks
n <- length(ltmp$t)-1
print(c(ltmp$t[1],ltmp$t[n+1]))
print(c(ltmp$y[1,1],ltmp$y[n+1,1]))
```

`BVPshoot2` *Solves a second-order BVP using the shooting method*

Description

Solves $y^{('')} = f(t, y, y^{'})$ on the interval $[t_0, t_f]$ with boundary conditions $y(t_0) = y_0$, $y(t_f) = y_f$, using the shooting method and a root finder. The associated IVP is solved using 4th-order Runge-Kutta with `RK4ODE`. The second initial value is found using the bisection method with `roots_bisec`.

Usage

```
BVPshoot2(f, t0, tf, y0, yf, h, s_guess = 1, tol = 1e-09, ...)
```

Arguments

f	A function of the form f(t,y,dy) returning a numeric scalar. This defines the second-order ODE.
t0	Initial time.
tf	Final time.
y0	Boundary value at t_0, i.e. $y(t_0) = y_0$.
yf	Boundary value at t_f, i.e. $y(t_f) = y_f$.
h	Step size for the RK4 integration.
s_guess	A numeric starting guess for $y'(t_0)$ (default is 1), or a 2-element numeric vector giving a bracketing interval.
tol	A numeric value that tells the bisection algorithms when to stop. Default is 1e-9.
...	Additional arguments passed to f.

Details

It is important to consider the uniqueness of the solution of a BVP. If the BVP admits infinitely many solutions (a family of solutions), BVPshoot2 will find only one of them, depending on what initial condition for the first derivative of the associated IVP, was found using the bisection method.

Value

A list with elements t (time grid) and y (solution matrix), where y[,1] contains $y(t)$ and y[,2] its derivative.

Examples

```
# y″+ y = 9*sin(2*t); y(0)=-1, y(3*pi/2)=0
# Unique solution: y(t) = 3*sin(2*t) - cos(t)

# Define y″=f(t,y,y')
f <- function(t,y,dy) {ff <- -9*sin(2*t)-y; return(ff)}

# Solution interval
t0 <- 0
tf <- 3*pi/2

# Boundary values
y0 <- -1
yf <- 0

# Step size
h <- 0.01

# Solution
ltmp <- BVPshoot2(f,t0,tf,y0,yf,h)

# Check
# Number of steps
```

```
n <- length(ltmp$t)-1
print(c(ltmp$t[1],ltmp$t[n+1]))
print(c(ltmp$y[1,1],ltmp$y[n+1,1]))
```

---

condet *Determinant of a square matrix*

Description

Calculates the determinant of a square matrix of size n, using the reduction of the matrix in upper triangular form.

Usage

```
condet(A)
```

Arguments

A	An $n \times n$ square matrix of.

Value

A real number corresponding to the determinant of A.

Examples

```
# Identity matrix of size 10
A <- diag(10)
print(condet(A))

# Random matrix with integer elements
A <- matrix(sample(-5:5,size=25,replace=TRUE),ncol=5)
print(condet(A))
```

---

decidepoly_n *Degree of best-interpolating polynomial*

Description

The degree is chosen making use of divided differences. As more and more points are used for the interpolation, the components of columns of divided differences corresponding to higher orders, are smaller and smaller. They are exactly zero when the function to interpolate is a polynomial of degree, say, k. More specifically, all divided differences of order $k + 1$ and above are exactly zero. The criterion used suggests a value k if the average of the absolute value of all divided differences of order $k + 1$ is less than a given threshold `thr` (default 1e-6).

Usage

```
decidepoly_n(x, f, thr = 1e-06, ntrial = 30)
```

Arguments

x	A vector of real numbers. Grid points corresponding to the tabulated (known) values of the function.
f	A vector of real numbers. Tabulated (known) values of the function, corresponding to the grid x.
thr	A real number. This is the threshold to decide when a column in the triangular matrix of divided differences has a small-enough average value (small than thr). Default is thr=1e-6.
ntrial	A positive integer to decide how many random selections of the provided known (tabulated) points have to be carried out. Default is ntrial=30.

Details

The divided differences depend on the specific points selected to calculate the interpolated curve. To avoid potential bias that might occur when the tabulated points used are not distributed uniformly, several random selections of tabulated points are performed (default ntrial=30) and the highest k is returned.

Value

An integer corresponding to the best interpolating polynomial's degree.

Examples

```
# Tabulated grid points for function cos(x)
x <- seq(0,3*pi/2,length=20)
f <- cos(x)

# Suggested polynomial degree (default ntrial)
k <- decidepoly_n(x,f)
print(k)

# Increase number of random selections (ntrial=50)
k <- decidepoly_n(x,f,ntrial=50)
print(k)
```

deriv_irr *First derivative for an irregular grid*

Description

Computes the first derivative at a given point, for a function known only through its values tabulated on an irregular grid, where the distance between successive points of the variable is in general not constant. The algorithm is based on the divided differences (see `divdif`).

Usage

```
deriv_irr(x0, x, f)
```

Arguments

`x0`	A vector of real numbers. These are the values where the first derivative needs to be calculated. These values need to be within the interval defined by the tabulated grid, `x`.
`x`	A vector of real numbers. Grid points corresponding to the tabulated (known) values of the function.
`f`	A vector of real numbers. Tabulated (known) values of the function, corresponding to the grid `x`.

Details

This numerical derivative should be used only when the function is known at specific points. When the analytic form of the function is available and the grid of values of the independent variable can be arbitrarily chosen, then it is better to compute the derivative using other more appropriate and faster methods.

Value

A vector of real numbers. These are the numeric approximations to the first derivative of the function at all values in `x0`. The first derivative is exact when the function is a polynomial of degree n, where n is less than the number of tabulated values.

Examples

```
# Tabulated values: f(x) = 2*x^2-1
x <- c(0,1,3,7)
f <- 2*x^2-1

# The derivative needs to be computed at three values
x0 <- c(1.1,4,6.5)

# First derivatives
f1 <- deriv_irr(x0,x,f)
print(f1)
```

deriv_reg *First derivative on a regular grid*

Description

Computes the first derivative of a function at selected points using the forward difference, backward difference, or centred difference. A regularly spaced grid with corresponding values of the function must be available, as well as a subset of the same grid points at which the derivative must be calculated. For forward and backward differences, the last, respectively the derivative cannot be calculated at the first or last grid point. For centred difference it cannot be calculated at both first and last grid point.

Usage

```
deriv_reg(x0, x, f, scheme = "c")
```

Arguments

x0	A numeric vector. Values at which the derivative is computed. Must be an exact subset of x. Approximated values of x will not be accepted.
x	A numeric vector. Regular grid points where the function is tabulated.
f	A numeric vector. Tabulated values of the function at grid x.
scheme	A one-letter character indicating which difference to use. Possible values are "c", "f", "b" for centred, forward and backward, respectively.

Value

A vector of real numbers. These are the numeric approximations to the first derivative of the function at all values in x0.

Examples

