

Package: fntl (via r-universe)

September 5, 2024

Title Numerical Tools for 'Rcpp' and Lambda Functions

Version 0.1.1

Description Provides a 'C++' API for routinely used numerical tools such as integration, root-finding, and optimization, where function arguments are given as lambdas. This facilitates 'Rcpp' programming, enabling the development of 'R'-like code in 'C++' where functions can be defined on the fly and use variables in the surrounding environment.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

URL <https://github.com/andrewraim/fntl>

Depends R (>= 4.3)

Suggests knitr, numDeriv, rmarkdown, quarto, tidyverse, tinytest

LinkingTo Rcpp

Imports Rcpp

VignetteBuilder quarto

Repository <https://andrewraim.r-universe.dev>

RemoteUrl <https://github.com/andrewraim/fntl>

RemoteRef HEAD

RemoteSha df04a3ea64ffe1ee7508a46ca9148393fb0a1ec1

Contents

fntl-package	2
args	2
deriv	3
findroot	5
gradient0	6

hessian0	6
integrate0	7
jacobian0	8
matrix_apply	8
multivariate-optimization	9
outer	11
solve_cg	13
univariate-optimization	14
which0	14

Index 16

fntl-package	<i>fntl</i>
--------------	-------------

Description

Package documentation

Author(s)

Maintainer: Andrew M. Raim <andrew.raim@gmail.com> ([ORCID](#))

See Also

Useful links:

- <https://github.com/andrewraim/fntl>

args	<i>Arguments</i>
------	------------------

Description

Get an arguments list for internal methods with the default settings. This object can be adjusted and passed to the respective function.

Usage

findroot_args()

optimize_args()

integrate_args()

cg_args()

bfgs_args()

lbfgsb_args()

neldermead_args()

nlm_args()

richardson_args()

Value

An argument list corresponding to the specified function. The elements of the list are named and supplied with default values. See the package vignette for further details.

- `findroot_args` is documented in the section "Root-Finding".
- `optimize_args` is documented in the section "Univariate Optimization".
- `integrate_args` is documented in the section "Integration".
- `cg_args` is documented in the section "Conjugate Gradient".
- `bfgs_args` is documented in the section "BFGS".
- `lbfgsb_args` is documented in the section "L-BFGS-B".
- `neldermead_args` is documented in the section "Nelder-Mead".
- `nlm_args` is documented in the section "Newton-Type Algorithm for Nonlinear Optimization".
- `richardson_args` is documented in the section "Richardson Extrapolated Finite Differences".

deriv

Numerical Derivatives via Finite Differences

Description

Numerical Derivatives via Finite Differences

Usage

`fd_deriv1(f, x, i, h, fd_type)`

`fd_deriv2(f, x, i, j, h_i, h_j, fd_type)`

`deriv1(f, x, i, args, fd_type)`

`deriv2(f, x, i, j, args, fd_type)`

Arguments

f	Function to differentiate.
x	Scalar at which to evaluate the derivative.
i	First coordinate to differentiate.
h	Step size in the first coordinate.
fd_type	Type of derivative: 0 for symmetric difference, 1 for forward difference, and 2 for backward difference.
j	Second coordinate to differentiate.
h_i	Step size in the first coordinate.
h_j	Step size in the second coordinate.
args	List of additional arguments from the function <code>richardson_args</code> .

Value

`fd_deriv1` and `fd_deriv2` return a single numeric value corresponding to the first and second derivative via finite differences. `deriv1` and `deriv2` return a list with the form of a `richardson_result` described in section "Richardson Extrapolated Finite Differences" of the package vignette.

