Package: pnd (via r-universe)

November 23, 2026

Type Package

Title Parallel Numerical Derivatives, Gradients, Jacobians, and Hessians of Arbitrary Accuracy Order

Version 0.2.0

Maintainer Andreï Victorovitch Kostyrka <andrei.kostyrka@gmail.com>

Description Numerical derivatives through finite-difference approximations can be calculated using the 'pnd' package with parallel capabilities and optimal step-size selection to improve accuracy. These functions facilitate efficient computation of derivatives, gradients, Jacobians, and Hessians, allowing for more evaluations to reduce the mathematical and machine errors. Designed for compatibility with the 'numDeriv' package, which has not received updates in several years, it introduces advanced features such as computing derivatives of arbitrary order, improving the accuracy of Hessian approximations by avoiding repeated differencing, and parallelising slow functions on Windows, Mac, and Linux.

License EUPL

Encoding UTF-8

URL https://github.com/Fifis/pnd

BugReports https://github.com/Fifis/pnd/issues

Depends R (>= 3.4.0)

Imports parallel, Rdpack

Suggests numDeriv, knitr, rmarkdown, testthat (>= 3.0.0)

RdMacros Rdpack

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Andreï Victorovitch Kostyrka [aut, cre]

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

Date/Publication 2026-11-23 14:59:52 UTC

RemoteUrl https://github.com/cran/pnd

RemoteRef HEAD

RemoteSha 708927f293bef592605bee942663c0f42289fd4b

Contents

alignStrings . . . 2
checkCores . . . 3
checkDimensions . . . 4
dupRowInds . . . 6
fdCoef . . . 7
formatMat . . . 9
GenD . . . 10
generateGrid . . . 14
generateGrid2 . . . 14
Grad . . . 15
gradstep . . . 19
Jacobian . . . 21
plot.stepsize . . . 24
print.Hessian . . . 25
printMat . . . 27
runParallel . . . 28
solveVandermonde . . . 30
step.CR . . . 31
step.DV . . . 34
step.K . . . 36
step.M . . . 39
step.plugin . . . 42
step.SW . . . 44
stepx . . . 46
stopParallel . . . 47

Index **48**

`alignStrings` *Align printed output to the longest argument*

Description

Align printed output to the longest argument

Usage

```
alignStrings(x, names = NULL, pad = c("l", "c", "r"))
```

Arguments

x	A numeric vector or matrix to be aligned with a vector of column names.
names	Optional: a character vector of element or column names to be output first. Overrides existing names. Numeric inputs are converted to character automatically.
pad	A single character: "l" for left padding (flush-right justification), "c" for centre, and "r" for right padding (flush-left justification).

Value

A character matrix with the first row of names and the rest aligned content

Examples

```
x <- structure(1:4, names = month.name[1:4])
print(alignStrings(x, names(x)), quote = FALSE)
print(alignStrings(x, names(x), pad = "c"), quote = FALSE)  # Centring
print(alignStrings(x, names(x), pad = "r"), quote = FALSE)  # Left alignment

x <- matrix(c(1, 2.3, 4.567, 8, 9, 0), nrow = 2, byrow = TRUE)
colnames(x) <- c("Andy", "Bradley", "Ci")
alignStrings(x, pad = "c")
```

---

checkCores *Number of core checks and changes*

Description

Number of core checks and changes

Usage

```
checkCores(cores = NULL)
```

Arguments

cores	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.

Value

An integer with the number of cores.

Examples

```
checkCores()
checkCores(2)
suppressWarnings(checkCores(1000))
```

checkDimensions *Determine function dimensionality and vectorisation*

Description

Determine function dimensionality and vectorisation

Usage

```
checkDimensions(
  FUN,
  x,
  f0 = NULL,
  func = NULL,
  elementwise = NA,
  vectorised = NA,
  multivalued = NA,
  deriv.order = 1,
  acc.order = 2,
  side = 0,
  h = NULL,
  zero.tol = NULL,
  cores = 1,
  preschedule = TRUE,
  cl = NULL,
  ...
)

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

Arguments

FUN	A function returning a numeric scalar or a vector whose derivatives are to be computed.
x	Numeric vector or scalar: the point(s) at which the derivative is estimated. Function values must be finite.
f0	Optional numeric: if provided, used to determine the vectorisation type to save time.
func	For compatibility with numDeriv::grad() only. If instead of FUN, func is used, it will be reassigned to FUN with a warning.
elementwise	Logical: is the domain effectively 1D, i.e. is this a mapping $\mathbb{R} \mapsto \mathbb{R}$ or $\mathbb{R}^n \mapsto \mathbb{R}^n$. If NA, compares the output length to the input length.
vectorised	Logical: if TRUE, the function is assumed to be vectorised: it will accept a vector of parameters and return a vector of values of the same length. Use FALSE for

functions that take vector arguments and return outputs of arbitrary length (for $\mathbb{R}^n \mapsto \mathbb{R}^k$ functions). If NA, checks the output length and assumes vectorisation if it matches the input length; this check is necessary and potentially slow.

`multivalued` Logical: if TRUE, the function is assumed to return vectors longer than 1. Use FALSE for element-wise functions. If NA, attempts inferring it from the function output.

`deriv.order` Integer or vector of integers indicating the desired derivative order, $\mathrm{d}^d/\mathrm{d}x^d$, for each element of x.

`acc.order` Integer or vector of integers specifying the desired accuracy order for each element of x. The truncation error will be of the order $O(h^{\mathrm{acc.order}})$.

`side` Integer scalar or vector indicating the type of finite difference: 0 for central, 1 for forward, and -1 for backward differences. Central differences are recommended unless computational cost is prohibitive.

`h` Numeric scalar used for test stencil construction; NULL or character uses the median of `stepx()`.

`zero.tol` Small positive integer: if abs(x) >= zero.tol, then, the automatically guessed step size is relative or near-relative (x multiplied by the step), unless an auto-selection procedure is requested; otherwise, it is absolute.

`cores` Number of cores for coordinate-wise detection evaluations.

`preschedule` Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.

`cl` An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.

`...` Additional arguments passed to FUN. Important! Since R does partial matching for argument names, using arguments like f(x, a) and passing a = 1 to `GenD()`, `Grad()`, or `Jacobian()` will result in the interpretation acc.order = 1 because the latter is a named argument of these functions. Either supply the full names of all similar-looking arguments or name arguments differently (e.g. a0 instead of a).

Details

The following combinations of parameters are allowed, suggesting specific input and output handling by other functions:

	elementwise	!elementwise
!multivalued, vectorised	FUN(xgrid)	*(undefined)*
!multivalued, !vectorised	[mc]lapply(xgrid, FUN)	[mc]lapply gradient
multivalued, vectorised	*(undefined)*	FUN(xgrid) Jacobian
multivalued, !vectorised	*(undefined)*	[mc]lapply Jacobian

Some combinations are impossible: multi-valued functions cannot be element-wise, and single-valued vectorised functions must be element-wise.

In brief, testing the input and output length and vectorisation capabilities may result in five cases, unlike 3 in `numDeriv::grad()` that does not provide checks for Jacobians.

Value

A named logical vector indicating if a function is element-wise or not, vectorised or not, and multivalued or not. Its `seconds` attribute is `NA` when `f0` is supplied or `options(pnd.timing = FALSE)`.

Examples

```
checkDimensions(sin, x = 1:4, h = 1e-5)  # Rn -> Rn vectorised
checkDimensions(function(x) integrate(sin, 0, x)$value, x = 1:4, h = 1e-5)  # non vec
checkDimensions(function(x) sum(sin(x)), x = 1:4, h = 1e-5)  # Rn -> R, gradient
checkDimensions(function(x) c(sin(x), cos(x)), x = 1, h = 1e-5)  # R -> Rn, Jacobian
checkDimensions(function(x) c(sin(x), cos(x)), x = 1:4, h = 1e-5)  # vec Jac
checkDimensions(function(x) c(integrate(sin, 0, x)$value, integrate(sin, -x, 0)$value),
                x = 1:4, h = 1e-5)  # non-vectorised Jacobian
```

---

dupRowInds *Repeated indices of the first unique value*

Description

Repeated indices of the first unique value

Usage

```
dupRowInds(m)
```

Arguments

`m`	A matrix or a data frame. This function is an inverse function to such operations as `m[c(1:3, 1, 1, 2), ]`: the matrix with potentially duplicated rows is taken as input, and indices of each row in `unique(m)` are returned. This function is faster – at least in the examples tested so far – than `match(data.frame(t(m)), data.frame(t(unique(m))))`.

Value

A vector of row indices in `unique(m)`.

Examples

```
dupRowInds(mtcars[rep(1:10, 10), rep(1:10, 10)])
dupRowInds(matrix(rnorm(1000), ncol = 10))
```

`fdCoef` *Finite-difference coefficients for arbitrary grids*

Description

This function computes the coefficients for a numerical approximation to derivatives of any specified order. It provides the minimally sufficient stencil for the chosen derivative order and desired accuracy order. It can also use any user-supplied stencil (uniform or non-uniform). For that stencil $\{b_i\}_{i=1}^{n}$, it computes the optimal weights $\{w_i\}$ that yield the numerical approximation of the derivative:

$$\frac{d^d f}{dx^d} \approx h^{-d} \sum_{i=1}^{n} w_i f(x + b_i \cdot h)$$

Usage

