Package 'numDeriv'

February 20, 2015

Version 2012.9-1

Title Accurate Numerical Derivatives

Description This package provide methods for calculating (usually) accurate numerical first and second order derivatives. Accurate calculations are done using Richardson's extrapolation or, when applicable, a complex step derivative is available. A simple difference method is also provided. Simple difference is (usually) less accurate but is much quicker than Richardson's extrapolation and provides a useful cross-check. Methods are provided for real scalar and vector valued functions.

Depends R (>= 1.8.1)

LazyLoad yes

License GPL-2

Copyright 2006-2011, Bank of Canada. 2012, Paul Gilbert

Author Paul Gilbert and Ravi Varadhan

Maintainer Paul Gilbert <pgilbert.ttv9z@ncf.ca>

URL http:http://optimizer.r-forge.r-project.org/

Repository CRAN

Repository/R-Forge/Project optimizer

Repository/R-Forge/Revision 679

Repository/R-Forge/DateTimeStamp 2012-09-30 23:59:38

Date/Publication 2012-10-14 06:18:55

NeedsCompilation no

R topics documented:

`numDeriv-package` *Accurate Numerical Derivatives*

Description

Calculate (accurate) numerical approximations to derivatives.

Details

The main functions are

```
grad   to calculate the gradient (first derivative) of a scalar
     real valued function (possibly applied to all elements
     of a vector argument).

jacobian  to calculate the gradient of a real m-vector valued
     function with real n-vector argument.

hessian   to calculate the Hessian (second derivative) of a scalar
     real valued function with real n-vector argument.

genD   to calculate the gradient and second derivative of a
     real m-vector valued function with real n-vector
  argument.
```

Author(s)

Paul Gilbert, based on work by Xingqiao Liu, and Ravi Varadhan (who wrote complex-step derivative codes)

References

Linfield, G. R. and Penny, J. E. T. (1989) *Microcomputers in Numerical Analysis*. New York: Halsted Press.

Fornberg, B. and Sloan, D, M. (1994) "A review of pseudospectral methods for solving partial differential equations." *Acta Numerica*, 3, 203-267.

Lyness, J. N. and Moler, C. B. (1967) "Numerical Differentiation of Analytic Functions." *SIAM Journal for Numerical Analysis*, 4(2), 202-210.

00.numDeriv.Intro *Accurate Numerical Derivatives*

Description

Calculate (accurate) numerical approximations to derivatives.

Details

See numDeriv-package (in the help system use package?numDeriv or ?"numDeriv-package") for an overview.

genD *Generate Bates and Watts D Matrix*

Description

Generate a matrix of function derivative information.

Usage

```
    genD(func, x, method="Richardson",
                  method.args=list(), ...)
    ## Default S3 method:
genD(func, x, method="Richardson",
      method.args=list(), ...)
```

Arguments

func	a function for which the first (vector) argument is used as a parameter vector.
x	The parameter vector first argument to func.
method	one of "Richardson" or "simple" indicating the method to use for the aproximation.
method.args	arguments passed to method. See grad. (Arguments not specified remain with their default values.)
...	any additional arguments passed to func. WARNING: None of these should have names matching other arguments of this function.

Details

The derivatives are calculated numerically using Richardson improvement. Methods "simple" and "complex" are not supported in this function. The "Richardson" method calculates a numerical approximation of the first and second derivatives of `func` at the point `x`. For a scalar valued function these are the gradient vector and Hessian matrix. (See `grad` and `hessian`.) For a vector valued function the first derivative is the Jacobian matrix (see `jacobian`). For the Richardson method `methods.args=list(eps=1e-4, d=0.0001, zero.tol=sqrt(.Machine$double.eps/7e-7), r=4, v=2)` is set as the default. See `grad` for more details on the Richardson's extrapolation parameters.