```
x <- seq(0, 1, length.out = 11)
f <- x^3 + x^2 - x - 1
x0 <- c(0.2, 0.5, 0.8)
deriv_reg(x0, x, f)
```

divdif *Divided differences*

Description

Calculation of all the n*(n+1)/2 divided differences related to n tabulated points of a function. The values returned fill half of a n X n matrix, the other half being filled with zeros.

Usage

```
divdif(x, f)
```

Arguments

x	A vector of real numbers. Grid points corresponding to the tabulated (known) values of the function.
f	A vector of real numbers. Tabulated (known) values of the function, corresponding to the grid x.

Value

A matrix of size n X n, where n is the length of x. Each column of this matrix contains the divided differences at a specified level. Thus, column 1 contains the level 1 values, i.e. the n tabulated points, column 2 contains the n-1 divided differences calculated with the adjacent couples of grid points, column 3 contains all n-2 level 3 divided differences, and so on. In each column the remaining slots (no slots in the first column, one slot in the second column, two slots in the third column, etc) are filled with zeros.

Examples

```
# Tabulated values: f(x)=x^3-4x^2+3x+2
x <- c(-1,1,2,4)
f <- c(-6,2,0,14)

# Matrix filled with divided differences and zeros
P <- divdif(x,f)
print(P)

# Add two tabulated points to previous set
x <- c(x,0,3)
f <- c(f,2,2)

# New divided differences appear, but
# the old ones are unchanged
P <- divdif(x,f)
print(P)
```

`EPSturmLiouville2` *Sturm–Liouville eigenproblem with homogeneous Dirichlet boundary conditions*

Description

Solves

$$-\frac{d}{dx}\left(p(x)\,y'(x)\right) + q(x)\,y(x) = \lambda\,w(x)\,y(x)$$

on $[a, b]$ with $y(a) = 0$ and $y(b) = 0$. The equation is discretised on the interior nodes of a **uniform** grid and assembled into matrices `K` and `W` so that `K u = lambda W u`. The problem is reduced to a symmetric standard eigenproblem and solved.

Usage

```
EPSturmLiouville2(
  p,
  q,
  w,
  x,
  nev = NULL,
  normalize = TRUE,
  return_matrices = FALSE,
  check_inputs = TRUE,
  tol_uniform = 1e-12
)
```

Arguments

`p`	Function `p(x)` or numeric vector at midpoints.
`q`	Function `q(x)` or numeric vector at nodes.
`w`	Function `w(x)` or numeric vector at nodes.
`x`	Numeric grid including endpoints (`x[1]=a`, `x[n+1]=b`); must be uniform.
`nev`	Integer number of eigenpairs to return (smallest); default all interior modes.
`normalize`	Logical; if `TRUE`, scale interior eigenvectors so that $\sum_i h\,w_i\,u_i^2 = 1$. Default `TRUE`.
`return_matrices`	Logical; if `TRUE`, also return `K` and `W`. Default `FALSE`.
`check_inputs`	Logical; run basic checks (uniform grid, positivity of `p`, `w`). Default `TRUE`.
`tol_uniform`	Tolerance for uniform-grid check. Default `1e-12`.

Details

Coefficients may be given as functions or numeric vectors:

- p: function on midpoints or numeric vector of length length(x)-1 (midpoints).
- q, w: functions on nodes or numeric vectors of length length(x) (nodes).

Homogeneous Dirichlet conditions are enforced by construction: unknowns are interior only; the returned full eigenfunctions have zero endpoints.

Value

A list with

- values: eigenvalues (ascending).
- vectors_interior: interior eigenvectors (matrix (n-1) x k).
- vectors_full: full eigenfunctions with zero endpoints (matrix (n+1) x k).
- x, h, nev_used.
- K, W if return_matrices=TRUE.

Examples

```
# p=1, q=0, w=1 on [0, pi]  -> eigenvalues ~ 1^2, 2^2, 3^2, ...
a <- 0; b <- pi; n <- 200
x <- seq(a, b, length.out = n+1)
pfun <- function(s) 1            # scalars are accepted; will be replicated
qfun <- function(s) 0
wfun <- function(s) 1
ep <- EPSturmLiouville2(pfun, qfun, wfun, x, nev = 4, normalize = TRUE)
round(ep$values, 3)              # ~ c(1, 4, 9, 16)
```

EulerODE *Euler method for systems of ODEs*

Description

Solves a system of m first-order ODEs using the explicit Euler method.

Usage

```
EulerODE(f, t0, tf, y0, h, ...)
```

Arguments

f	A function of the form f(t,y) returning a numeric vector. It must be defined before using EulerODE. This function is the right hand side of the ODE, i.e. the gradient of the ODE system.
t0	Initial time.
tf	Final time.
y0	A numeric vector with initial values (length = m).
h	Step size.
...	Other parameters potentially needed by the gradient function.

Details

The method is accurate and stable when the stepsize h is relatively small. The local error is $O(h^2)$, while the global error is $O(h)$. Other numerical methods are generally used to calculate solutions with a higher accuracy.

Value

A list with elements t (time points) and y (solution matrix). The first row of the matrix contains the initial values of y at time t0. Each column of the matrix contains the numerical solution for each one of the m functions of the system of ODEs.

Examples

```
# IVP: \eqn{dy/dt=6-2y,\ y(0)=0}.
# Define gradient
f <- function(t,y) {dy <- 6-2*y; return(dy)}

# Solution interval
t0 <- 0
tf <- 2

# Initial condition
y0 <- 0

# Step
h <- 0.1

# Numerical solution
ltmp <- EulerODE(f,t0,tf,y0,h)

# Print grid
print(ltmp$t)

# Print numerical solution
print(ltmp$y)

# Example with two ODEs.
# \eqn{dy_1/dt=y_1+2y_2}
```

```
# \eqn{dy_2/dt=(3/2)y_1-y_2}
# \eqn{y_1(0)=1, y_2(0)=-2}

# Define gradient
dy <- function(t,y) {
  dy1 <- y[1]+2*y[2]
  dy2 <- 1.5*y[1]-y[2]
  return(c(dy1,dy2))
}

# Solution interval
t0 <- 0
tf <- 2

# Initial conditions
y0 <- c(1,-2)

# Step
h <- 0.1

# Numerical solution
ltmp <- EulerODE(dy,t0,tf,y0,h)

# Print grid
print(ltmp$t)

# Print numerical solution y1
print(ltmp$y[,1])

# Print numerical solution y2
print(ltmp$y[,2])
```

`forwdif` *Forward differences*

Description

Computes forward differences of all orders up to n, based on n+1 tabulated points on a regular grid.

Usage