```
fdCoef(
  deriv.order = 1L,
  side = c(0L, 1L, -1L),
  acc.order = 2L,
  stencil = NULL,
  zero.action = c("drop", "round", "none"),
  zero.tol = NULL
)
```

Arguments

`deriv.order`	Order of the derivative (d in $\frac{d^d f}{dx^d}$) for which a numerical approximation is needed.
`side`	Integer that determines the type of finite-difference scheme: `0` for central (AKA symmetrical or two-sided; the default), `1` for forward, and `-1` for backward. Using `2` (for 'two-sided') triggers a warning and is treated as `0`. Unless the function is prohibitively expensive, central differences are strongly recommended for their accuracy.
`acc.order`	Order of accuracy: defines how the approximation error scales with the step size h, specifically $O(h^a)$, where a is the accuracy order and depends on the higher-order derivatives of the function.
`stencil`	Optional custom vector of points for function evaluation. Must include at least `d+1` points for the `d`-th order derivative.
`zero.action`	Character string specifying how to handle near-zero weights: `"drop"` to omit small (less in absolute value than `zero.tol`) weights and corresponding stencil points, `"round"` to round small weights to zero, and `"none"` to leave all weights as calculated. E.g. the stencil for $f'(x)$ is `(-1, 0, 1)` with weights `(-0.5, 0, 0.5)`; using `"drop"` eliminates the zero weight, and the redundant `f(x)` is not computed.
`zero.tol`	Non-negative scalar defining the threshold: weights smaller in absolute value than `zero.tol` are considered near-zero.

Details

This function relies on the approach of approximating numerical derivatives by weighted sums of function values described in (Fornberg 1988). It reproduces all tables from this paper exactly; see the example below to create Table 1.

The finite-difference coefficients for any given stencil are given as a solution of a linear system. The capabilities of this function are similar to those of (Taylor 2016), but instead of matrix inversion, the (Björck and Pereyra 1970) algorithm is used because the left-hand-side matrix is a Vandermonde matrix, and its inverse may be very inaccurate, especially for long one-sided stencils.

The weights computed for the stencil via this algorithm are very reliable; numerical simulations in (Higham 1987) show that the relative error is low even for ill-conditioned systems. (Kostyrka 2025) computes the exact relative error of the weights on the stencils returned by this function; the zero tolerance is based on these calculations.

Value

A list containing the `stencil` used and the corresponding `weights` for each point.

References

Björck Å, Pereyra V (1970). "Solution of Vandermonde systems of equations." *Mathematics of computation*, **24**(112), 893–903.

Fornberg B (1988). "Generation of Finite Difference Formulas on Arbitrarily Spaced Grids." *Mathematics of Computation*, **51**(184), 699–706. doi:10.1090/S0025571819880935077O.

Higham NJ (1987). "Error analysis of the Björck-Pereyra algorithms for solving Vandermonde systems." *Numerische Mathematik*, **50**(5), 613–632.

Kostyrka AV (2025). "What are you doing, step size: fast computation of accurate numerical derivatives with finite precision." Working paper.

Taylor CR (2016). "Finite Difference Coefficients Calculator." https://web.archive.org/web/20261111034753/https://web.media.mit.edu/~crtaylor/calculator.html.

Examples

```
fdCoef()  # Simple two-sided derivative
fdCoef(2) # Simple two-sided second derivative
fdCoef(acc.order = 4)$weights * 12  # Should be (1, -8, 8, -1)

# Using a custom stencil for the first derivative: x-2h and x+h
fdCoef(stencil = c(-2, 1), acc.order = 1)

# Reproducing Table 1 from Fornberg (1988) (cited above)
pad9 <- function(x) {l <- length(x); c(a <- rep(0, (9-l)/2), x, a)}
f <- function(d, a) pad9(fdCoef(deriv.order = d, acc.order = a,
                                zero.action = "round")$weights)
t11 <- t(sapply((1:4)*2, function(a) f(d = 1, a)))
t12 <- t(sapply((1:4)*2, function(a) f(d = 2, a)))
```

```
t13 <- t(sapply((1:3)*2, function(a) f(d = 3, a)))
t14 <- t(sapply((1:3)*2, function(a) f(d = 4, a)))
t11 <- cbind(t11[, 1:4], 0, t11[, 5:8])
t13 <- cbind(t13[, 1:4], 0, t13[, 5:8])
t1 <- data.frame(OrdDer = rep(1:4, times = c(4, 4, 3, 3)),
                 OrdAcc = c((1:4)*2, (1:4)*2, (1:3)*2, (1:3)*2),
                 rbind(t11, t12, t13, t14))
colnames(t1)[3:11] <- as.character(-4:4)
print(t1, digits = 4)
```

`formatMat` *Round a matrix to N significant digits in mixed FP/exp notation*

Description

Round a matrix to N significant digits in mixed FP/exp notation

Usage

```
formatMat(x, digits = 3, shave.spaces = TRUE)
```

Arguments

`x`	Numeric matrix.
`digits`	Positive integer: the number of digits after the decimal point in scientific notation (i.e. one less than the number of significant digits).
`shave.spaces`	Logical: if true, removes spaces to ensure compact output; if false, results in nearly fixed-width output (almost).

Value

A character matrix of formatted entries.

Examples

```
x <- matrix(c(1234567, 12345.67, 123.4567,
              1.23456, -1.23456e-1, 0,
              -1.23456e-4, 1.23456e-2, -1.23456e-6), nrow = 3)
print(formatMat(x), quote = FALSE)
print(formatMat(x, digits = 1), quote = FALSE)
```

GenD *Numerical derivative matrices with parallel capabilities*

Description

Computes numerical derivatives of a scalar or vector function using finite-difference methods. This function serves as a backbone for `Grad()` and `Jacobian()`, allowing for detailed control over the derivative computation process, including order of derivatives, accuracy, and step size. `GenD` is fully vectorised over different coordinates of the function argument, allowing arbitrary accuracies, sides, and derivative orders for different coordinates.

Usage

```
GenD(
  FUN,
  x,
  elementwise = NA,
  vectorised = NA,
  multivalued = NA,
  deriv.order = 1L,
  side = 0,
  acc.order = 2L,
  stencil = NULL,
  h = NULL,
  zero.tol = NULL,
  h0 = NULL,
  control = list(),
  f0 = NULL,
  cores = 1,
  preschedule = TRUE,
  cl = NULL,
  func = NULL,
  method = NULL,
  method.args = list(),
  ...
)

## S3 method for class 'GenD'
print(
  x,
  digits = 4,
  shave.spaces = TRUE,
  begin = "",
  sep = "  ",
  end = "",
  ...
)
```

Arguments

FUN	A function returning a numeric scalar or a vector whose derivatives are to be computed.
x	Numeric vector or scalar: the point(s) at which the derivative is estimated. Function values must be finite.
elementwise	Logical: is the domain effectively 1D, i.e. is this a mapping $\mathbb{R} \mapsto \mathbb{R}$ or $\mathbb{R}^n \mapsto \mathbb{R}^n$. If NA, compares the output length to the input length.
vectorised	Logical: if TRUE, the function is assumed to be vectorised: it will accept a vector of parameters and return a vector of values of the same length. Use FALSE for functions that take vector arguments and return outputs of arbitrary length (for $\mathbb{R}^n \mapsto \mathbb{R}^k$ functions). If NA, checks the output length and assumes vectorisation if it matches the input length; this check is necessary and potentially slow.
multivalued	Logical: if TRUE, the function is assumed to return vectors longer than 1. Use FALSE for element-wise functions. If NA, attempts inferring it from the function output.
deriv.order	Integer or vector of integers indicating the desired derivative order, $\mathrm{d}^d/\mathrm{d}x^d$, for each element of x.
side	Integer scalar or vector indicating the type of finite difference: 0 for central, 1 for forward, and -1 for backward differences. Central differences are recommended unless computational cost is prohibitive.
acc.order	Integer or vector of integers specifying the desired accuracy order for each element of x. The truncation error will be of the order $O(h^{\mathrm{acc.order}})$.
stencil	Optional custom vector of points for function evaluation. Must include at least d+1 points for the d-th order derivative.
h	Numeric or character specifying the step size(s) for the numerical difference or a method of automatic step determination ("CR", "CRm", "DV", or "SW" to be used in gradstep()). The default value is described in ?GenD.
zero.tol	Small positive integer: if abs(x) >= zero.tol, then, the automatically guessed step size is relative or near-relative (x multiplied by the step), unless an auto-selection procedure is requested; otherwise, it is absolute.
h0	Numeric scalar or vector: initial step size for automatic search with gradstep().
control	A named list of tuning parameters passed to gradstep().
f0	Optional numeric: if provided, used to determine the vectorisation type to save time.
cores	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.
preschedule	Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.
cl	An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.

func	For compatibility with numDeriv::grad() only. If instead of FUN, func is used, it will be reassigned to FUN with a warning.
method	For compatibility with numDeriv::grad() only. Supported values: "simple" and "Richardson". Non-null values result in a warning.
method.args	For compatibility with numDeriv::grad() only. Check ?numDeriv::grad for a list of values. Recognised arguments result in a warning.
...	Additional arguments passed to FUN. Important! Since R does partial matching for argument names, using arguments like f(x, a) and passing a = 1 to GenD(), Grad(), or Jacobian() will result in the interpretation acc.order = 1 because the latter is a named argument of these functions. Either supply the full names of all similar-looking arguments or name arguments differently (e.g. a0 instead of a).
digits	Positive integer: the number of digits after the decimal point in scientific notation (i.e. one less than the number of significant digits).
shave.spaces	Logical: if true, removes spaces to ensure compact output; if false, results in nearly fixed-width output (almost).
begin	A character to put at the beginning of each line, usually "", "(", or "c(" (the latter is useful if console output is used in calculations).
sep	The column delimiter, usually " ", "|", "&" (for LaTeX), or ", ".
end	A character to put at the end of each line, usually "" or ")".

Details

For computing gradients and Jacobians, use convenience wrappers Jacobian and Grad.

If the step size is too large, the slope of the secant poorly estimates the derivative; if it is too small, it leads to numerical instability due to the function value rounding.

The optimal step size for one-sided differences typically approaches Mach.eps^(1/2) to balance the Taylor series truncation error with the rounding error due to storing function values with limited precision. For two-sided differences, it is proportional to Mach.eps^(1/3). However, selecting the best step size typically requires knowledge of higher-order derivatives, which may not be readily available. Luckily, using h = "SW" invokes a reliable automatic data-driven step-size selection. Other options include "DV", "CR", and "CRm". With default zero.tol, the step size is .Machine$double.eps^(1/(deriv.order + acc.order - 0.5)) for abs(x) < zero.tol, (2.2e-16)^(1/(deriv.order + acc.order)) * abs(x) for abs(x) > 1, and interpolates exponentially between these values for zero.tol <= abs(x) <= 1.

The use of f0 can reduce computation time similar to the use of f.lower and f.upper in uniroot().

Unlike numDeriv::grad() and numDeriv::jacobian(), this function fully preserves the names of x and FUN(x).

Additional arguments to the print method (...) are ignored.

Value

A vector or matrix containing the computed derivatives, structured according to the dimensionality of x and FUN. If the function is classified as single-valued, returns a gradient vector; otherwise, returns a Jacobian matrix. With options(pnd.timing = TRUE) (the default), timing records

elapsed seconds for preparation, the evaluation phase, and the complete call. Preparation includes checks, step selection, stencils and derivative assembly. Evaluation includes batching, worker startup, scheduling and communication. `function.calls` counts calls in the evaluation phase, including failed batch attempts. Dividing evaluation time by this count gives elapsed time per call, including overhead. Timing uses `proc.time()`; resolution depends on the platform. Set `options(pnd.timing = FALSE)` to omit timing and these attributes.

See Also

`gradstep()` for automatic step-size selection; `runParallel()` for parallel backends and cleanup.

Examples