The the first order derivative with respect to x_i is

$$f_i'(x) =< f(x_1, \ldots, x_i + d, \ldots, x_n) - f(x_1, \ldots, x_i - d, \ldots, x_n) > /(2 * d)$$

The second order derivative with respect to x_i is

$$f_i''(x) =< f(x_1, \ldots, x_i + d, \ldots, x_n) - 2 * f(x_1, \ldots, x_n) + f(x_1, \ldots, x_i - d, \ldots, x_n) > /(d^2)$$

The second order derivative with respect to x_i, x_j is

$$f_{i,j}''(x) =< f(x_1, \ldots, x_i + d, \ldots, x_j + d, \ldots, x_n) - 2 * f(x_1, \ldots, x_n) +$$

$$f(x_1, \ldots, x_i - d, \ldots, x_j - d, \ldots, x_n) > /(2 * d^2) - (f_i''(x) + f_j''(x))/2$$

Value

A list with elements as follows: `D` is a matrix of first and second order partial derivatives organized in the same manner as Bates and Watts, the number of rows is equal to the length of the result of `func`, the first p columns are the Jacobian, and the next p(p+1)/2 columns are the lower triangle of the second derivative (which is the Hessian for a scalar valued `func`). `p` is the length of `x` (dimension of the parameter space). `f0` is the function value at the point where the matrix `D` was calculated. The `genD` arguments `func`, `x`, `d`, `method`, and `method.args` also are returned in the list.

References

Linfield, G.R. and Penny, J.E.T. (1989) "Microcomputers in Numerical Analysis." Halsted Press.

Bates, D.M. & Watts, D. (1980), "Relative Curvature Measures of Nonlinearity." J. Royal Statistics Soc. series B, 42:1-25

Bates, D.M. and Watts, D. (1988) "Non-linear Regression Analysis and Its Applications." Wiley.

See Also

`hessian`, `grad`

Examples

```
func <- function(x){c(x[1], x[1], x[2]^2)}
z <- genD(func, c(2,2,5))
```

grad *Numerical Gradient of a Function*

Description

Calculate the gradient of a function by numerical approximation.

Usage

```
grad(func, x, method="Richardson", method.args=list(), ...)

## Default S3 method:
grad(func, x, method="Richardson",
  method.args=list(), ...)
```

Arguments

func	a function with a scalar real result (see details).
x	a real scalar or vector argument to func, indicating the point(s) at which the gradient is to be calculated.
method	one of "Richardson", "simple", or "complex" indicating the method to use for the approximation.
method.args	arguments passed to method. Arguments not specified remain with their default values as specified in details
...	an additional arguments passed to func. WARNING: None of these should have names matching other arguments of this function.

Details

The function grad calculates a numerical approximation of the first derivative of func at the point x. Any additional arguments in ... are also passed to func, but the gradient is not calculated with respect to these additional arguments. It is assumed func is a scalar value function. If a vector x produces a scalar result then grad returns the numerical approximation of the gradient at the point x (which has the same length as x). If a vector x produces a vector result then the result must have the same length as x, and it is assumed that this corresponds to applying the function to each of its arguments (for example, sin(x)). In this case grad returns the gradient at each of the points in x (which also has the same length as x – so be careful). An alternative for vector valued functions is provided by jacobian.

If method is "simple", the calculation is done using a simple epsilon difference. For method "simple" methods.args=list(eps=1e-4) is the default. Only eps is used by this method.

If method is "complex", the calculation is done using the complex step derivative approach described in Lyness and Moler. This method requires that the function be able to handle complex valued arguments and return the appropriate complex valued result, even though the user may only be interested in the real-valued derivatives. For cases where it can be used, it is faster than

Richardson's extrapolation, and it also provides gradients that are correct to machine precision (16 digits). For method "complex", `methods.args` is ignored. The algorithm uses an `eps` of `.Machine$double.eps` which cannot (and should not) be modified.