```
forwdif(f)
```

Arguments

`f`	A vector of real numbers. Tabulated (known) values of the function, corresponding to a regular grid.

Details

The forward difference of first order is

$$f(x_i + h) - f(x_i)$$

Forward differences of higher orders follow from this one, where the function f is replaced by the forward difference of previous order. All values are contained in a $(n+1) \times (n+1)$ upper triangular matrix.

Value

An upper triangular matrix with $n + 1$ rows and $n + 1$ columns. The first column includes the tabulated values of the function. The second column includes the n forward differences of first order and a zero. The third column includes the $n - 1$ forward differences of second order and two zeros. And so on.

Examples

```
# Tabulated values: f(x) = x^3+x^2-x-1
x <- c(0,1,2,3)
f <- x^3+x^2-x-1

# Triangular matrix with forward differences
F <- forwdif(f)
print(F)
```

`gauss_elim` *Gaussian Elimination*

Description

Solution of a system of n equations in n unknowns, using Gaussian elimination.

Usage

```
gauss_elim(M)
```

Arguments

`M`	The $n \times (n + 1)$ augmented matrix of coefficients corresponding to the system of n linear equations in n unknowns, $Ax = b$.

Details

The linear system to solve is $Ax = b$, where A is the $n \times n$ matrix of coefficients of the n unknowns in the $n \times 1$ vector x, and b is the $n \times 1$ vector of known numbers. Gaussian elimination consists of a series of so-called row operations that transform A in an upper-triangular matrix. The system corresponding to the transformed matrix can be solved very quickly.

Value

A vector of length n containing the n numeric solutions for the n unknowns. If the system has no solutions or an infinite number of solutions, the function returns NULL and dumps a warning message.

Examples

```
# System of three equations in three unknowns
#
# 3x_1 + x_2 +  x_3 = 6
#  x_1 - x_2 + 2x_3 = 4
# -x_1 + x_2 +  x_3 = 2

# Augmented matrix M=(A|b)
M <- matrix(c(3,1,-1,1,-1,1,1,2,1,6,4,2),ncol=4)

# Solution via Gauss elimination
x <- gauss_elim(M)
print(x)
```

Gquad *Numerical integration using n-point Gaussian quadrature.*

Description

Computes the definite integral of $f(x)$ between a and b, using the method of Gaussian quadrature. The default number of points is

Usage

```
Gquad(f, a = -1, b = 1, n = 5)
```

Arguments

f	A function to integrate.
a	Lower bound of integration (default -1).
b	Upper bound of integration (default 1).
n	Number of quadrature points (default 5).

Value

A list with three elements. The first is a numeric vector containing the nodes of the quadrature. The second is a numeric vector containing the corresponding weight. The third is a real number corresponding to the approximate value of the integral.

Examples

```
# Integral in [-1,1] of 2x-1.
# Value is -2 and n=1 is enough for exact result

# Define the function
f <- function(x) {ff <- 2*x-1; return(ff)}

# 1-point quadrature
ltmp <- Gquad(f,-1,1,n=1)

# The only zero is x1=0
print(ltmp$xt)

# The only weight is w1=2
print(ltmp$wt)

# Quadrature gives exact integral
print(ltmp$itg)

# 2-point quadrature
ltmp <- Gquad(f,-1,1,n=2)
print(ltmp) # Same result but more zeros and weights

# Default, n=5, is accurate enough
ltmp <- Gquad(exp,-1,1)
print(ltmp$itg)

# Different extremes of integration
ltmp <- Gquad(exp,1,4)
print(ltmp$itg)
```

`GSeidel` *The Gauss-Seidel algorithm*

Description

Implementation of the Gauss-Seidel iterative method to solve a system $Ax = b$ of n linear equations in n unknowns.

Usage

```
GSeidel(A, b, x0 = NULL, tol = 1e-06, nmax = 1e+05, ddominant = TRUE)
```

Arguments

`A`	The $n \times n$ matrix of coefficients of the unknowns in the linear system.

b	A vector of n constants representing the right-hand side of the linear system. This function does not work out solutions of homogeneous systems, where the b is a vector of zeros (null vector). Therefore input with b equal to a null vector is rejected.
x0	A vector of n starting numeric values for the iterations. If no values are entered for x0, a column of zero will be adopted by default.
tol	A real number indicating the threshold under which the relative increment from one solution approximation to the next is small enough to stop iteration. The default value is tol=1e-6.
nmax	An integer. The maximum number of iterations allowed, if convergence according to the criterion is not reached.
ddominant	A logical variable. If FALSE, the method is applied also if the matrix of coefficients is not diagonally dominant (default is TRUE).

Details

Gauss-Seidel is a variant of the Jacobi method, as it guarantees a finite solution for linear systems characterised by a diagonally dominant matrix A of coefficients. This means that each element on its diagonal must be, in absolute value, larger than the sum of the absolute value of all the elements in the corresponding row. Gauss-Seidel differs from Jacobi as it promises to converge faster than Jacobi.

Value

A numeric vector of length n with values approximating the system's solution.

Examples

```
# Simple system with solution 1,2,3
A <- matrix(c(3,1,2,-1,-4,2,1,1,7),ncol=3)
b <- c(4,-4,27)

# Solution
x <- GSeidel(A,b)
print(x)

# Start from a different point
x0 <- c(-1,2,8)
x <- GSeidel(A,b,x0)
print(x)
```

HeunODE *Heun method for systems of ODEs*

Description

Solves a system of m first-order ODEs using the Heun method (also known as the improved Euler method).

Usage

```
HeunODE(f, t0, tf, y0, h, ...)
```

Arguments

`f`	A function of the form `f(t,y)` returning a numeric vector. It must be defined before using `HeunODE`. This function is the right hand side of the ODE, i.e. the gradient of the ODE system.
`t0`	Initial time.
`tf`	Final time.
`y0`	A numeric vector with initial values (length = m).
`h`	Step size.
`...`	Other parameters potentially needed by the gradient function.

Details

The method improves upon the Euler method by using an average of the slopes at the beginning and end of each time step. It is more accurate, with local error $O(h^3)$ and global error $O(h^2)$.

Value

A list with elements `t` (time points) and `y` (solution matrix). The first row of the matrix contains the initial values of `y` at time `t0`. Each column of the matrix contains the numerical solution for each one of the m functions of the system of ODEs.

Examples

```
# IVP: \eqn{dy/dt=6-2y,\ y(0)=0}.
# Define gradient
f <- function(t,y) {dy <- 6-2*y; return(dy)}

# Solution interval
t0 <- 0
tf <- 2

# Initial condition
y0 <- 0

# Step
h <- 0.1

# Numerical solution
ltmp <- HeunODE(f,t0,tf,y0,h)
```

```
# Print grid
print(ltmp$t)