Examples

```
args = richardson_args()

f = sin # Try 2nd derivatives of a univariate function
x0 = 0.5
print(-sin(x0)) ## Exact answer for f'(x0)

fd_deriv2(f, x0, i = 0, j = 0, h_i = 0.001, h_j = 0.001, fd_type = 0)
fd_deriv2(f, x0, i = 0, j = 0, h_i = 0.001, h_j = 0.001, fd_type = 1)
fd_deriv2(f, x0, i = 0, j = 0, h_i = 0.001, h_j = 0.001, fd_type = 2)

deriv2(f, x0, i = 0, j = 0, args, fd_type = 0)

# Try 2nd derivatives of a bivariate function
f = function(x) { sin(x[1]) + cos(x[2]) }
x0 = c(0.5, 0.25)

print(-sin(x0[1])) ## Exact answer for f_xx(x0)
print(-cos(x0[2])) ## Exact answer for f_yy(x0)
print(0)          ## Exact answer for f_xy(x0,y0)

numDeriv::hessian(f, x0)

fd_deriv2(f, x0, i = 0, j = 0, h_i = 0.001, h_j = 0.001, fd_type = 0)
fd_deriv2(f, x0, i = 0, j = 0, h_i = 0.001, h_j = 0.001, fd_type = 1)
fd_deriv2(f, x0, i = 0, j = 0, h_i = 0.001, h_j = 0.001, fd_type = 2)

fd_deriv2(f, x0, i = 0, j = 1, h_i = 0.001, h_j = 0.001, fd_type = 0)
fd_deriv2(f, x0, i = 0, j = 1, h_i = 0.001, h_j = 0.001, fd_type = 1)
```

```

fd_deriv2(f, x0, i = 0, j = 1, h_i = 0.001, h_j = 0.001, fd_type = 2)

fd_deriv2(f, x0, i = 1, j = 1, h_i = 0.001, h_j = 0.001, fd_type = 0)
fd_deriv2(f, x0, i = 1, j = 1, h_i = 0.001, h_j = 0.001, fd_type = 1)
fd_deriv2(f, x0, i = 1, j = 1, h_i = 0.001, h_j = 0.001, fd_type = 2)

deriv2(f, x0, i = 1, j = 1, args, fd_type = 0)
deriv2(f, x0, i = 1, j = 1, args, fd_type = 1)
deriv2(f, x0, i = 1, j = 1, args, fd_type = 2)

```

findroot

Find Root

Description

Find Root

Usage

```
findroot_bisect(f, lower, upper, args)
```

```
findroot_brent(f, lower, upper, args)
```

Arguments

f	Function for which a root is desired.
lower	Lower limit of search interval. Must be finite.
upper	Upper limit of search interval. Must be finite.
args	List of additional arguments from the function findroot_args.

Value

A list with the form of a findroot_result described in section "Root-Finding" of the package vignette.

Examples

```

f = function(x) { x^2 - 1 }
args = findroot_args()
findroot_bisect(f, 0, 10, args)
findroot_brent(f, 0, 10, args)

```

gradient0 *Numerical Gradient Vector*

Description

Numerical Gradient Vector

Usage

```
gradient0(f, x, args)
```

Arguments

f	Function to differentiate.
x	Vector at which to evaluate the gradient.
args	List of additional arguments from the function richardson_args.

Value

A list with the form of a `gradient_result` described in section "Gradient" of the package vignette.

Examples

```
f = function(x) { sum(sin(x)) }  
args = richardson_args()  
x0 = seq(0, 1, length.out = 5)  
cos(x0) ## Exact answer  
gradient0(f, x0, args)  
numDeriv::grad(f, x0)
```

hessian0 *Numerical Hessian*

Description

Numerical Hessian

Usage

```
hessian0(f, x, args)
```

Arguments

f	Function to differentiate.
x	Vector at which to evaluate the Hessian.
args	List of additional arguments from the function richardson_args.

Value

A list with the form of a `hessian_result` described in section "Hessian" of the package vignette.

Examples

```
f = function(x) { sum(x^2) }
x0 = seq(1, 10, length.out = 5)
args = richardson_args()
hessian0(f, x0, args)
numDeriv::hessian(f, x0)
```

integrate0

Integration

Description

Compute the integral $\int_a^b f(x)dx$.

Usage

```
integrate0(f, lower, upper, args)
```

Arguments

<code>f</code>	Function to integrate.
<code>lower</code>	Lower limit of integral.
<code>upper</code>	Upper limit of integral.
<code>args</code>	List of additional arguments from the function <code>integrate_args</code> .

Value

A list with the form of a `integrate_result` described in section "Integration" of the package vignette.