```
# Case 1: Vector argument, vector output
f1 <- sin
g1 <- GenD(FUN = f1, x = 1:100)
g1.true <- cos(1:100)
plot(1:100, g1 - g1.true, main = "Approximation error of d/dx sin(x)")

# Case 2: Vector argument, scalar result
f2 <- function(x) sum(sin(x))
g2    <- GenD(FUN = f2, x = 1:4)
g2.h2 <- Grad(FUN = f2, x = 1:4, h = 7e-6)
g2 - g2.h2  # Tiny differences due to different step sizes
g2.auto <- Grad(FUN = f2, x = 1:4, h = "SW")
print(attr(g2.auto, "step.search")$exitcode)  # Success

# Case 3: vector input, vector output of different length
f3 <- function(x) c(sum(sin(x)), prod(cos(x)))
x3 <- 1:3
j3 <- GenD(f3, x3, multivalued = TRUE)
print(j3)

# Gradients for vectorised functions -- e.g. leaky ReLU
LReLU <- function(x) ifelse(x > 0, x, 0.01*x)
system.time(replicate(10, suppressMessages(GenD(LReLU, runif(30, -1, 1)))))
system.time(replicate(10, suppressMessages(GenD(LReLU, runif(30, -1, 1)))))

# Saving time for slow functions by using pre-computed values
x <- 1:4
finner <- function(x) sin(x*log(abs(x)+1))
fouter <- function(x) integrate(finner, 0, x, rel.tol = 1e-12, abs.tol = 0)$value
# The outer function is non-vectorised
fslow <- function(x) {Sys.sleep(0.01); sum(sapply(x, fouter))}
f0 <- sum(sapply(x, fouter))
system.time(GenD(fslow, x, side = 1, acc.order = 2, f0 = f0))
# Now, with extra checks, it will be slower
system.time(GenD(fslow, x, side = 1, acc.order = 2))
# Printing whilst preserving names
x <- structure(1:3, names = c("Andy", "Bradley", "Ca"))
print(Grad(function(x) prod(sin(x)), 1))  # 1D derivative
print(Grad(function(x) prod(sin(x)), x, h = "CR"))
```

```
print(Jacobian(function(x) c(prod(sin(x)), sum(exp(x))), x))
```

`generateGrid` *Create a grid of points for a gradient / Jacobian*

Description

Create a grid of points for a gradient / Jacobian

Usage

```
generateGrid(x, h, stencils, elementwise, vectorised)
```

Arguments

`x`	Numeric vector or scalar: the point(s) at which the derivative is estimated. Function values must be finite.
`h`	Numeric scalar or vector of already-selected absolute step sizes.
`stencils`	A list of outputs from `fdCoef()` for each coordinate of `x`.
`elementwise, vectorised`	Logical flags from `checkDimensions()`, not `NA`.

Value

A list with points for evaluation, summation weights for derivative computation, and indices for combining values.

Examples

```
generateGrid(1:4, h = 1e-5, elementwise = TRUE, vectorised = TRUE,
             stencils = lapply(1:4, function(a) fdCoef(acc.order = a)))
```

`generateGrid2` *Generate grid points for Hessians*

Description

Creates a list of evaluation points for second derivatives: both diagonal ($\partial^2/\partial x_i^2$) and cross ($\partial^2/\partial x_i \partial x_j$).

Usage

```
generateGrid2(x, side, acc.order, h)
```

Arguments

x	Numeric vector or scalar: the point(s) at which the derivative is estimated. Function values must be finite.
side	Integer vector of difference directions, of length length(x).
acc.order	Integer vector of accuracy orders, of length length(x).
h	Numeric vector of absolute step sizes, of length length(x).

Value

A list with elements:

- xlist: a list of evaluation points,
- weights: the finite-difference weights (one per point),
- i1, i2: integer vectors giving partial-derivative indices.

The length of each vector matches xlist.

See Also

GenD(), Hessian().

Examples

```
generateGrid2(1:4, side = rep(0, 4), acc.order = c(2, 6, 4, 2),
              h = c(1e-5, 1e-4, 2e-5, 1e-6))
```

Grad *Gradient computation with parallel capabilities*

Description

Computes numerical derivatives and gradients of scalar-valued functions using finite differences. This function supports both two-sided (central, symmetric) and one-sided (forward or backward) derivatives. It can utilise parallel processing to accelerate computation of gradients for slow functions or to attain higher accuracy faster.

Usage

```
Grad(
  FUN,
  x,
  elementwise = NA,
  vectorised = NA,
  multivalued = NA,
  deriv.order = 1L,
  side = 0,
```

```
  acc.order = 2,
  stencil = NULL,
  h = NULL,
  zero.tol = NULL,
  h0 = NULL,
  control = list(),
  f0 = NULL,
  cores = 1,
  preschedule = TRUE,
  cl = NULL,
  func = NULL,
  method = NULL,
  method.args = list(),
  ...
)
```

Arguments

FUN	A function returning a numeric scalar or a vector whose derivatives are to be computed.
x	Numeric vector or scalar: the point(s) at which the derivative is estimated. Function values must be finite.
elementwise	Logical: is the domain effectively 1D, i.e. is this a mapping $\mathbb{R} \mapsto \mathbb{R}$ or $\mathbb{R}^n \mapsto \mathbb{R}^n$. If NA, compares the output length to the input length.
vectorised	Logical: if TRUE, the function is assumed to be vectorised: it will accept a vector of parameters and return a vector of values of the same length. Use FALSE for functions that take vector arguments and return outputs of arbitrary length (for $\mathbb{R}^n \mapsto \mathbb{R}^k$ functions). If NA, checks the output length and assumes vectorisation if it matches the input length; this check is necessary and potentially slow.
multivalued	Logical: if TRUE, the function is assumed to return vectors longer than 1. Use FALSE for element-wise functions. If NA, attempts inferring it from the function output.
deriv.order	Integer or vector of integers indicating the desired derivative order, $\mathrm{d}^d/\mathrm{d}x^d$, for each element of x.
side	Integer scalar or vector indicating the type of finite difference: 0 for central, 1 for forward, and -1 for backward differences. Central differences are recommended unless computational cost is prohibitive.
acc.order	Integer or vector of integers specifying the desired accuracy order for each element of x. The truncation error will be of the order $O(h^{\mathrm{acc.order}})$.
stencil	Optional custom vector of points for function evaluation. Must include at least d+1 points for the d-th order derivative.
h	Numeric or character specifying the step size(s) for the numerical difference or a method of automatic step determination ("CR", "CRm", "DV", or "SW" to be used in gradstep()). The default value is described in ?GenD.
zero.tol	Small positive integer: if abs(x) >= zero.tol, then, the automatically guessed step size is relative or near-relative (x multiplied by the step), unless an auto-selection procedure is requested; otherwise, it is absolute.

h0	Numeric scalar or vector: initial step size for automatic search with gradstep().
control	A named list of tuning parameters passed to gradstep().
f0	Optional numeric: if provided, used to determine the vectorisation type to save time.
cores	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.
preschedule	Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.
cl	An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.
func	For compatibility with numDeriv::grad() only. If instead of FUN, func is used, it will be reassigned to FUN with a warning.
method	For compatibility with numDeriv::grad() only. Supported values: "simple" and "Richardson". Non-null values result in a warning.
method.args	For compatibility with numDeriv::grad() only. Check ?numDeriv::grad for a list of values. Recognised arguments result in a warning.
...	Additional arguments passed to FUN. Important! Since R does partial matching for argument names, using arguments like f(x, a) and passing a = 1 to GenD(), Grad(), or Jacobian() will result in the interpretation acc.order = 1 because the latter is a named argument of these functions. Either supply the full names of all similar-looking arguments or name arguments differently (e.g. a0 instead of a).

Details

This function aims to be 100% compatible with the syntax of numDeriv::grad(), but there might be differences in the step size because some choices made in numDeriv are not consistent with theory.

There is one feature of the default step size in numDeriv that deserves an explanation. In that package (but not in pnd),

- If method = "simple", then, simple forward differences are used with a fixed step size eps, which we denote by ε.
- If method = "Richardson", then, central differences are used with a fixed step $h := |d \cdot x| + \varepsilon(|x| < \text{zero.tol})$, where d = 1e-4 is the relative step size and eps becomes an extra addition to the step size for the arguments that are closer to zero than zero.tol.

We believe that the latter may lead to mistakes when the user believes that they can set the step size for near-zero arguments, whereas in reality, a combination of d and eps is used.

Here is the synopsis of the old arguments:

side numDeriv uses NA for handling two-sided differences. The pnd equivalent is 0, and NA is replaced with 0.

eps If numDeriv method = "simple", then, eps = 1e-4 is the absolute step size and forward differences are used. If method = "Richardson", then, eps = 1e-4 is the absolute increment of the step size for small arguments below the zero tolerance.

d If numDeriv method = "Richardson", then, d*abs(x) is the step size for arguments above the zero tolerance and the baseline step size for small arguments that gets incremented by eps.

r The number of Richardson extrapolations that successively reduce the initial step size. For two-sided differences, each extrapolation increases the accuracy order by 2.

v The reduction factor in Richardson extrapolations.

Here are the differences in the new compatible implementation.

eps If numDeriv method = "simple", then, 1e-5 * .Machine$double.eps^(1/3 - 1/4) is used because one-sided differences require a smaller step size to reduce the truncation error. If method = "Richardson", then, eps = 1e-5.

d If numDeriv method = "Richardson", then, d*abs(x) is the step size for arguments above the zero tolerance and the baseline step size for small arguments that gets incremented by eps.

r Half the accuracy order for central differences, i.e. the number of Richardson extrapolations that successively reduce the initial step size.

v Ignored (in the old implementation, used to be the reduction factor in Richardson extrapolations).

Grad does an initial check and calls GenD() with a set of appropriate parameters (multivalued = FALSE if the check succeeds). In case of parameter mismatch, throws an error.

Value

Numeric vector of the gradient. If FUN is classified as multi-valued, an error is issued suggesting the use of Jacobian(). Timing attributes are described in GenD().

See Also

GenD(), Jacobian(), runParallel()

Examples

```
f <- function(x) sum(sin(x))
g1 <- Grad(FUN = f, x = 1:4)
g2 <- Grad(FUN = f, x = 1:4, h = 7e-6)
g2 - g1  # Tiny differences due to different step sizes
g.auto <- Grad(FUN = f, x = 1:4, h = "SW")
print(g.auto)
attr(g.auto, "step.search")$exitcode  # Success

# Gradients for vectorised functions -- e.g. leaky ReLU
LReLU <- function(x) ifelse(x > 0, x, 0.01*x)
Grad(LReLU, seq(-1, 1, 0.1))
```

gradstep *Automatic step selection for gradients*

Description

Automatic step selection for gradients

Usage

```
gradstep(
  FUN,
  x,
  h0 = NULL,
  method = c("plugin", "SW", "CR", "CRm", "DV", "M", "K"),
  control = NULL,
  cores = 1,
  preschedule = getOption("pnd.preschedule", TRUE),
  cl = NULL,
  ...
)

## S3 method for class 'stepsize'
print(x, ...)

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

Arguments

FUN	Scalar-valued function for which the optimal numerical derivative step size is needed.
x	Numeric vector or scalar: the point at which the derivative is computed and the optimal step size is estimated.
h0	Numeric vector or scalar: initial step size, defaulting to stepx() (rescaled for "M" and "K").
method	Character indicating the method: "CR" for (Curtis and Reid 1974), "CRm" for modified Curtis–Reid, "DV" for (Dumontet and Vignes 1977), "SW" (Stepleman and Winarsky 1979), "M" for (Mathur 2012), "K" for Kostyrka (2026, experimental), and "plugin" for the single-step plug-in estimator.
control	A named list of tuning parameters for the method. If NULL, default values are used. See the documentation for the respective methods. Note that iteration history including function evaluations is returned, but different methods have slightly different diagnostic outputs.
cores	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.

preschedule	Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.
cl	An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.
...	Passed to FUN.