# Print numerical solution
print(ltmp$y)

# Example with two ODEs.
# \eqn{dy_1/dt=y_1+2y_2}
# \eqn{dy_2/dt=(3/2)y_1-y_2}
# \eqn{y_1(0)=1, y_2(0)=-2}

# Define gradient
dy <- function(t,y) {
  dy1 <- y[1]+2*y[2]
  dy2 <- 1.5*y[1]-y[2]
  return(c(dy1,dy2))
}

# Solution interval
t0 <- 0
tf <- 2

# Initial conditions
y0 <- c(1,-2)

# Step
h <- 0.1

# Numerical solution
ltmp <- HeunODE(dy,t0,tf,y0,h)

# Print grid
print(ltmp$t)

# Print numerical solution y1
print(ltmp$y[,1])

# Print numerical solution y2
print(ltmp$y[,2])
```

`illcond_sample` *Ill-conditioned sampling*

Description

Random sampling, based on the uniform distribution, of the right-hand side, b, of a linear system and of a perturbation, Δb, so that the solution of $Ax = b$ is very different from the solution of $Ax = b + \Delta b$.

Usage

```
illcond_sample(A, bmax = 100, Dbmax = 1, ncyc = 1e+05, iseed = NULL)
```

Arguments

A	The $n \times n$ matrix of coefficients of the unknowns in the linear system.
bmax	A numeric number providing the interval, (0,bmax), in which the n uniformly random components of b are selected. Default value is 100.
Dbmax	A numeric number providing the interval, (0,Dbmax), in which the n uniformly random components of Δb are selected. Default value is 1.
ncyc	An integer indicating the number of uniform random selection of the n components of b and the n components of Δb. The higher this number, the higher the chance of getting a high relative solution number, but the longer the execution time of the function. Default is 100000.
iseed	An integer. The seed starting random generation. If a value is provided, the (pseudo-)random generation will reproduce exactly the same b and Δb. Default is NULL, which means that the seed will be randomly chosen at every execution of the function.

Details

The degree of ill-conditioning of a system is not only measured by the matrix's condition number, but also from the solution relative error. If Δx is the difference between the solution, x, of the system related to b and the solution, x', of the system related to $b' = b - \Delta b$, then the ratio of the norm of Δx and the norm of x, is the solution relative error. Norms are Frobenius norms. This function returns a named list with b and Db the chosen b and Δb, based on random sampling of a specified region.

Value

A named list with names b, a vector equal of the right-hand side of the linear system, and Db, a vector equal to the perturbations, Δb, to be applied to b.

Examples

```
# This is a simple but ill-conditioned matrix
A <- matrix(c(2,1,1.99,1),ncol=2)

# Select b and Db randomly, starting with iseed=2341
ltmp <- illcond_sample(A,iseed=2341)
names(ltmp)

# b and b'
b <- ltmp$b
Db <- ltmp$Db
b2 <- b-Db

# Solution for b
x <- solve(A,b)
```

```
print(x)

# Solution for b'
x2 <- solve(A,b2)
print(x2)

# Difference
Dx <- x-x2

# Solution relative error (Frobenius norm)
print(norm(Dx,"F")/norm(x,"F"))

# Upper limit
Ainv <- solve(A)
print(norm(A,"F")*norm(Ainv,"F")*norm(Db,"F")/norm(b,"F"))
```

`linpol` *1D linear interpolation*

Description

Classic linear interpolation between two tabulated (known) points of a one-variable function.

Usage

```
linpol(x, f, x0)
```

Arguments

`x`	A vector of real numbers. Grid points corresponding to the tabulated (known) values of the function.
`f`	A vector of real numbers. Tabulated (known) values of the function, corresponding to the grid `x`.
`x0`	A vector of real numbers. These are the grid points chosen for the interpolation. All points of this grid need to be within the tabulated grid.

Value

A vector of real numbers. These are the actual interpolated values (calculated using linear interpolation), corresponding to all values of the grid `x0`.

Examples

```
# Tabulated values: f(x) = 2*x^2-1
x <- c(0,1,3,7)
f <- 2*x^2-1

# Grid for interpolation
```

```
x0 <- seq(0,7,length=501)

# Interpolated points
f <- linpol(x,f,x0)
print(f)
```

LUdeco *LU decomposition*

Description

Transform an $n \times n$ matrix into a product of a lower-triangular and upper-triangular matrices, using the Crout (method="crout" - default) or Doolittle (method= "doolittle") method.

Usage

```
LUdeco(A, method = "crout")
```

Arguments

A	An $n \times n$ matrix.
method	A character string. This calls two different procedures for the decomposition. Only "crout" (default) and "doolittle" are recognised methods. A different character string forces the function to return NULL.

Details

The "crout" method returns the upper triangular matrix, U, with ones on its diagonal. The "doolittle" method returns the lower triangular matrix, L, with ones on its diagonal.

Some matrices do not have an LU decomposition unless a row permutation is done to the matrix. In this function, the order of such a permutation is included in the named vector ord, returned as part of the output. When the vector is equal to 1,2,...,n (first n numbers, naturally ordained), this means that there was no need of permuting the original matrix to carry out the LU decomposition.

Value

A named list with the lower triangular, L, upper triangular, U, matrices, and with a vector, ord, containing the permutation needed to achieve the LU factorisation.

Examples

```
# 3X3 matrix
#
# [ 3  1  1
#   1 -1  2
#  -1  1  1]
```

```
# Input matrix
A <- matrix(c(3,1,-1,1,-1,1,1,2,1),ncol=3)

# LU decomposition
ltmp <- LUdeco(A)
print(ltmp$L)
print(ltmp$U)
print(ltmp$ord) # No permutation needed

# The product is the original matrix, A
print(ltmp$L%*%ltmp$U)

# Singular matrix with LU decomposition
A <- matrix(c(1,0,0,0,1,1,1,0,0),ncol=3)
print(det(A))
ltmp <- LUdeco(A,"doolittle")
print(ltmp$L)
print(ltmp$U)
print(ltmp$ord) # No permutation needed

# The product is the original matrix, A
print(ltmp$L%*%ltmp$U)

# Singular matrix without LU decomposition
A <- matrix(c(1,0,0,0,0,0,0,0,0),ncol=3)
ltmp <- LUdeco(A)
print(ltmp)
#
```

`nevaitpol` *Neville-Aitken algorithm for polynomial interpolation*

Description

Hierarchical series of linearly-interpolated P_{ij} values calculated using Neville-Aitken's algorithm. In the P_{ij} expression, j is the level of the algorithm and i the leftmost grid-point of the tabulated function points.

Usage

```
nevaitpol(x, f, x0)
```

Arguments

`x`	A vector of real numbers. Grid points corresponding to the tabulated (known) values of the function.
`f`	A vector of real numbers. Tabulated (known) values of the function, corresponding to the grid `x`.
`x0`	A vector of real numbers. These are the grid points chosen for the interpolation. All points of this grid need to be within the tabulated grid.

Value

An upper triangular matrix of size n containing the linearly-interpolated values. `P[i,j]` is zero for $i + j > n + 1$.

Examples