If method is "Richardson", the calculation is done by Richardson's extrapolation (see e.g. Linfield and Penny, 1989, or Fornberg and Sloan, 1994.) This method should be used if accuracy, as opposed to speed, is important (but see method "complex" above). For this method `methods.args=list(eps=1e-4, d=0.0001, zero` is set as the default. `d` gives the fraction of `x` to use for the initial numerical approximation. The default means the initial approximation uses `0.0001 * x`. `eps` is used instead of `d` for elements of `x` which are zero (absolute value less than zero.tol). `zero.tol` tolerance used for deciding which elements of `x` are zero. `r` gives the number of Richardson improvement iterations (repetitions with successly smaller `d`. The default `4` general provides good results, but this can be increased to `6` for improved accuracy at the cost of more evaluations. `v` gives the reduction factor. `show.details` is a logical indicating if detailed calculations should be shown.

The general approach in the Richardson method is to iterate for `r` iterations from initial values for interval value `d`, using reduced factor `v`. The the first order approximation to the derivative with respect to x_i is

$$f_i'(x) =< f(x_1, \ldots, x_i + d, \ldots, x_n) - f(x_1, \ldots, x_i - d, \ldots, x_n) > /(2 * d)$$

This is repeated `r` times with successively smaller `d` and then Richardson extraplolation is applied.

If elements of `x` are near zero the multiplicative interval calculation using `d` does not work, and for these elements an additive calculation using `eps` is done instead. The argument `zero.tol` is used determine if an element should be considered too close to zero. In the iterations, interval is successively reduced to eventual be `d/v^r` and the square of this value is used in second derivative calculations (see genD) so the default `zero.tol=sqrt(.Machine$double.eps/7e-7)` is set to ensure the interval is bigger than `.Machine$double.eps` with the default `d`, `r`, and `v`.

Value

A real scalar or vector of the approximated gradient(s).

References

Linfield, G. R. and Penny, J. E. T. (1989) *Microcomputers in Numerical Analysis*. New York: Halsted Press.

Fornberg, B. and Sloan, D, M. (1994) "A review of pseudospectral methods for solving partial differential equations." *Acta Numerica*, 3, 203-267.

Lyness, J. N. and Moler, C. B. (1967) "Numerical Differentiation of Analytic Functions." *SIAM Journal for Numerical Analysis*, 4(2), 202-210.

See Also

jacobian, hessian, genD, numericDeriv

Examples

```
grad(sin, pi)
grad(sin, (0:10)*2*pi/10)
func0 <- function(x){ sum(sin(x))  }
grad(func0 , (0:10)*2*pi/10)

func1 <- function(x){ sin(10*x) - exp(-x) }

curve(func1,from=0,to=5)

x <- 2.04
numd1 <- grad(func1, x)
exact <- 10*cos(10*x) + exp(-x)
c(numd1, exact, (numd1 - exact)/exact)

x <- c(1:10)
numd1 <- grad(func1, x)
numd2 <- grad(func1, x, "complex")
exact <- 10*cos(10*x) + exp(-x)
cbind(numd1, numd2, exact, (numd1 - exact)/exact, (numd2 - exact)/exact)

sc2.f <- function(x){
  n <- length(x)
  sum((1:n) * (exp(x) - x)) / n
  }

sc2.g <- function(x){
  n <- length(x)
  (1:n) * (exp(x) - 1) / n
  }

x0 <- rnorm(100)
exact <- sc2.g(x0)

g <- grad(func=sc2.f, x=x0)
max(abs(exact - g)/(1 + abs(exact)))

gc <- grad(func=sc2.f, x=x0, method="complex")
max(abs(exact - gc)/(1 + abs(exact)))
```

`hessian` *Calculate Hessian Matrix*

Description

Calculate a numerical approximation to the Hessian matrix of a function at a parameter value.

Usage

```
hessian(func, x, method="Richardson", method.args=list(), ...)

    ## Default S3 method:
hessian(func, x, method="Richardson",
        method.args=list(), ...)
```

Arguments

func	a function for which the first (vector) argument is used as a parameter vector.
x	the parameter vector first argument to func.
method	one of "Richardson" or "complex" indicating the method to use for the approximation.
method.args	arguments passed to method. See grad. (Arguments not specified remain with their default values.)
...	an additional arguments passed to func. WARNING: None of these should have names matching other arguments of this function.