Details

We recommend using the Stepleman–Winarsky algorithm because it does not suffer from overestimation of the truncation error in the Curtis–Reid approach and from sensitivity to near-zero third derivatives in the Dumontet–Vignes approach. It really tries multiple step sizes.

Value

A list similar to the one returned by optim() and made of concatenated elements of coordinate-wise lists: par – the optimal step sizes found, value – the estimated numerical gradient, counts – method-specific counts for each coordinate, abs.error – estimates of the truncation and rounding errors, exitcode – method-specific termination codes; see the respective methods. message – summary messages of the exit status. original is a list of lists including the full step size search path, argument grids, function values on those grids, estimated error ratios, and estimated derivative values for each coordinate.

References

Curtis AR, Reid JK (1974). "The Choice of Step Lengths When Using Differences to Approximate Jacobian Matrices." *IMA Journal of Applied Mathematics*, **13**(1), 121–126. doi:10.1093/imamat/13.1.121.

Dumontet J, Vignes J (1977). "Détermination du pas optimal dans le calcul des dérivées sur ordinateur." *RAIRO. Analyse numérique*, **11**(1), 13–25. doi:10.1051/m2an/1977110100131.

Mathur R (2012). *An Analytical Approach to Computing Step Sizes for Finite-Difference Derivatives*. Ph.D. thesis, University of Texas at Austin. http://hdl.handle.net/2152/ETD-UT-2012-05-5275.

Stepleman RS, Winarsky ND (1979). "Adaptive numerical differentiation." *Mathematics of Computation*, **33**(148), 1257–1264. doi:10.1090/s0025571819790537 9698.

See Also

step.CR() for Curtis–Reid (1974) and its modification, step.plugin() for the one-step plug-in solution, step.DV() for Dumontet–Vignes (1977), step.SW() for Stepleman–Winarsky (1979), step.M() for Mathur (2012), and step.K() for Kostyrka (2026).

Examples

```
gradstep(x = 1, FUN = sin, method = "CR")
gradstep(x = 1, FUN = sin, method = "CRm")
```

```
gradstep(x = 1, FUN = sin, method = "DV")
gradstep(x = 1, FUN = sin, method = "SW")
gradstep(x = 1, FUN = sin, method = "M")
gradstep(x = 1, FUN = sin, method = "K")
# Works for gradients
gradstep(x = 1:4, FUN = function(x) sum(sin(x)), method = "CR")
gradstep(x = 1:4, FUN = function(x) sum(sin(x)), method = "CRm")
gradstep(x = 1:4, FUN = function(x) sum(sin(x)), method = "DV")
gradstep(x = 1:4, FUN = function(x) sum(sin(x)), method = "SW")
gradstep(x = 1:4, FUN = function(x) sum(sin(x)), method = "M")
gradstep(x = 1:4, FUN = function(x) sum(sin(x)), method = "K")
print(step.CR(x = 1, sin))
print(step.DV(x = 1, sin))
print(step.plugin(x = 1, sin))
print(step.SW(x = 1, sin))
print(step.M(x = 1, sin))
print(step.K(x = 1, sin))
f <- function(x) x[1]^3 + sin(x[2])*exp(x[3])
print(gradstep(x = c(2, pi/4, 0.5), f))
```

`Jacobian` *Jacobian matrix computation with parallel capabilities*

Description

Computes the numerical Jacobian for vector-valued functions. Its columns are partial derivatives of the function with respect to the input elements. This function supports both two-sided (central, symmetric) and one-sided (forward or backward) derivatives. It can utilise parallel processing to accelerate computation of gradients for slow functions or to attain higher accuracy faster. Fork-based parallelism uses `parallel::mclapply()` on macOS and Linux; Windows automatically reuses PSOCK workers. See `runParallel()` for backend options and cleanup. Clusters use `parallel::parLapply()` or `parallel::parLapplyLB()` on all platforms.

Usage

```
Jacobian(
  FUN,
  x,
  elementwise = NA,
  vectorised = NA,
  multivalued = NA,
  deriv.order = 1L,
  side = 0,
  acc.order = 2,
  stencil = NULL,
  h = NULL,
  zero.tol = NULL,
  h0 = NULL,
```

```
  control = list(),
  f0 = NULL,
  cores = 1,
  preschedule = TRUE,
  cl = NULL,
  func = NULL,
  method = NULL,
  method.args = list(),
  ...
)
```

Arguments

FUN	A function returning a numeric scalar or a vector whose derivatives are to be computed.
x	Numeric vector or scalar: the point(s) at which the derivative is estimated. Function values must be finite.
elementwise	Logical: is the domain effectively 1D, i.e. is this a mapping $\mathbb{R} \mapsto \mathbb{R}$ or $\mathbb{R}^n \mapsto \mathbb{R}^n$. If NA, compares the output length to the input length.
vectorised	Logical: if TRUE, the function is assumed to be vectorised: it will accept a vector of parameters and return a vector of values of the same length. Use FALSE for functions that take vector arguments and return outputs of arbitrary length (for $\mathbb{R}^n \mapsto \mathbb{R}^k$ functions). If NA, checks the output length and assumes vectorisation if it matches the input length; this check is necessary and potentially slow.
multivalued	Logical: if TRUE, the function is assumed to return vectors longer than 1. Use FALSE for element-wise functions. If NA, attempts inferring it from the function output.
deriv.order	Integer or vector of integers indicating the desired derivative order, $\mathrm{d}^d/\mathrm{d}x^d$, for each element of x.
side	Integer scalar or vector indicating the type of finite difference: 0 for central, 1 for forward, and -1 for backward differences. Central differences are recommended unless computational cost is prohibitive.
acc.order	Integer or vector of integers specifying the desired accuracy order for each element of x. The truncation error will be of the order $O(h^{\mathrm{acc.order}})$.
stencil	Optional custom vector of points for function evaluation. Must include at least d+1 points for the d-th order derivative.
h	Numeric or character specifying the step size(s) for the numerical difference or a method of automatic step determination ("CR", "CRm", "DV", or "SW" to be used in gradstep()). The default value is described in ?GenD.
zero.tol	Small positive integer: if abs(x) >= zero.tol, then, the automatically guessed step size is relative or near-relative (x multiplied by the step), unless an auto-selection procedure is requested; otherwise, it is absolute.
h0	Numeric scalar or vector: initial step size for automatic search with gradstep().
control	A named list of tuning parameters passed to gradstep().

f0	Optional numeric: if provided, used to determine the vectorisation type to save time.
cores	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.
preschedule	Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.
cl	An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.
func	For compatibility with numDeriv::grad() only. If instead of FUN, func is used, it will be reassigned to FUN with a warning.
method	For compatibility with numDeriv::grad() only. Supported values: "simple" and "Richardson". Non-null values result in a warning.
method.args	For compatibility with numDeriv::grad() only. Check ?numDeriv::grad for a list of values. Recognised arguments result in a warning.
...	Additional arguments passed to FUN. Important! Since R does partial matching for argument names, using arguments like f(x, a) and passing a = 1 to GenD(), Grad(), or Jacobian() will result in the interpretation acc.order = 1 because the latter is a named argument of these functions. Either supply the full names of all similar-looking arguments or name arguments differently (e.g. a0 instead of a).

Value

Matrix where each row corresponds to a function output and each column to an input coordinate. For scalar-valued functions, the output is returned as a row matrix. Timing attributes are described in GenD().

See Also

GenD(), Grad()

Examples

```
slowFun <- function(x) {Sys.sleep(0.01); sum(sin(x))}
slowFunVec <- function(x) {Sys.sleep(0.01);
                           c(sin = sum(sin(x)), exp = sum(exp(x)))}
true.g <- cos(1:4)  # Analytical gradient
true.j <- rbind(cos(1:4), exp(1:4)) # Analytical Jacobian
x0 <- c(each = 1, par = 2, is = 3, named = 4)

# Compare computation times
system.time(g.slow <- numDeriv::grad(slowFun, x = x0) - true.g)
system.time(j.slow <- numDeriv::jacobian(slowFunVec, x = x0) - true.j)
system.time(g.fast <- Grad(slowFun, x = x0, cores = 2) - true.g)
system.time(j.fast <- Jacobian(slowFunVec, x = x0, cores = 2) - true.j)
system.time(j.fast4 <- Jacobian(slowFunVec, x = x0, acc.order = 4, cores = 2) - true.j)
```

```
stopParallel()

# Compare accuracy
rownames(j.slow) <- paste0("numDeriv.jac.", c("sin", "exp"))
rownames(j.fast) <- paste0("pnd.jac.order2.", rownames(j.fast))
rownames(j.fast4) <- paste0("pnd.jac.order4.", rownames(j.fast4))
# Discrepancy
print(rbind(numDeriv.grad = g.slow, pnd.Grad = g.fast, j.slow, j.fast, j.fast4), 2)
# The order-4 derivative is more accurate for functions
# with non-zero third and higher derivatives -- look at pnd.jac.order.4
```

`plot.stepsize` *Step-size selection visualisation*

Description

Plots step-size diagnostics, highlighting various ranges obtained during step-size selection for numerical differentiation. Works for all implemented methods.

Usage

```
## S3 method for class 'stepsize'
plot(x, ...)

## S3 method for class 'gradstep'
plot(x, index = 1, ...)
```

Arguments

`x`	List returned by `step...` functions.
`...`	Additional graphical parameters passed to `plot()`.
`index`	Integer index or character name of the coordinate of `x` to plot.

Value

The plotted step-size result.