```
# Tabulated values: f(x) = x^3-2*x^2+3*x-1
x <- c(0.1,0.4,0.6,0.8,0.9)
f <- x^3-2*x^2+3*x-1

# Interpolation point
x0 <- 0.75

# Upper-triangular matrix of N-A values
P <- nevaitpol(x,f,x0)

# From level 4 onward the interpolated value
# does not change because f(x) is a 3rd-degree polynomial
print(P)
```

`numint_reg` *Numerical integration using the trapezoid or simpson's rule*

Description

Computes the definite integral of $f(x)$ between a and b, using one of the three numerical integration Newton-Cotes rules, trapezoid, Simpson's 1/3 or Simpson's 3/8.

Usage

```
numint_reg(x, f, scheme = "sim13")
```

Arguments

`x`	A vector of real numbers. Grid points corresponding to the tabulated (known) values of the function. These must be equally spaced (regular grid).
`f`	A vector of real numbers. Tabulated (known) values of the function, corresponding to the grid `x`.
`scheme`	A character indicating the integration rule to follow. Possible values are "trap" (trapezoid rule), "sim13" (Simpson's 1/3 rule), and "sim38" (Simpson's 3/8 rule). Default scheme is "sim13".

Details

The default method is Simpson's 1/3 rule. For this method to be applied correctly, the number of regular intervals must be even. If this is not the case, the area corresponding to the last interval will be calculated with the trapezoid rule with a warning being posted.

When using the Simpson's 3/8 rule, the number of regular intervals must be a multiple of 3. If this is not the case, the last or last two intervals will be computed with the trapezoid rule.

Value

A real number, corresponding to the numeric approximation of the definite integral of $f(x)$.

Examples

```
# Tabulated values: f(x) = x^2
x <- seq(0,2,length.out=21) # number of intervals is even
f <- x^2

# Integral between 0 and 2
# The correct result is 2^3/3=8/3=2.6666...
nvalue <- numint_reg(x,f) # Defaul nethod simpson's 1/3
print(nvalue)

# If the number of intervals is not even,
# a warning is issued
y <- seq(0,2,length.out=22)
g <- y^2
nvalue <- numint_reg(y,g)
print(nvalue)
```

---

oddity *Parity of a permutation*

Description

Given a permutation of the integers from 1 to n, this function calculates its parity (+1 or -1), i.e. the number of swapping that take the permutation back to the natural ordering $1, 2, \ldots, n$.

Usage

```
oddity(x)
```

Arguments

x	A vector containing a permutation of the first n integers, $1, 2, \ldots, n$.

Value

A real number equal to +1 or -1, indicating the parity of the given permutation.

Examples

```
# Identity permutation (10 elements)
x <- 1:10
print(oddity(x))

# One swap
x[2] <- 5
x[5] <- 2
print(oddity(x))
```

PJacobi *The Jacobi method*

Description

Implementation of the Jacobi iterative method to solve a system $Ax = b$ of n linear equations in n unknowns.

Usage

```
PJacobi(A, b, x0 = NULL, tol = 1e-06, nmax = 1e+05, ddominant = TRUE)
```

Arguments

A	The $n \times n$ matrix of coefficients of the unknowns in the linear system.
b	A vector of n constants representing the right-hand side of the linear system. This function does not work out solutions of homogeneous systems, where the b is a vector of zeros (null vector). Therefore input with b equal to a null vector is rejected.
x0	A vector of n starting numeric values for the iterations. If no values are entered for x0, a column of zero will be adopted by default.
tol	A real number indicating the threshold under which the relative increment from one solution approximation to the next is small enough to stop iteration. The default value is tol=1e-6.
nmax	An integer. The maximum number of iterations allowed, if convergence according to the criterion is not reached.
ddominant	A logical variable. If FALSE, the method is applied also if the matrix of coefficients is not diagonally dominant (default is TRUE).

Details

The Jacobi method guarantees a finite solution for linear systems characterised by a diagonally dominant matrix A of coefficients. This means that each element on its diagonal must be, in absolute value, larger than the sum of the absolute value of all the elements in the corresponding row.

Value

A numeric vector of length n with values approximating the system's solution.

Examples

```
# Simple system with solution 1,2,3
A <- matrix(c(3,1,2,-1,-4,2,1,1,7),ncol=3)
b <- c(4,-4,27)

# Solution
x <- PJacobi(A,b)
print(x)

# Start from a different point
x0 <- c(-1,2,8)
x <- PJacobi(A,b,x0)
print(x)
```

`polydivdif` *Approximating polynomial for divided differences*

Description

Calculation of a polynomial of order n via divided differences. All n tabulated points provided (with n greater or equal than 2) are used by default for the calculation, but the option is available to use only np points, where np must be greater or equal than 2. In case only part of the n available tabulated points is used (np < n), the first two points are fixed to be equal to the smallest and largest tabulated grid x points; the remaining np-2 points are selected randomly among the n-2 remaining ones.

Usage

```
polydivdif(x0, x, f, np = length(x))
```

Arguments

`x0`	A vector of real numbers. These are the grid points chosen for the interpolation.
`x`	A vector of real numbers. Grid points corresponding to the tabulated (known) values of the function.
`f`	A vector of real numbers. Tabulated (known) values of the function, corresponding to the grid `x`.
`np`	An integer. The number of known points used for the interpolation. np > 2 because the smallest and largest value of `x` have to be always among the known points. Aside from the points at the extremes of the interpolation interval, the other points are chosen randomly.

Value

A named list of length 3 and names `x`, `f` and `f0`.

`x` Tabulated grid points used for the interpolation.

`f` Tabulated function points used for the interpolation. They correspond to `x`.

`f0` Interpolated values. They correspond to the input vector `x0`.

Examples

```
# Tabulated grid points for function sin(x)
x <- seq(0,3*pi/2,length=20)
f <- sin(x)

# Grid of interpolated points
x0 <- seq(0,3*pi/2,length=200)

# Interpolation using all 20 tabulated points
ltmp <- polydivdif(x0,x,f)
plot(ltmp$x,ltmp$f,pch=16)
points(x0,ltmp$f0,type="l")

# Interpolation using only five points (dangerous!)
ltmp <- polydivdif(x0,x,f,np=5)
points(ltmp$x,ltmp$f,col=2,cex=1.5)
points(x0,ltmp$f0,type="l",col=2)
```

---

polysolveLS *Polynomial Least Squares*

Description

Find the parameters, $a_1, \ldots, a_{m+1}$, of the polynomial model of degree m (1D function), using the least squares technique on a group of n data points.

Usage