Details

The function hessian calculates an numerical approximation to the n x n second derivative of a scalar real valued function with n-vector argument.

The argument method can be "Richardson" or "complex". Method "simple" is not supported.

For method "complex" the Hessian matrix is calculated as the Jacobian of the gradient. The function grad with method "complex" is used, and methods.args is ignored for this (an eps of .Machine$double.eps is used - see grad for more details). However, jacobian is used in the second step, with method "Richardson" and argument methods.args is used for this. The default is methods.args=list(eps=1e-4, d=0.1, zero.tol=sqrt(.Machine$double.eps/7e-7), r=4, v=2, show.details=
(These are the defaults for hessian with method "Richardson", which are slightly different from the defaults for jacobian with method "Richardson".)

Methods "Richardson" uses genD and extracts the second derivative. For this method methods.args=list(eps=1e-4, d=0
is set as the default.

Value

An n by n matrix of the Hessian of the function calculated at the point x.

See Also

jacobian, grad, genD

Examples

```
sc2.f <- function(x){
  n <- length(x)
  sum((1:n) * (exp(x) - x)) / n
  }
```

```
  sc2.g <- function(x){
    n <- length(x)
    (1:n) * (exp(x) - 1) / n
    }

  x0 <- rnorm(5)
  hess <- hessian(func=sc2.f, x=x0)
  hessc <- hessian(func=sc2.f, x=x0, "complex")
  all.equal(hess, hessc, tolerance = .Machine$double.eps)

#  Hessian = Jacobian of the gradient
  jac  <- jacobian(func=sc2.g, x=x0)
  jacc <- jacobian(func=sc2.g, x=x0, "complex")
  all.equal(hess, jac, tolerance = .Machine$double.eps)
  all.equal(hessc, jacc, tolerance = .Machine$double.eps)
```

`jacobian` *Gradient of a Vector Valued Function*

Description

Calculate the m by n numerical approximation of the gradient of a real m-vector valued function with n-vector argument.

Usage

```
    jacobian(func, x, method="Richardson", method.args=list(), ...)

    ## Default S3 method:
jacobian(func, x, method="Richardson",
       method.args=list(), ...)
```

Arguments

`func`	a function with a real (vector) result.
`x`	a real or real vector argument to func, indicating the point at which the gradient is to be calculated.
`method`	one of `"Richardson"`, `"simple"`, or `"complex"` indicating the method to use for the approximation.
`method.args`	arguments passed to method. See `grad`. (Arguments not specified remain with their default values.)
`...`	any additional arguments passed to `func`. WARNING: None of these should have names matching other arguments of this function.

Details

For $f : R^n -> R^m$ calculate the mxn Jacobian dy/dx. The function `jacobian` calculates a numerical approximation of the first derivative of `func` at the point `x`. Any additional arguments in ... are also passed to `func`, but the gradient is not calculated with respect to these additional arguments.

If method is "Richardson", the calculation is done by Richardson's extrapolation. See `link{grad}` for more details. For this method `methods.args=list(eps=1e-4, d=0.0001, zero.tol=sqrt(.Machine$double.eps/` is set as the default.

If method is "simple", the calculation is done using a simple epsilon difference. For method "simple" `methods.args=list(eps=1e-4)` is the default. Only `eps` is used by this method.

If method is "complex", the calculation is done using the complex step derivative approach described in Lyness and Moler. This method requires that the function be able to handle complex valued arguments and return the appropriate complex valued result, even though the user may only be interested in the real case. For cases where it can be used, it is faster than Richardson's extrapolation, and it also provides gradients that are correct to machine precision (16 digits). For method "complex", `methods.args` is ignored. The algorithm uses an `eps` of `.Machine$double.eps` which cannot (and should not) be modified.

Value

A real m by n matrix.

See Also

`grad`, `hessian`, `numericDeriv`

Examples

```
func2 <- function(x) c(sin(x), cos(x))
x <- (0:1)*2*pi
jacobian(func2, x)
jacobian(func2, x, "complex")
```

Index