Examples

```
sCR <- step.CR(sin, 1)
sK <- step.K(sin, 1)
plot(sCR)
plot(sK)
f <- function(x) prod(sin(x))
s <- gradstep(f, 1:4, method = "CR")
plot(s, 3)
```

print.Hessian *Numerical Hessians*

Description

Computes the second derivatives of a function with respect to all combinations of its input coordinates. Arbitrary accuracies and sides for different coordinates of the argument vector are supported.

Usage

```
## S3 method for class 'Hessian'
print(
  x,
  digits = 4,
  shave.spaces = TRUE,
  begin = "",
  sep = "  ",
  end = "",
  ...
)

Hessian(
  FUN,
  x,
  side = 0,
  acc.order = 2,
  h = NULL,
  h0 = NULL,
  control = list(),
  f0 = NULL,
  cores = 1,
  preschedule = TRUE,
  cl = NULL,
  func = NULL,
  ...
)
```

Arguments

x	Numeric vector or scalar: point at which the derivative is estimated. FUN(x) must return a finite value.
digits	Positive integer: the number of digits after the decimal point in scientific notation (i.e. one less than the number of significant digits).
shave.spaces	Logical: if true, removes spaces to ensure compact output; if false, results in nearly fixed-width output (almost).

begin	A character to put at the beginning of each line, usually "", "(", or "c(" (the latter is useful if console output is used in calculations).
sep	The column delimiter, usually " ", "|", "&" (for LaTeX), or ", ".
end	A character to put at the end of each line, usually "" or ")".
...	Additional arguments passed to FUN, except compatibility controls method and method.args.
FUN	A function returning a numeric scalar. Vector-valued functions are not supported; use Jacobian() instead. If instead of FUN, func is passed, it will be reassigned to FUN with a warning.
side	Integer scalar or vector indicating difference type: 0 for central, 1 for forward, and -1 for backward differences. Central differences are recommended unless computational cost is prohibitive.
acc.order	Integer scalar or vector specifying the desired accuracy order for each input coordinate. The error typically scales as $O(h^{\text{acc.order}})$.
h	Numeric scalar or vector specifying the step size for the numerical difference. Automatic step-size selection via character values is not implemented for Hessians. Must be length 1 or match length(x). Matrices of step sizes are not supported. Suggestions how to handle all pairs of coordinates are welcome.
h0	Reserved for an initial step size for automatic step-size selection. Currently ignored.
control	Reserved for automatic step-size selection controls. Currently ignored.
f0	Reserved for a precomputed function value. Currently ignored.
cores	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.
preschedule	Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.
cl	An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.
func	Deprecated; for numDeriv::hessian() compatibility only.

Details

Additional arguments to the print method (. . .) are ignored.

The optimal step size for 2nd-order-accurate central-differences-based Hessians is of the order Mach.eps^(1/4) to balance the Taylor series truncation error with the rounding error. However, selecting the best step size typically requires knowledge of higher-order cross derivatives and is highly technically involved. Future releases will allow character arguments to invoke automatic data-driven step-size selection.

Some numerical packages use the option (or even the default behaviour) of computing not only the i < j cross-partials for the Hessian, but all pairs of i and j. The upper and lower triangular matrices are filled, and the matrix is averaged with its transpose to obtain a Hessian – this is the behaviour of optimHess(). However, it can be shown that H[i, j] and H[j, i] use the same evaluation grid, and

with a single parallelisable evaluation of the function on that grid, no symmetrisation is necessary because the result is mathematically and computationally identical. In pnd, only the upper triangular matrix is computed, saving time and ensuring unambiguous results owing to the interchangeability of summation terms (ignoring the numerical error in summation as there is nothing that can be done apart from compensated summation, e.g. via Kahan's algorithm).

Value

A matrix with as many rows and columns as `length(x)`. Unlike the output of `numDeriv::hessian()`, this output preserves the names of `x`. With `options(pnd.timing = TRUE)`, `timing` records preparation, evaluation and total elapsed seconds; `function.calls` counts unique stencil evaluations.

See Also

`Grad()` for gradients, `GenD()` for generalised numerical differences.

Examples

```
f <- function(x) prod(sin(x))
Hessian(f, 1:4)
# Large matrices

  system.time(Hessian(f, 1:100))
```

---

printMat *Print a matrix with separators*

Description

Print a matrix with separators

Usage

```
printMat(
  x,
  digits = 3,
  shave.spaces = TRUE,
  begin = "",
  sep = "  ",
  end = "",
  print = TRUE,
  format = TRUE
)
```

Arguments

x	A numeric matrix to print line by line.
digits	Positive integer: the number of digits after the decimal point in scientific notation (i.e. one less than the number of significant digits).
shave.spaces	Logical: if true, removes spaces to ensure compact output; if false, results in nearly fixed-width output (almost).
begin	A character to put at the beginning of each line, usually "", "(", or "c(" (the latter is useful if console output is used in calculations).
sep	The column delimiter, usually " ", "|", "&" (for LaTeX), or ", ".
end	A character to put at the end of each line, usually "" or ")".
print	If TRUE, outputs the lines of the matrix rows into the console.
format	If FALSE, skips the formatting part.

Value

An invisible character vector of formatted rows.

Examples

```
x <- matrix(c(-1234567, 12345.67, 123.4567,
              1.23456, -1.23456e-1, 0,
              1.23456e-4, 1.23456e-2, -1.23456e-6), nrow = 3)
printMat(x)
printMat(x, 2, TRUE, "c(", ", ", ")")  # Ready row vectors
```

---

runParallel *Run a function in parallel over a list (internal use only)*

Description

Run a function in parallel over a list (internal use only)

Usage

```
runParallel(FUN, x, cores = 1L, cl = NULL, preschedule = FALSE)
```

Arguments

FUN	A function of only one argument. If there are more arguments, use a wrapper such as FUN2 <- function(x) FUN(x, a = 1) and call it.
x	A list to parallelise the evaluation of FUN over: either numbers or expressions.
cores	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.

cl	An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.
preschedule	Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.

Details

If no custom cluster is supplied or no default cluster is registered, options(pnd.parallel = "auto") uses mclapply() on Linux/macOS and automatically creates reusable PSOCK workers on Windows. Set options(pnd.parallel = "PSOCK") to use these workers on any platform, or "mclapply" to explicitly request forking on Linux/macOS. Repeated calls reuse the pool; larger requests replace it and smaller requests use a subset. Call stopParallel() to release it. Package unloading and session exit also attempt clean-up. Supplied/default clusters remain owned by the caller; their size overrides cores (the CRAN check limit of 2 is handled separately). Nested calls inside automatically created workers run serially unless given a separate cluster.

PSOCK workers are separate R sessions. The data needed by FUN should be in its enclosing function environment or passed as arguments; objects used only from the global environment need explicit export to a custom user-supplied cluster. This also applies to unevaluated extra arguments referring to globals when dimension checks are skipped. Use a local objective closure to avoid this dependency.

Value

The value that lapply(x, FUN) would have returned.

Examples

```
fslow <- function(x) Sys.sleep(x)
x <- rep(0.05, 6)
cl <- parallel::makeCluster(2)
print(t1 <- system.time(runParallel(fslow, x)))
print(t2 <- system.time(runParallel(fslow, x, cl = cl)))
print(t3 <- system.time(runParallel(fslow, x, cores = 2)))
parallel::stopCluster(cl)
cat("Parallel overhead at 2 cores: ", round(t2[3]*200/t1[3]-100), "%\n", sep = "")
# Ignore on Windows
cat("makeCluster() overhead at 2 cores: ", round(100*t2[3]/t3[3]-100), "%\n", sep = "")
# Store objective data locally so PSOCK workers can use it
f <- local({scale <- 2; function(x) scale * x^2})
runParallel(f, 1:3, cores = 2)
stopParallel()
```

`solveVandermonde` *Numerically stable non-confluent Vandermonde system solver*

Description

Numerically stable non-confluent Vandermonde system solver

Usage

```
solveVandermonde(s, b)
```

Arguments

`s`	Numeric vector of stencil points defining the Vandermonde matrix on the left-hand side, where each element $S_{i,j}$ is calculated as `s[j]^(i-1)`.
`b`	Numeric vector of the right-hand side of the equation. This vector must be the same length as `s`.

Details

This function utilises the (Björck and Pereyra 1970) algorithm for an accurate solution to non-confluent Vandermonde systems, which are known for their numerical instability. Unlike Gaussian elimination, which suffers from ill conditioning, this algorithm achieves numerical stability through exploiting the ordering of the stencil. An unsorted stencil will trigger a warning. Additionally, the stencil must contain unique points, as repeated values make the Vandermonde matrix confluent and therefore non-invertible.

This implementation is a verbatim translation of Algorithm 4.6.2 from (Golub and Van Loan 2013), which is robust against the issues typically associated with Vandermonde systems.

See (Higham 1987) for an in-depth error analysis of this algorithm.

Value

A numeric vector of coefficients solving the Vandermonde system, matching the length of `s`.

References

Björck Å, Pereyra V (1970). "Solution of Vandermonde systems of equations." *Mathematics of computation*, **24**(112), 893–903.

Golub GH, Van Loan CF (2013). *Matrix computations*, 4 edition. Johns Hopkins University Press.

Higham NJ (1987). "Error analysis of the Björck-Pereyra algorithms for solving Vandermonde systems." *Numerische Mathematik*, **50**(5), 613–632.

Examples

```
# Approximate the 4th derivative on a non-negative stencil
solveVandermonde(s = 0:5, b = c(0, 0, 0, 0, 24, 0))

# Small numerical inaccuracies: note the 6.66e-15 in the 4th position --
# it should be rounded towards zero:
solveVandermonde(s = -3:3, b = c(0, 1, rep(0, 5))) * 60
```

`step.CR` *Curtis–Reid automatic step selection*

Description

Curtis–Reid automatic step selection

Usage

```
step.CR(
  FUN,
  x,
  deriv.order = 1,
  acc.order = 1,
  h0 = NULL,
  max.rel.error = .Machine$double.eps^(7/8),
  aim = NULL,
  tol = NULL,
  range = NULL,
  maxit = 20L,
  seq.tol = 1e-04,
  cores = 1,
  preschedule = getOption("pnd.preschedule", TRUE),
  cl = NULL,
  ...
)
```

Arguments

`FUN`	Function for which the optimal numerical derivative step size is needed.
`x`	Numeric scalar: the point at which the derivative is computed and the optimal step size is estimated.
`deriv.order`	Order of the derivative (d in $\frac{d^d f}{dx^d}$) for which a numerical approximation is needed.
`acc.order`	Order of accuracy: defines how the approximation error scales with the step size h, specifically $O(h^a)$, where a is the accuracy order and depends on the higher-order derivatives of the function.
`h0`	Numeric scalar: initial step size, defaulting to `stepx(x, deriv.order, acc.order)`.

