Package ‘dfdr’

July 22, 2025

Type Package

Title Automatic Differentiation of Simple Functions

Version 0.2.0

Description Implementation of automatically computing derivatives of functions (see Mailund Thomas (2017) <doi:10.1007/978-1-4842-2881-4>). Moreover, calculating gradients, Hessian and Jacobian matrices is possible.

License GPL-3

Encoding UTF-8

Imports methods, purrr, rlang, R6, pryr

Suggests tinytest

NeedsCompilation no

Author Thomas Mailund [aut],
Konrad Krämer [aut, cre]

Maintainer Konrad Krämer <konrad_kraemer@yahoo.de>

Repository CRAN

Date/Publication 2023-02-23 10:30:02 UTC

Contents

d *Differentiate a function for a single variable.*

Description

Differentiate a function for a single variable.

Usage

```
d(f, x, derivs = NULL)
```

Arguments

`f` The function to differentiate.

`x` The variable that f should be differentiated with respect to.

`derivs` An S4 class of type *fcts* that defines additional derivatives. See `fcts` for details.

Details

The following functions are already supported:
sin, sinh, asin, cos, cosh, acos, tan, tanh, atan, exp, log, sqrt, c, vector, numeric, rep and matrix. Notably, for the functions: c, vector, numeric, rep and matrix the function is ignored during differentiation.

Value

For example function f and symbol x:
df/dx

Examples

```
library(dfdr)
d(sin, x)

f <- function(x) -sin(x)
d(f, x)

# Initialize list
lst <- dfdr::fcts()
# The function which should be added
f <- function(x) x^2
# The dervative function of f
f_deriv <- function(x) 2*x
# add new entry to list
lst <- fcts_add_fct(lst, f, f_deriv)
g <- function(z) f(z)
d(g, z, lst)
```

fcts *S4 class* fcts

Description

A S4 class containing additional functions which can be used for calculating derivatives with d().
To create a class the function *fcts()* should be used.
Adding functions is only possible *via* the function *add_fct*.

Details

The following functions are already supported:
sin, sinh, asin, cos, cosh, acos, tan, tanh, atan, exp, log, sqrt, c, vector, numeric, rep and matrix.
Notably, for the functions: c, vector, numeric, rep and matrix the function is ignored during differentiation.

Slots

- `funs` A list containing the specified functions. This slot should not be accessed and is used only internally.

See Also

d()

Examples

```
library(dfdr)
# Initialize list
lst <- dfdr::fcts()

# The function which should be added
f <- function(x) x^2
# The dervative function of f
f_deriv <- function(x) 2*x

# add new entry to list
lst <- fcts_add_fct(lst, f, f_deriv)

g <- function(z) f(z)
df <- d(g, z, lst)
df
```

fcts_add_fct *appending a S4 class of type* fcts

Description

A function which appends a S4 class of type *fcts* with a new function-derivative pair.

Usage

```
fcts_add_fct(lst, f, f_deriv, keep = FALSE)
```

Arguments

`lst`	is the S4 class of type *fcts*. Newly created by `fcts()`
`f`	is the function which should be differentiated. The argument has to be of type function.
`f_deriv`	is a function defining the derivative of *f*. The argument has to be of type function.
`keep`	is a logical value. If set to TRUE the function *f* is ignored of `d()`. The default value is FALSE.

Details

The following functions are already supported:
sin, sinh, asin, cos, cosh, acos, tan, tanh, atan, exp, log, sqrt, c, vector, numeric, rep and matrix. Notably, for the functions: c, vector, numeric, rep and matrix the function is ignored during differentiation.

Value

a S4 class of type *fcts* extended by the new function-derivative pair.

Note

The body of *f* and *f_deriv* have to be defined without curly brackets.