```
polysolveLS(pts, m, tol = NULL)
```

Arguments

`pts`	A $n \times 2$ matrix or data frame where each row contains the coordinates of a data point used for regression.
`m`	An integer. The degree of the polynomial to be used as model for the regression.

`tol` A real number. The solution of a linear system can be compromised when the condition number of the matrix of coefficients is particularly high (ill-conditioned matrices). `tol` is the reciprocal of the condition number. For values of tol smaller than 1e-17, ill-conditioning is deemed to be sever enough not to guarantee an accurate solution. For such values the function stops execution, returning an error message. In fact, the solution can still be accurate, notwithstanding ill-conditioning, and the user can force the calculation of a solution using a value of `tol` smaller than 1e-17. Default is `NULL`, corresponding to a `tol=1e-17`.

Details

The polynomial model has the following analytic form:

$$y = a_1 x^m + a_2 x^{m-1} + \cdots + a_m x + a_{m+1}$$

The n data points are contained in a matrix or data frame with 2 columns, containing the coordinates of each data point, and n rows. The least squares procedure is carried out as solution of a matrix equation, via the `solveLS` function.

Value

A named list with two elements:

a A vector of length m containing the m numeric values of the estimated polynomial's coefficients.If more than one solution is possible, (infinite-solutions case) the function returns a `NULL` and prints out a related message.

SSE A real number. The numerical value of the sum of squared residuals.

Examples

```
# 21 points close to the quadratic x^2 - 5*x + 6
x <- seq(-2,5,length=21)
set.seed(7766)
eps <- rnorm(21,mean=0,sd=0.5)
y <- x^2-5*x+6+eps

# Data frame
pts <- data.frame(x=x,y=y)

# Regression
ltmp <- polysolveLS(pts,m=2)
print(names(ltmp))
print(ltmp$a)
print(ltmp$SSE)
```

RK4ODE *Runge-Kutta 4th order method for systems of ODEs*

Description

Solves a system of m first-order ODEs using the classical fourth-order Runge-Kutta method.

Usage

```
RK4ODE(f, t0, tf, y0, h, ...)
```

Arguments

f	A function of the form f(t,y) returning a numeric vector. It must be defined before using RK4ODE. This function is the right hand side of the ODE, i.e. the gradient of the ODE system.
t0	Initial time.
tf	Final time.
y0	A numeric vector with initial values (length = m).
h	Step size.
...	Other parameters potentially needed by the gradient function.

Details

This method achieves high accuracy by evaluating the gradient function four times per step. It has local error $O(h^5)$ and global error $O(h^4)$. It is one of the most widely used methods for solving initial value problems numerically.

Value

A list with elements t (time points) and y (solution matrix). The first row of the matrix contains the initial values of y at time t0. Each column of the matrix contains the numerical solution for each one of the m functions of the system of ODEs.

Examples

```
# IVP: \eqn{dy/dt=6-2y,\ y(0)=0}.
# Define gradient
f <- function(t,y) {dy <- 6-2*y; return(dy)}

# Solution interval
t0 <- 0
tf <- 2

# Initial condition
y0 <- 0
```

```
# Step
h <- 0.1

# Numerical solution
ltmp <- RK4ODE(f,t0,tf,y0,h)

# Print grid
print(ltmp$t)

# Print numerical solution
print(ltmp$y)

# Example with two ODEs.
# \eqn{dy_1/dt=y_1+2y_2}
# \eqn{dy_2/dt=(3/2)y_1-y_2}
# \eqn{y_1(0)=1, y_2(0)=-2}

# Define gradient
dy <- function(t,y) {
  dy1 <- y[1]+2*y[2]
  dy2 <- 1.5*y[1]-y[2]
  return(c(dy1,dy2))
}

# Solution interval
t0 <- 0
tf <- 2

# Initial conditions
y0 <- c(1,-2)

# Step
h <- 0.1

# Numerical solution
ltmp <- RK4ODE(dy,t0,tf,y0,h)

# Print grid
print(ltmp$t)

# Print numerical solution y1
print(ltmp$y[,1])

# Print numerical solution y2
print(ltmp$y[,2])
```

---

roots_bisec *Bisection method for roots*

Description

Find the zero of a function of one variable, $f(x)$, given a starting value close to the zero, or an interval including the zero.

Usage

```
roots_bisec(
  fn,
  x0 = 0,
  lB = NULL,
  rB = NULL,
  tol = 1e-09,
  imax = 1e+06,
  eps = 0.1,
  message = TRUE,
  logg = FALSE,
  ...
)
```

Arguments

fn	A function of one variable. If the function includes variables, these will have to be passed as additional variables, using the same names in the function's definition (see examples).
x0	A numeric variable. The starting value to find the initial interval, when a search interval is not provided. The default value is 'x0=0'.
lB	A numeric variable indicating the lower (left) extreme of the search interval. If not given, this number will be selected starting from 'x0' and in small steps 'eps' of values smaller than 'x0', until a value of 'lB' is found for which the function 'f' has sign opposite of the sign it has at 'rB'. Default is for 'lB' not to be entered ('lB=NULL').
rB	Same as 'lB', but corresponding to the upper (right) extreme of the search interval. Default is for 'rB' not to be entered ('rB=NULL').
tol	A real number, in general a small number. The width of the smallest interval containing the zero of the function just before the algorithm stops. This means that the largest error $\|x - x_t\|$ between the numerical value of the root found, x, and its correct value, x_t, is tol. Default value is 1e-9.
imax	A positive integer. The maximum number of bisections of the interval, while searching the zero of the function. The default value is 1e6, although convergence is normally obtained with a number of bisections much smaller than 'imax'. 'imax' is important to stop search in those cases in which the function has no zeros in the search interval provided.
eps	A real number. The step size needed for the selection of a search interval, when this is not provided. In such a situation, symmetric intervals with increasing width around 'x0' are considered where the left and right extremes are 'x0-i*eps' and 'x0+i*eps', respectively, where 'i' is a positive integer, progressively increasing from 1 to the maximum allowed value 'imax'. Search for the selected

	interval stops when is the signs of the function 'f' calculated at the extremes are opposite. If the search interval is not found, a warning message is printed and NULL is returned. Default value is 0.1.
message	A logical variable to state whether messages about the root and the largest error have to be printed. The default is for the messages to be printed (message=TRUE).
logg	A logical variable to state whether information on the series of bisected intervals is printed (TRUE) or not (FALSE). Default is for such information not to be printed (FALSE).
...	Parameters passed to function 'fn', if needed.

Details

Finding the zero of $f(x)$ is equivalent to finding the roots of the equation:

$$f(x) = 0$$

The algorithm used is based on the bisection method that needs an initial interval within which the root is supposed to reside. When multiple roots are involved, the algorithm will only find one among those inside the chosen interval. The algorithm can be started also with just one value, x0, supposedly close to the wanted root. In this case, an interval is selected so that the function at the extremes of the interval has opposite signs. If such an interval is not found, the function dumps a warning message and returns NULL. The bisection method has a slow convergence rate and it does not converge at all in specific situations.

Value

A numeric value, the zero of the function (or, equivalently, the root of the equation $f(x) = 0$).

Examples

```
# The quadratic equation x^2-5*x+6=0 has two roots, 2 and 3
ff <- function(x) return(x^2-5*x+6)