`max.rel.error`	Error bound for the relative function-evaluation error ($\frac{\hat{f}(\hat{x})-f(x)}{f(x)}$). Measures how noisy a function is. If the function is relying on numerical optimisation routines, consider setting to `sqrt(.Machine$double.eps)`. If the function has full precision to the last bit, set to `.Machine$double.eps/2`.
`aim`	Positive real scalar: desired ratio of truncation-to-rounding error. The original version over-estimates the truncation error, hence a higher `aim` is recommended. For the modernised version, aim should be equal to `deriv.order / acc.order`.
`tol`	Numeric scalar greater than 1: tolerance multiplier for determining when to stop the algorithm based on the current estimate being between `aim/tol` and `aim*tol`.
`range`	Numeric vector of length 2 defining the valid search range for the step size.
`maxit`	Integer: maximum number of algorithm iterations to prevent infinite loops in degenerate cases.
`seq.tol`	Numeric scalar: maximum relative difference between old and new step sizes for declaring convergence.
`cores`	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.
`preschedule`	Logical: if `TRUE`, enables pre-scheduling for `mclapply()` or disables load balancing with `parLapply()`. Recommended for functions that take less than 0.1 s per evaluation.
`cl`	An optional user-supplied `cluster` object (created by `makeCluster` or similar functions). If not `NULL`, the code uses `parLapply()` (if `preschedule` is `TRUE`) or `parLapplyLB()` on that cluster on all platforms.
`...`	Passed to `FUN`.

Details

This function computes the optimal step size for central differences using the (Curtis and Reid 1974) algorithm. If the estimated truncation error is near zero, the search stops.

For the modified method, error estimates compare accuracy orders `acc.order` and `acc.order + 2`.

TODO: mention that f must be one-dimensional

The arguments passed to `...` must not partially match those of `step.CR()`. For example, if `cl` exists, then, attempting to avoid cluster export by using `step.CR(f, x, h = 1e-4, cl = cl, a = a)` will result in an error: `a` matches `aim` and `acc.order`. Redefine the function for this argument to have a name that is not equal to the beginning of one of the arguments of `step.CR()`.

The original version of the article uses `deriv.order = 1`, `acc.order = 1`, `aim = 100`.

Value

A list similar to the one returned by `optim()`:

- `par` – the optimal step size found.
- `value` – the estimated numerical derivative (using central differences; especially useful for computationally expensive functions).
- `counts` – the number of recorded iterations.

- `abs.error` – an estimate of the truncation and rounding errors.
- `exitcode` – an integer code indicating the termination status:
  - `0` – Optimal termination within tolerance.
  - `1` – Estimated truncation error is near zero; a larger step size is preferred.
  - `2` – No change in step size within tolerance.
  - `3` – Solution lies at the boundary of the allowed value range.
  - `4` – Step trimmed to 0.1|x| when |x| is not tiny and within range.
  - `5` – Maximum number of iterations reached.
- `message` – A summary message of the exit status.
- `iterations` – A list including the full step size search path, argument grids, function values on those grids, estimated error ratios, and estimated derivative values.

References

Curtis AR, Reid JK (1974). "The Choice of Step Lengths When Using Differences to Approximate Jacobian Matrices." *IMA Journal of Applied Mathematics*, **13**(1), 121–126. doi:10.1093/imamat/13.1.121.

Examples

```
f <- function(x) x^4
step.CR(x = 2, f)  # Wrap plot(...) around each of these functions
step.CR(x = 2, f, h0 = 1e-3)
step.CR(x = 2, f, acc.order = 2)
step.CR(x = 2, f, deriv.order = 2)
step.CR(x = 2, f, deriv.order = 3, acc.order = 4)

# A bad start: too far away
step.CR(x = 2, f, h0 = 1000)  # Bad exit code + a suggestion to extend the range
step.CR(x = 2, f, h0 = 1000, range = c(1e-10, 1e5))  # Problem solved

library(parallel)
cl <- makePSOCKcluster(names = 2, outfile = "")
abc <- 2
f <- function(x, abc) {Sys.sleep(0.02); abc*sin(x)}
x <- pi/4
system.time(step.CR(f, x, h = 1e-4, cores = 1, abc = abc))  # To remove speed-ups
system.time(step.CR(f, x, h = 1e-4, cores = 2, abc = abc))  # Faster
f2 <- function(x) f(x, abc)
clusterExport(cl, c("f2", "f", "abc"))
system.time(step.CR(f2, x, h = 1e-4, cl = cl))  # Also fast
stopCluster(cl)
stopParallel()
```

step.DV *Dumontet–Vignes automatic step selection*

Description

Dumontet–Vignes automatic step selection

Usage

```
step.DV(
  FUN,
  x,
  h0 = 1e-05 * max(abs(x), sqrt(.Machine$double.eps)),
  range = h0/c(1e+06, 1e-06),
  max.rel.error = .Machine$double.eps^(7/8),
  ratio.limits = c(2, 15),
  maxit = 40L,
  cores = 1,
  preschedule = getOption("pnd.preschedule", TRUE),
  cl = NULL,
  ...
)
```

Arguments

FUN	Function for which the optimal numerical derivative step size is needed.
x	Numeric scalar: the point at which the derivative is computed and the optimal step size is estimated.
h0	Numeric scalar: initial step size, defaulting to `1e-5 * max(abs(x), sqrt(.Machine$double.eps))`. This step size for first derivatives is internally translated into the initial step size for third derivatives by multiplying it by the machine epsilon raised to the power -2/15.
range	Numeric vector of length 2 defining the valid search range for the step size.
max.rel.error	Positive numeric scalar > 0 indicating the maximum relative error of function evaluation. For highly accurate functions with all accurate bits it is equal to `.Machine$double.eps/2`. For noisy functions (derivatives, integrals, output of optimisation routines etc.), it is higher, typically `sqrt(.Machine$double.eps)`. Dumontet and Vignes recommend `.Machine$double.eps^(3/4)` = `2e-12` for common functions.
ratio.limits	Numeric vector of length 2 defining the acceptable ranges for step size: the algorithm stops if the relative perturbation of the third derivative by amplified rounding errors falls within this range.
maxit	Maximum number of algorithm iterations to avoid infinite loops in cases the desired relative perturbation factor cannot be achieved within the given range. Consider extending the range if this limit is reached.

`cores`	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.
`preschedule`	Logical: if `TRUE`, enables pre-scheduling for `mclapply()` or disables load balancing with `parLapply()`. Recommended for functions that take less than 0.1 s per evaluation.
`cl`	An optional user-supplied `cluster` object (created by `makeCluster` or similar functions). If not `NULL`, the code uses `parLapply()` (if `preschedule` is `TRUE`) or `parLapplyLB()` on that cluster on all platforms.
`...`	Passed to FUN.

Details

This function computes the optimal step size for central differences using the (Dumontet and Vignes 1977) algorithm. If the estimated third derivative is exactly zero, the function assumes a third derivative of 1 to prevent division-by-zero errors.

Note: the iteration history tracks the third derivative, not the first.

Value

A list similar to the one returned by `optim()`:

- `par` – the optimal step size found.
- `value` – the estimated numerical first derivative (using central differences).
- `counts` – the number of iterations (each iteration includes four function evaluations).
- `abs.error` – an estimate of the truncation and rounding errors.
- `exitcode` – an integer code indicating the termination status:
  - `0` – Optimal termination within tolerance.
  - `1` – Third derivative is zero; large step size preferred.
  - `3` – Solution lies at the boundary of the allowed value range.
  - `4` – Step trimmed to 0.1|x| when |x| is not tiny and within range.
  - `5` – Maximum number of iterations reached; optimal step size is within the allowed range.
  - `6` – Maximum number of iterations reached; optimal step size was outside allowed range and had to be snapped to a boundary or to 0.1|x|.
  - `7` – No search was performed (used when `maxit = 1`).
- `message` – A summary message of the exit status.
- `iterations` – A list including the full step size search path (note: for the third derivative), argument grids, function values on those grids, and estimated third derivative values.

References

Dumontet J, Vignes J (1977). "Détermination du pas optimal dans le calcul des dérivées sur ordinateur." *RAIRO. Analyse numérique*, **11**(1), 13–25. doi:10.1051/m2an/1977110100131.

Examples

```
f <- function(x) x^4
step.DV(x = 2, f)
step.DV(x = 2, f, h0 = 1e-3)

# Alternative plug-in estimator with only one evaluation of f'''
step.DV(x = 2, f, maxit = 1)
step.plugin(x = 2, f)
```

---

step.K *Kink-based step selection*

Description

Optimal step-size search by fitting the rounding and truncation branches of observed error estimates and locating the transition between them.

Usage

```
step.K(
  FUN,
  x,
  h0 = NULL,
  deriv.order = 1,
  acc.order = 2,
  range = NULL,
  shrink.factor = 0.5,
  max.rel.error = .Machine$double.eps^(7/8),
  cores = 1,
  preschedule = getOption("pnd.preschedule", TRUE),
  cl = NULL,
  ...
)
```

Arguments

FUN	Function for which the optimal numerical derivative step size is needed.
x	Numeric scalar: the point at which the derivative is computed and the optimal step size is estimated.
h0	Numeric scalar: initial step size, defaulting to `2^round(log2(0.001 * max(abs(x), 1)))`.
deriv.order	Integer or vector of integers indicating the desired derivative order, $\mathrm{d}^d/\mathrm{d}x^d$, for each element of x.
acc.order	Integer or vector of integers specifying the desired accuracy order for each element of x. The truncation error will be of the order $O(h^{\mathrm{acc.order}})$.

`range`	Positive finite vector of length 2 defining the initial search interval. A narrow interval is extended, with a warning, to cover steps on both sides of the usual machine-precision estimate. The interval used is saved in `args$range`.
`shrink.factor`	A positive scalar less than 1 that is used to create a sequence of step sizes. The recommended value is 0.5. Change to 0.25 for a faster search. This number should be a negative power of 2 for the most accurate representation.
`max.rel.error`	Error bound for the relative function-evaluation error ($\frac{\hat{f}(\hat{x})-f(x)}{f(x)}$). Measures how noisy a function is. If the function is relying on numerical optimisation routines, consider setting to `sqrt(.Machine$double.eps)`. If the function has full precision to the last bit, set to `.Machine$double.eps/2`.
`cores`	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.
`preschedule`	Logical: if `TRUE`, enables pre-scheduling for `mclapply()` or disables load balancing with `parLapply()`. Recommended for functions that take less than 0.1 s per evaluation.
`cl`	An optional user-supplied `cluster` object (created by `makeCluster` or similar functions). If not `NULL`, the code uses `parLapply()` (if `preschedule` is `TRUE`) or `parLapplyLB()` on that cluster on all platforms.
`...`	Passed to FUN.