Examples

```
library(dfdr)
# Initialize list
lst <- dfdr::fcts()

# The function which should be added
f <- function(x) x^2
# The dervative function of f
f_deriv <- function(x) 2*x

# add new entry to list
lst <- fcts_add_fct(lst, f, f_deriv)
```

```
g <- function(z) f(z)
df <- d(g, z, lst)
df
```

gradient *Compute the gradient-function of a function.*

Description

Creates a function that computes the derivative of a function with respect to each parameter and return a vector of these.

Usage

```
gradient(f, use_names, ...)
```

Arguments

f	A function
use_names	Should the gradient add variable names to the output of the function?
...	The variable names for which gradients should be calculated

Value

A function that computes the gradient of f at any point.

Examples

```
f <- function(x, y) x^2 + y^2
df <- gradient(f, FALSE, x, y)
df(1, 1)
```

hessian *Compute the Hessian-function of a function.*

Description

Creates a function that computes the second-order derivatives of a function with respect to each pair of parameters and return a vector of these.

Usage

```
hessian(f, use_names = FALSE, ...)
```

Arguments

`f` A function

`use_names` Should the gradient add variable names to the output of the function?

`...` The variable names for which gradients should be calculated

Value

A function that computes the gradient of f at any point.

Examples

```
f <- function(x, y) x**2 + y**2
h <- hessian(f, FALSE, x, y)
h(0, 0)
```

jacobian *jacobian function*

Description

Creates a function that computes the jacobi-matrix of a function for one specific variable. Hereinafter the variable is called *y*. The derivative is calculated with respect to one of the arguments of the function. Subsequently, the variable is called *x*. The returned function can be called at any possible point of *x*.

Usage

```
jacobian(f, y, x, derivs = NULL, num_functions = NULL)
```

Arguments

`f` A function

`y` The variables to compute the derivatives of (the dependent variable). For example: *df/dx*

`x` The variables to which respect the variables are calcualted (the independent variable). For example: *df/dx*

`derivs` optional input defining own functions which should be used. See `d()` for details.

`num_functions` optional input defining number of functions otherwise a squared matrix form is assumed.

Details

The function *jacobian* is intended for using it for functions accepting vectors (in case of *x*) and returns a vector (for *y*).

Mentionable, only integers are allowed for indexing the vectors. Moreover, only one element at the time can be changed. For instance, *y[1]* is permitted. In contrast, *y[1.5]* or *y[variable]* will throw an error.

As usually it is possible to define new variables. If *x* and/or *y* are found at the right side of the assignment operator the variable is replaced in all following lines. See the example below:

```
# Old code
a <- x[1]
b <- 3
y[1] <- a*b
# New code
b <- 3
y[1] <- a*3
```

Furthermore, it is possible to use *if, else if, else* blocks within the function. However, the dependent variable have to be located at the left side of the assignment operator. This restriction is necessary as variables found in previous lines are replaced in the following lines.

```
# allowed code
f <- function(x, t) {
   y <- numeric(2)
   y[1] <- 2*x[1]^3
   if(t < 3) {
      y[2] <- x[2]^2
   } else {
      y[2] <- x[2]^4
   }
  return(y)
}
# not allowed code
f <- function(x, t) {
   y <- numeric(2)
   y[1] <- 2*x[1]^3
   a <- 0
   if(t < 3) {
      a <- x[2]^2
   } else {
      a <- x[2]^4
   }
   y[2] <- a
  return(y)
}
```

Value

A function that computes the jacobi-matrix of f. Notably, it expects the dame arguments as the input function *f*.

Examples

```
f <- function(x) {
   y <- numeric(2)
   y[1] <- x[1]^2 + sin(4)
   y[2] <- x[2]*7
   return(y)
}
jac <- dfdr::jacobian(f, y, x)
jac(c(1, 2))
```

simplify *Simplify an expression by computing the values for constant expressions*

Description

Simplify an expression by computing the values for constant expressions

Usage

```
simplify(expr)
```

Arguments

expr	An expression

Value

a simplified expression

Examples

```
ex <- quote(a*0 + b^2 + 0)
simplify(ex)
```

Index