# Find root 2, starting from a single point close to 2
x0 <- 1
x <- roots_bisec(ff,x0=1)
print(x)

# Find root 3, using an interval (no message printing)
x <- roots_bisec(ff,lB=2.8,rB=4,message=FALSE)
print(x)

# Function with a parameter f(x) = exp(x) - k
ff <- function(x,k=2) return(exp(x)-k)

# Solution of exp(x)=3 is log(3)
x <- roots_bisec(ff,k=3)
print(log(3))
```

roots_newton *Newton method for roots*

Description

Find the zero of a function of one variable, $f(x)$, given a starting value close to the zero, using Newton method.

Usage

```
roots_newton(
  f0,
  f1,
  x0 = 0,
  tol = 1e-09,
  imax = 1e+06,
  ftol = NULL,
  message = TRUE,
  logg = FALSE,
  ...
)
```

Arguments

f0	A function of one variable. If the function includes variables, these will have to be passed as additional variables, using the same names in the function's definition (see examples).
f1	A function equal to the first derivative of 'f0'. Parameters that are potentially included in 'f0', must be also included in 'f1'.
x0	A numeric variable. The initial guess starting Newton's algorithm.
tol	A real small number. The smallest difference between the new zero's approximation and the previous one, above which the algorithm keeps working. As soon as the difference is less than 'tol', the algorithm stops and the current approximation is returned as the final approximation to the function's root. Default value is 1e-9.
imax	A positive integer. The maximum number of iterations of the algorithm. The default value is 1e6, although convergence is normally obtained with a number of iterations much smaller than imax. imax is important to stop search in those cases in which the algorithm gets stuck in endless loops (non-convergence).
ftol	A real small number. When 'ftol' is not NULL (default value), Newton's algorithm stops when $\|f(x)\| < ftol$. This parameter essentially introduces a different stopping criterion.
message	A logical variable to state whether messages about the root and the error have to be printed. The default is for the messages to be printed ('message=TRUE').

logg	A logical variable to state whether information on the series of approximating roots is printed (TRUE) or not (FALSE). Default is for such information not to be printed (FALSE).
...	Parameters passed to the two functions 'f0' and 'f1', if any.

Details

Finding the zero of $f(x)$ is equivalent to finding the roots of the equation:

$$f(x) = 0$$

The algorithm used is based on Newton method that needs an initial guess, x0, and the analytic expression of the function's first derivative. The method has a much faster convergence rate than both the bisection and secant methods, but it does not converge when the initial guess or any other subsequent approximations accidentally coincide with an optimal point of the function, i.e. a point at which the first derivative is zero. The algorithm can also potentially be stuck in an endless loop of repeating values for special combinations of functions and initial guess.

Value

A numeric value, the zero of the function (or, equivalently, the root of the equation $f(x) = 0$).

Examples

```
# The quadratic equation x^2-5*x+6=0 has two roots, 2 and 3
f0 <- function(x) return(x^2-5*x+6)

# First derivative
f1 <- function(x) return(2*x-5)

# Find root 2, starting from a single point close to 2
x0 <- 1
x <- roots_newton(f0,f1,x0=1)
print(x)

# Find root 3 (no message printing)
x <- roots_newton(f0,f1,x0=4,message=FALSE)
print(x)

# Function with a parameter f(x) = exp(kx) - 2
f0 <- function(x,k=2) return(exp(k*x)-2)

# First derivative (it includes the parameter)
f1 <- function(x,k=2) return(k*exp(k*x))

# Solution of exp(2x)-2=0 is log(2)/2
x <- roots_newton(f0,f1,k=2)
print(log(2)/2)
```

roots_secant *Secant method for roots*

Description

Find the zero of a function of one variable, $f(x)$, given a starting value close to the zero, using the secant method.

Usage

```
roots_secant(
  fn,
  x0,
  x1,
  imax = 1e+06,
  ftol = 1e-09,
  message = TRUE,
  logg = FALSE,
  ...
)
```

Arguments

fn	A function of one variable. If the function includes variables, these will have to be passed as additional variables, using the same names in the function's definition (see examples).
x0, x1	Two numeric variables. The initial guesses starting the algorithm.
imax	A positive integer. The maximum number of iterations of the algorithm. The default value is 1e6, although convergence is normally obtained with a number of iterations much smaller than imax. imax is important to stop search in those cases in which the algorithm gets stuck in endless loops (non-convergence).
ftol	A real small number. The algorithm stops when $\|f(x)\| < ftol$. Default value is 1e-09.
message	A logical variable to state whether messages about the root and the error have to be printed. The default is for the messages to be printed (message=TRUE).
logg	A logical variable to state whether information on the series of approximating roots is printed (TRUE) or not (FALSE). Default is for such information not to be printed (FALSE).
...	Parameters passed to function 'fn', if needed.

Details

Finding the zero of $f(x)$ is equivalent to finding the roots of the equation:

$$f(x) = 0$$

The algorithm used is essentially a reworking of Newton-Raphson, where the first derivative is replaced by a finite difference computed with values x0 and x1. Thus two values, x0 and x1, needs to be selected to start the procedure. The convergence for this method is in general achieved faster than with the bisection method and slightly less fast than with Newton-Raphson. The algorithm can fail to converge when the secant in one of the iterations is parallel to the x axis.

Value

A numeric value, the zero of the function (or, equivalently, the root of the equation $f(x) = 0$).

Examples

```
# The quadratic equation x^2-5*x+6=0 has two roots, 2 and 3
fn <- function(x) return(x^2-5*x+6)

# Find root 2, starting from two points at the left of 2
x0 <- 0
x1 <- 1
x <- roots_secant(fn,x0,x1)
print(x)

# Find root 3 (no message printing)
x0 <- 5
x1 <- 4
x <- roots_secant(fn,x0,x1,message=FALSE)
print(x)

# Function with a parameter f(x) = exp(kx) - 2
fn <- function(x,k=2) return(exp(k*x)-2)

# Solution of exp(2x)-2=0 is log(2)/2
x0 <- 0
x1 <- 1
x <- roots_secant(fn,x0,x1,k=2)
print(log(2)/2)
```

---

solve_tridiag *Tridiagonal linear system*

Description

Solution of a system of n equations in n unknowns, where the coefficients form a tridiagonal matrix.

Usage

```
solve_tridiag(M)
```

Arguments

M	The $n \times (n+1)$ augmented matrix of coefficients corresponding to the system of n linear equations in n unknowns, $Ax = b$.

Details

The linear system to solve is $Ax = b$, where A is the $n \times n$ matrix of coefficients of the n unknowns in the $n \times 1$ vector x, and b is the $n \times 1$ vector of known numbers. Matrix A is a tridiagonal matrix. This means that A is a sparse matrix with non-zero elements on the main diagonal and the two diagonals adjacent to the main diagonal. The special form of the matrix of coefficients makes it possible to solve the related system using a fast algorithm, here the Thomas algorithm.

Value

A vector of length n containing the n numeric solutions for the n unknowns. If the system has no solutions or an infinite number of solutions, the function returns NULL and dumps a warning message.

Examples