Details

This function estimates the optimal step size by fitting a V-shaped curve to error estimates on log–log axes, i.e. finding a regression kink, using unconstrained BFGS optimisation and a mean pseudo-Huber penalty, which limits the influence of irregular estimates. The rounding and truncation branches have slopes `-deriv.order` and `acc.order`; for two-point central differences, these are -1 and 2, respectively. The fitted branches are then used to minimise the estimated squared error of the interpolated derivative. This accounts for the different noise amplification of the target and auxiliary formulae, and for function values shared by neighbouring stencils. The calculation assumes independent evaluation errors with similar variances near `x`. If too few estimates follow the expected truncation slope, a rounding-error fallback is used instead. A valid truncation branch requires 3 consecutive centred slopes within 10% of `acc.order`, without bridging missing estimates. The fit uses all finite estimates up to the end of the longest such run. A failed fit or a kink with observations on only one side invokes the fallback.

The accuracy of the estimated higher-order derivatives starts at 4; zero or non-finite estimates are retried at accuracy 2 using the inner nodes. Auxiliary weighted sums within their arithmetic rounding bound are treated as unresolved rather than positive observations in the fit. All stencils reuse the evaluated grid; no extra steps are evaluated for retries. On fallback, choose the widest evaluated step for which the rounding intervals of all finer usable derivatives have a common intersection, with three consecutive truncation estimates no greater than their rounding bounds. Changes between adjacent target derivatives must also meet that condition, to detect higher-order truncation. If none exists, keep the rounding-error fallback.

Interpolation uses the existing derivative estimates and their leading dependence on `h^acc.order`, avoiding further function evaluations. For continuous functions, $f(x + \delta) \approx f(x)$ for small δ.

The estimated step must lie in the search interval. Interpolation requires two finite neighbouring derivative estimates; otherwise the nearest usable grid step is returned with its derivative estimate.

Steps with infinite or coincident stencil arguments are excluded. The function stops if no usable grid step remains. These checks reuse the evaluated grid and make no further function calls.

Value

A list similar to the one returned by `optim()`:

- `par` – the optimal step size found.
- `value` – the estimated numerical derivative (using central differences).
- `counts` – the number of paired grid points, with one additional central evaluation for even derivative orders.
- `abs.error` – separate fitted error components, or interpolated grid estimates on fallback; the estimated rounding component describes noise in the truncation-error estimates.
- `exitcode` – an integer code indicating the termination status:
  - `0` – A V-shaped curve was fitted without errors or sanity-check failures.
  - `1` – The truncation branch is ill-behaved; a fail-safe value is returned.
  - `2` – Too few positive finite estimates; a fail-safe value is returned.
  - `3` – The fitted kink has observations on only one side; a fallback is returned.
  - `4` – Step trimmed to 0.1|x| when |x| is not tiny and within range.
  - `5` – The fit or selected step failed a safeguard; a fallback is returned.
- `message` – A summary message of the exit status.
- `iterations` – A list including the step and argument grids, function values on those grids, estimated derivative values, estimated error values, and predicted model-based errors. The `fhigher` column contains the higher-order auxiliary derivative; `aprime` records its selected accuracy order. `fit` contains the initial and fitted log-kink coordinates (`gamma`, `beta`), unscaled initial residual `mad`, fixed knee `rho` (at least one binary-logarithm unit), fitting `subset`, mean pseudo-Huber `loss`, optimiser `convergence` and whether the fit was `accepted`; a `NULL` is returned if no fit was attempted. `noise` records the auxiliary-to-target noise ratio and the overlap of neighbouring target stencils; `calibrated` indicates whether these were used to select the step for interpolation. `fallback` records the quantities used when the truncation branch cannot be found. Its `target` is the rounding error predicted at the minimum of the combined-error model. It is `NA` when the estimates do not identify a finite minimum. The calculation uses the median absolute higher-derivative estimate and the function values at the smallest step for which all points in the difference formula are available. `mean.sign` indicates whether rounding estimates tend to rise or fall as steps increase. `stable.step` is the evaluated step supported by agreement with finer estimates, or `NA` if that check found no suitable step; `par` retains the model's initial proposal.

References

There are no references for Rd macro `\insertAllCites` on this help page.

Examples

```
plot(step.K(sin, 1))
step.K(exp, 1, range = c(1e-12, 1e-0))
step.K(atan, 1)
```

```
# Edge case 1: function symmetric around x0, zero truncation error
step.K(sin, pi/2)
step.K(sin, pi/2, shrink.factor = 0.8)

# Edge case 2: the truncation error is always zero and f(x0) = 0
suppressWarnings(step.K(function(x) x^2, 0))
# Other edge cases
step.K(function(x) x^2, 1)
step.K(function(x) x^4, 0)
step.K(function(x) x^4, 0.1)
step.K(function(x) x^6 - x^4, 0.1)
step.K(atan, 3/4)
step.K(exp, 2, range = c(1e-16, 1e+1))
```

`step.M` *Mathur's AutoDX-like automatic step selection*

Description

Mathur's AutoDX-like automatic step selection

Usage

```
step.M(
  FUN,
  x,
  h0 = NULL,
  deriv.order = 1,
  acc.order = 2,
  range = NULL,
  shrink.factor = 0.5,
  min.valid.slopes = 5L,
  seq.tol = 0.1,
  correction = TRUE,
  max.rel.error = .Machine$double.eps^(7/8),
  cores = 1,
  preschedule = getOption("pnd.preschedule", TRUE),
  cl = NULL,
  ...
)
```

Arguments

`FUN`	Function for which the optimal numerical derivative step size is needed.
`x`	Numeric scalar: the point at which the derivative is computed and the optimal step size is estimated.

`h0` Numeric scalar: initial step size, defaulting to `2^round(log2(0.001 * max(abs(x), 1)))`.

`deriv.order` Integer or vector of integers indicating the desired derivative order, $\mathrm{d}^d/\mathrm{d}x^d$, for each element of `x`.

`acc.order` Integer or vector of integers specifying the desired accuracy order for each element of `x`. The truncation error will be of the order $O(h^{\mathrm{acc.order}})$.

`range` Positive finite vector of length 2 defining the initial search interval. A narrow interval is extended, with a warning, to cover steps on both sides of the usual machine-precision estimate. The interval used is saved in `args$range`.

`shrink.factor` A positive scalar less than 1 that is used to create a sequence of step sizes. The recommended value is 0.5. Change to 0.25 for a faster search. This number should be a negative power of 2 for the most accurate representation.

`min.valid.slopes`
Positive integer: how many points must form a sequence with the correct slope with relative difference from `acc.order` less than `seq.tol`. If `shrink.factor` is small (< 0.33), consider reducing this to 4.

`seq.tol` Numeric scalar: maximum relative deviation of truncation-error slopes from `acc.order`.

`correction` Logical: if `TRUE`, adjust the selected grid step to achieve the optimal truncation-to-rounding-error ratio (see Details); otherwise return the selected grid step.

`max.rel.error` Error bound for the relative function-evaluation error ($\frac{\hat{f}(\hat{x})-f(x)}{f(x)}$). Measures how noisy a function is. If the function is relying on numerical optimisation routines, consider setting to `sqrt(.Machine$double.eps)`. If the function has full precision to the last bit, set to `.Machine$double.eps/2`.

`cores` Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.

`preschedule` Logical: if `TRUE`, enables pre-scheduling for `mclapply()` or disables load balancing with `parLapply()`. Recommended for functions that take less than 0.1 s per evaluation.

`cl` An optional user-supplied `cluster` object (created by `makeCluster` or similar functions). If not `NULL`, the code uses `parLapply()` (if `preschedule` is `TRUE`) or `parLapplyLB()` on that cluster on all platforms.

`...` Passed to FUN.

Details

This function computes the optimal step size for central differences using the (Mathur 2012) algorithm. It consists of the following steps.

1. Choose a reasonably large (but not too large) initial step size h_0 and a reduction factor (1/2 for fast, 1/4 for slow functions is a reasonable choice).
2. Estimate truncation errors by subtracting derivatives at neighbouring grid steps.
3. Find the leftmost range of consecutive step sizes for which the slope of the truncation error is approximately equal (within 10% tolerance) to the accuracy order and which is long enough (e.g. at least length 5).

4. Select the smallest step in this range and, if `correction = TRUE`, adjust it towards the predicted minimum of the combined rounding and truncation errors.

The adjustment multiplies the selected step by $r[(1-r^{-a})/(1+r^{d})]^{1/(a+d)}$, where r is `1/shrink.factor`, d is `deriv.order` and a is `acc.order`.

Interpolation uses the existing derivative estimates and their leading dependence on `h^acc.order`, avoiding further function evaluations. For continuous functions, $f(x+\delta) \approx f(x)$ for small δ. Only the existing grid is used; no higher-accuracy retries or extra candidate steps are evaluated. Positive near-zero error estimates are retained, as in the original algorithm.

Value

A list similar to the one returned by `optim()`:

- `par` – the optimal step size found.
- `value` – the estimated numerical derivative (using central differences).
- `counts` – the number of paired grid points, with one additional central evaluation for even derivative orders.
- `abs.error` – an estimate of the truncation and rounding errors.
- `exitcode` – an integer code indicating the termination status:
  - `0` – Optimal termination due to a sequence of correct reductions.
  - `1` – Reductions are slightly outside the tolerance.
  - `2` – Tolerances are significantly violated; an approximate minimum is returned.
  - `3` – Not enough finite function values; a rule-of-thumb value is returned.
  - `4` – Step trimmed to 0.1|x| when |x| is not tiny and within range.
- `message` – A summary message of the exit status.
- `iterations` – A list including the step and argument grids, function values on those grids, estimated derivative values, and estimated error values. `fallback` records the quantities used when the truncation branch cannot be found. Its `target` is the rounding error predicted at the minimum of the combined-error model. It is `NA` when the estimates do not identify a finite minimum. The calculation uses the median absolute higher-derivative estimate and the function values at the smallest step for which all points in the difference formula are available. `mean.sign` indicates whether rounding estimates tend to rise or fall as steps increase.

References

Mathur R (2012). *An Analytical Approach to Computing Step Sizes for Finite-Difference Derivatives*. Ph.D. thesis, University of Texas at Austin. http://hdl.handle.net/2152/ETD-UT-2012-05-5275.