```
# System of four equations in four unknowns
#
# 2x_1 +  x_2               = 1
# 2x_1 + 3x_2 +  x_3        = 2
#         x_2 + 4x_3 + 2x_4 = 3
#                x_3 + 3x_4 = 4

# Augmented matrix M=(A|b)
M <- matrix(c(2,2,0,0,1,3,1,0,0,1,4,1,0,0,2,3,1,2,3,4),
            ncol=5)

# Solution via Thomas algorithm
x <- solve_tridiag(M)
print(x)
```

---

solveLS *Multilinear Least Squares*

Description

Find the parameters, $a_1, \ldots, a_{m+1}$, of the linear model with m parameters, using the least squares technique on a group of n data points in the m dimensional Cartesian space.

Usage

```
solveLS(x, intercept = TRUE, tol = NULL)
```

Arguments

x	A $n \times (m+1)$ matrix or data frame where the first m elements of each row contain the coordinates of a data point, and the last element contain the value corresponding to the linear model.
intercept	A logical variable. It indicates whether to omit or keep the constant a_{m+1} in the model. The default is intercept=TRUE as removing the constant is not advisable, unless there is an absolute certainty (for example in mechanistic models) that it has to be removed.
tol	A real number. The solution of a linear system can be compromised when the condition number of the matrix of coefficients is particularly high (ill-conditioned matrices). tol is the reciprocal of the condition number. For values of tol smaller than 1e-17, ill-conditioning is deemed to be sever enough not to guarantee an accurate solution. For such values the function stops execution, returning an error message. In fact, the solution can still be accurate, notwithstanding ill-conditioning, and the user can force the calculation of a solution using a value of tol smaller than 1e-17. Default is NULL, corresponding to a tol=1e-17.

Details

The linear model with m parameters has the following analytic form:

$$y = a_1 x_1 + a_2 x_2 + \cdots + a_m x_m + a_{m+1}$$

The n data points are contained in a matrix or data frame with $m+1$ columns and n rows. The first m elements of each row contain the coordinates of a data point; the last element contains the corresponding value of the linear fitting, y_i. The least squares procedure is carried out as solution of a matrix equation, via the solve function.

Value

A vector of length m containing the m numeric values of the estimated linear model parameters. The function also prints out the numerical value of the sum of squared residuals. If more than one solution is possible (infinite-solutions case) the function returns a NULL and prints out a related message.

Examples

```
# 5 points exactly on y = 2x_1 - x_2 + 3
p1 <- c(0,1,2)
p2 <- c(1,0,5)
p3 <- c(1,1,4)
p4 <- c(0,2,1)
p5 <- c(2,0,7)

# Assemble points in a single matrix for input
x <- matrix(c(p1,p2,p3,p4,p5),ncol=3,byrow=TRUE)

# Find the least squares estimate of a_1,a_2,a_3
a <- solveLS(x)
print(a)
```

`transform_upper` *Transform to upper triangular*

Description

Transform an $n \times n$ matrix to upper-triangular form, using a series of row operations.

Usage

```
transform_upper(M)
```

Arguments

`M`	An $n \times n$ or $n \times (n+1)$ matrix.

Details

The algorithm used for the transformation is Gauss elimination, which makes use of row operations. If the input matrix has $n + 1$ columns, the transformed $n \times (n + 1)$ matrix can be used to find the solution of the associated system of linear equations.

Value

The transformed $n \times n$ or $n \times (n + 1)$ matrix.

Examples

```
# 3X3 matrix
#
# [ 3  1  1
#   1 -1  2
#  -1  1  1]

# Input matrix
A <- matrix(c(3,1,2,1,1,-1,1,1,-1),ncol=3)

# Upper-triangular matrix
U <- transform_upper(A)
print(U)
```

`which_poly` *Find optimal polynomial model*

Description

`which_poly` tries polynomial regression with polynomials from degree 0 (a constant) to degree 6, on data provided. It then outputs values of the variance of the residuals for each degree and displays a plot of the same versus the degree number, in an effort to suggest the degree of the best polynomial for the regression. The regression coefficients can then be calculated with the function `polysolveLS`.

Usage

```
which_poly(pts, mmax = 6, plt = TRUE, tol = NULL)
```

Arguments

`pts`	A $n \times 2$ matrix or data frame where each row contains the coordinates of a data point used for regression.
`mmax`	An integer. The highest degree of the polynomial to be used to calculate the variance of the residuals. The default value is 6.
`plt`	A logical variable to command the display of the plot of the variance vs the polynomials' degree. The default is `plt=TRUE`.
`tol`	A real number. The solution of a linear system can be compromised when the condition number of the matrix of coefficients is particularly high (ill-conditioned matrices). `tol` is the reciprocal of the condition number. For values of tol smaller than 1e-17, ill-conditioning is deemed to be sever enough not to guarantee an accurate solution. For such values the function stops execution, returning an error message. In fact, the solution can still be accurate, notwithstanding ill-conditioning, and the user can force the calculation of a solution using a value of `tol` smaller than 1e-17. Default is `NULL`, corresponding to a `tol=1e-17`.

Details

The ability of a polynomial regression to account for most data variability, without including data noise is reflected in how the variance,

$$\sigma_e^2 = (\sum_{i=1}^{n} \epsilon_i^2)/(n - m - 1)$$

drops with the increasing degree of the polynomial used to perform the regression. A sudden drop, followed by values slowly decreasing, or alternating slightly increasing and decreasing behaviour, indicates that the degree corresponding to the sudden drop belongs to the polynomial modelling most data variability and neglecting data noise. As polynomial regression is normally used with polynomials of degree up to 4 or 5, a default set of polynomials up to degree 6 is here tried out. Degrees higher than 6 can be forced by the user, but the risk with higher degrees is that the system of normal equations connected with regression becomes severly ill conditioned. In such situations the user should change the tolerance (`tol`) to values smaller than the default `1e-17`.

Value

A data frame with two columns, the first named `m` and including the degrees of all polynomials tested. The second called `sige` and including the value of the variances corresponding to all values of `m`. The function also displays a plot of `sige` vs `m`, by default.

Examples

```
# 21 points close to the quadratic x^2 - 5*x + 6
x <- seq(-2,5,length=21)
set.seed(7766)
eps <- rnorm(21,mean=0,sd=0.5)
y <- x^2-5*x+6+eps

# Data frame
pts <- data.frame(x=x,y=y)

# Try function without plot
ddd <- which_poly(pts,plt=FALSE)
print(ddd)

# Try function with plot and extending
# highest polynomials' degree to 10
ddd <- which_poly(pts,mmax=10)
```

Index