Examples

```
f <- function(x) x^4  # The derivative at 1 is 4
step.M(x = 1, f)
step.M(x = 1, f, h0 = 1e-9) # Starting low
step.M(x = 1, f, h0 = 1000) # Starting high

f <- sin  # The derivative at pi/4 is sqrt(2)/2
```

```
plot(step.M(x = pi/2, f))
plot(step.M(x = pi/4, f))
step.M(x = pi/4, f, h0 = 1e-9) # Starting low
step.M(x = pi/4, f, h0 = 1000) # Starting high
# where the truncation error estimate is invalid
```

`step.plugin` *Plug-in step selection*

Description

Plug-in step selection

Usage

```
step.plugin(
  FUN,
  x,
  h0 = max(1e-05 * abs(x), stepx(x, deriv.order = 3)),
  max.rel.error = .Machine$double.eps^(7/8),
  range = h0/c(10000, 1e-04),
  cores = 1,
  preschedule = getOption("pnd.preschedule", TRUE),
  cl = NULL,
  ...
)
```

Arguments

`FUN`	Function for which the optimal numerical derivative step size is needed.
`x`	Numeric scalar: the point at which the derivative is computed and the optimal step size is estimated.
`h0`	Numeric scalar: initial step-size seed, defaulting to `max(1e-5*abs(x), stepx(x, deriv.order = 3))`. The third derivative is estimated using larger spacing `h0 * .Machine$double.eps^(-2/15)`.
`max.rel.error`	Positive numeric scalar > 0 indicating the maximum relative error of function evaluation. For highly accurate functions with all accurate bits it is equal to `.Machine$double.eps/2`. For noisy functions (derivatives, integrals, output of optimisation routines etc.), it is higher, typically `sqrt(.Machine$double.eps)`. Dumontet and Vignes recommend `.Machine$double.eps^(3/4)` = `2e-12` for common functions.
`range`	Numeric vector of length 2 defining the valid search range for the step size.
`cores`	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.

preschedule	Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.
cl	An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.
...	Passed to FUN.

Details

This function computes the optimal step size for central differences using the plug-in approach. The plug-in step-size rule is

$$h = \left(1.5\varepsilon \left| \frac{f(x)}{f'''(x)} \right| \right)^{1/3},$$

where ε is max.rel.error. The function value and third derivative are approximated on the initial evaluation grid. Zero or near-zero third-derivative estimates use safeguarded step sizes.

Value

A list similar to the one returned by optim():

- par – the optimal step size found.
- value – the estimated numerical first derivative (using central differences).
- counts – the number of iterations (in this case, it is 2).
- abs.error – an estimate of the truncation and rounding errors.
- exitcode – an integer code indicating the termination status:
  - 0 – Termination with checks passed tolerance.
  - 1 – Third derivative is exactly zero; large step size preferred.
  - 2 – Third derivative is too close to zero; large step size preferred.
  - 3 – Solution lies at the boundary of the allowed value range.
  - 4 – Step trimmed to 0.1|x| when |x| is not tiny and within range.
- message – A summary message of the exit status.
- iterations – A list including the two-step size search path, argument grids, function values on those grids, and estimated third derivative values.

References

There are no references for Rd macro \insertAllCites on this help page.

Examples

```
f <- function(x) x^4
step.plugin(x = 2, f)
step.plugin(x = 0, f)  # f''' = 0, setting a large one
```

`step.SW` *Stepleman–Winarsky automatic step selection*

Description

Stepleman–Winarsky automatic step selection

Usage

```
step.SW(
  FUN,
  x,
  h0 = 1e-05 * (abs(x) + (x == 0)),
  shrink.factor = 0.5,
  range = h0/c(1e+12, 1e-08),
  seq.tol = 1e-04,
  max.rel.error = .Machine$double.eps/2,
  maxit = 40L,
  cores = 1,
  preschedule = getOption("pnd.preschedule", TRUE),
  cl = NULL,
  ...
)
```

Arguments

`FUN`	Function for which the optimal numerical derivative step size is needed.
`x`	Numeric scalar: the point at which the derivative is computed and the optimal step size is estimated.
`h0`	Numeric scalar: initial step size, defaulting to a relative step of slightly greater than .Machine$double.eps^(1/3) (or absolute step if `x == 0`).
`shrink.factor`	A positive scalar less than 1 that is used to multiply the step size during the search. The authors recommend 0.25, but this may result in earlier termination at slightly sub-optimal steps. Change to 0.5 for a more thorough search.
`range`	Numeric vector of length 2 defining the valid search range for the step size.
`seq.tol`	Numeric scalar: maximum relative difference between old and new step sizes for declaring convergence.
`max.rel.error`	Positive numeric scalar > 0 indicating the maximum relative error of function evaluation. For highly accurate functions with all accurate bits it is equal to half of machine epsilon. For noisy functions (derivatives, integrals, output of optimisation routines etc.), it is higher.
`maxit`	Maximum number of algorithm iterations to avoid infinite loops. Consider trying some smaller or larger initial step size `h0` if this limit is reached.
`cores`	Integer specifying the number of CPU cores used for parallel computation. Recommended to be set to the number of physical cores on the machine minus one.

preschedule	Logical: if TRUE, enables pre-scheduling for mclapply() or disables load balancing with parLapply(). Recommended for functions that take less than 0.1 s per evaluation.
cl	An optional user-supplied cluster object (created by makeCluster or similar functions). If not NULL, the code uses parLapply() (if preschedule is TRUE) or parLapplyLB() on that cluster on all platforms.
...	Passed to FUN.

Details

This function computes the optimal step size for central differences using the (Stepleman and Winarsky 1979) algorithm.

Preliminary cancellation control applies only when the endpoint values have the same non-zero sign, as in Figure 1 of the paper. If both FUN(x) and the central difference are zero in this case, the cancellation measure (equations 3.10–3.11) is undefined and the function stops with an error. Try a different h0 or another step-selection method: equal function values on a grid do not establish that the derivative is zero.

Value

A list similar to the one returned by optim():

- par – the optimal step size found.
- value – the estimated numerical first derivative (using central differences).
- counts – the preliminary and main iteration counts.
- abs.error – an estimate of the truncation and rounding errors.
- exitcode – an integer code indicating the termination status:
  - 0 – Optimal termination within tolerance.
  - 2 – No change in step size within tolerance.
  - 3 – Solution lies at the boundary of the allowed value range.
  - 4 – Step trimmed to 0.1|x| when |x| is not tiny and within range.
  - 5 – Maximum number of iterations reached.
- message – A summary message of the exit status.
- iterations – A list including the full step size search path, argument grids, function values on those grids, estimated derivative values, estimated error values, and monotonicity check results.

References

Stepleman RS, Winarsky ND (1979). "Adaptive numerical differentiation." *Mathematics of Computation*, **33**(148), 1257–1264. doi:10.1090/s0025571819790537 9698.

Examples

```
f <- function(x) x^4  # The derivative at 1 is 4
step.SW(x = 1, f)
step.SW(x = 1, f, h0 = 1e-9) # Starting too low
# Starting somewhat high leads to too many preliminary iterations
step.SW(x = 1, f, h0 = 10)
step.SW(x = 1, f, h0 = 1000) # Starting absurdly high

f <- sin  # The derivative at pi/4 is sqrt(2)/2
step.SW(x = pi/4, f)
step.SW(x = pi/4, f, h0 = 1e-9) # Starting too low
step.SW(x = pi/4, f, h0 = 0.1) # Starting slightly high
# The following two examples fail because the truncation error estimate is invalid
step.SW(x = pi/4, f, h0 = 10)   # Warning
step.SW(x = pi/4, f, h0 = 1000) # Warning
```

---

stepx *Default step size at given points*

Description

Compute an appropriate finite-difference step size based on the magnitude of x, derivative order, and accuracy order. If the function and its higher derivatives belong to the same order of magnitude, this step is near-optimal. For small x, returns a hard bound to prevent large machine-rounding errors.

Usage

```
stepx(x, deriv.order = 1, acc.order = 2, zero.tol = NULL)
```

Arguments

x	Numeric vector or scalar: the point(s) at which the derivative is estimated. Function values must be finite.
deriv.order	Integer or vector of integers indicating the desired derivative order, $\mathrm{d}^d/\mathrm{d}x^d$, for each element of x.
acc.order	Integer or vector of integers specifying the desired accuracy order for each element of x. The truncation error will be of the order $O(h^{\mathrm{acc.order}})$.
zero.tol	Small positive integer: if abs(x) >= zero.tol, then, the automatically guessed step size is relative or near-relative (x multiplied by the step), unless an auto-selection procedure is requested; otherwise, it is absolute.

Value

A numeric vector of the same length as x with positive step sizes.

Examples

```
# The step-selection function is piecewise linear in log-coordinates
plot(-12:4, stepx(10^(-12:4)), log = "y", type = "l")
stepx(10^(-10:2), deriv.order = 2, acc.order = 4)
```

`stopParallel` *Stop automatically created parallel workers*

Description

Stops the reusable PSOCK cluster created by pnd. Supplied clusters and clusters registered through `parallel::setDefaultCluster()` are left to the caller. The next parallel call creates workers again if needed. See `runParallel()`.

Usage

```
stopParallel()
```

Value

`NULL`, invisibly.

Examples

```
stopParallel()
```

Index

alignStrings, 2

checkCores, 3
checkDimensions, 4
checkDimensions(), *14*

dupRowInds, 6

fdCoef, 7
fdCoef(), *14*
formatMat, 9

GenD, 10
GenD(), *5*, *12*, *15*, *17*, *18*, *23*, *27*
generateGrid, 14
generateGrid2, 14
Grad, 15
Grad(), *5*, *10*, *12*, *17*, *23*, *27*
gradstep, 19
gradstep(), *11*, *13*, *16*, *22*

Hessian *(*print.Hessian*)*, 25
Hessian(), *15*

Jacobian, 21
Jacobian(), *5*, *10*, *12*, *17*, *18*, *23*, *26*

plot.gradstep *(*plot.stepsize*)*, 24
plot.stepsize, 24
print.checkDimensions
        *(*checkDimensions*)*, 4
print.GenD *(*GenD*)*, 10
print.gradstep *(*gradstep*)*, 19
print.Hessian, 25
print.stepsize *(*gradstep*)*, 19
printMat, 27

runParallel, 28
runParallel(), *13*, *18*, *21*, *47*

solveVandermonde, 30

step.CR, 31
step.CR(), *20*, *32*
step.DV, 34
step.DV(), *20*
step.K, 36
step.K(), *20*
step.M, 39
step.M(), *20*
step.plugin, 42
step.plugin(), *20*
step.SW, 44
step.SW(), *20*
stepx, 46
stepx(), *5*, *19*
stopParallel, 47
stopParallel(), *29*