Examples

```
f = function(x) { exp(-x^2 / 2) }
args = integrate_args()
integrate0(f, 0, 10, args)
```

jacobian0

Numerical Jacobian Matrix

Description

Numerical Jacobian Matrix

Usage`jacobian0(f, x, args)`**Arguments**

<code>f</code>	Function to differentiate.
<code>x</code>	Vector at which to evaluate the Jacobian.
<code>args</code>	List of additional arguments from the function <code>richardson_args</code> .

ValueA list with the form of a `jacobian_result` described in section "Jacobian" of the package vignette.**Examples**

```
f = function(x) { cumsum(sin(x)) }
x0 = seq(1, 10, length.out = 5)
args = richardson_args()
out = jacobian0(f, x0, args)
print(out$value)
numDeriv::jacobian(f, x0)
```

matrix_apply*Matrix Apply Functions*

Description

Matrix Apply Functions

Usage`mat_apply(X, f)``row_apply(X, f)``col_apply(X, f)`

Arguments

X	A matrix
f	The function to apply.

Details

The `mat_apply`, `row_apply`, and `col_apply` C++ functions are intended to operate like the following calls in R, respectively.

```
apply(x, c(1,2), f)
apply(x, 1, f)
apply(x, 2, f)
```

The R functions exposed here are specific to numeric-valued matrices, but the underlying C++ functions are intended to work with any type of Rcpp Matrix.

Value

`mat_apply` returns a matrix. `row_apply` and `col_apply` return a vector. See section "Apply" of the package vignette for details.

Examples

```
X = matrix(1:12, nrow = 4, ncol = 3)
mat_apply(X, f = function(x) { x^(1/3) })
row_apply(X, f = function(x) { sum(x^2) })
col_apply(X, f = function(x) { sum(x^2) })
```

multivariate-optimization

Multivariate Optimization

Description

Multivariate Optimization

Usage

```
cg1(init, f, g, args)
cg2(init, f, args)
bfgs1(init, f, g, args)
bfgs2(init, f, args)
```

```
lbfgsb1(init, f, g, args)
```

```
lbfgsb2(init, f, args)
```

```
neldermead(init, f, args)
```

```
nlm1(init, f, g, h, args)
```

```
nlm2(init, f, g, args)
```

```
nlm3(init, f, args)
```

Arguments

<code>init</code>	Initial value
<code>f</code>	Function f to optimize
<code>g</code>	Gradient function of f .
<code>args</code>	List of additional arguments for optimization.
<code>h</code>	Hessian function of f .

Details

The argument `args` should be a list constructed from one of the following functions:

- `bfgs_args` for BFGS;
- `lbfgsb_args` for L-BFGS-B;
- `cg_args` for CG;
- `neldermead_args` for Nelder-Mead;
- `nlm_args` for the Newton-type algorithm used in `nlm`.

When `g` or `h` are omitted, the gradient or Hessian will be respectively be computed via finite differences.

Value

A list with results corresponding to the specified function. See the package vignette for further details.

- `cg1` and `cg2` return a `cg_result` which is documented in the section "Conjugate Gradient".
- `bfgs1` and `bfgs2` return a `bfgs_result` which is documented in the section "BFGS".
- `lbfgsb1` and `lbfgsb2` return a `lbfgsb_result` which is documented in the section "L-BFGS-B".
- `neldermead` returns a `neldermead_result` which is documented in the section "Nelder-Mead".
- `nlm1`, `nlm2`, and `nlm3` return a `nlm_result` which is documented in the section "Newton-Type Algorithm for Nonlinear Optimization".

Examples

```
f = function(x) { sum(x^2) }
g = function(x) { 2*x }
h = function(x) { 2*diag(length(x)) }
x0 = c(1,1)
```

```
args = cg_args()
cg1(x0, f, g, args)
cg2(x0, f, args)
```

```
args = bfgs_args()
bfgs1(x0, f, g, args)
bfgs2(x0, f, args)
```

```
args = lbfgsb_args()
lbfgsb1(x0, f, g, args)
lbfgsb2(x0, f, args)
```

```
args = neldermead_args()
neldermead(x0, f, args)
```

```
args = nlm_args()
nlm1(x0, f, g, h, args)
nlm2(x0, f, g, args)
nlm3(x0, f, args)
```

outer

Outer Matrix

Description

Compute "outer" matrices and matrix-vector products based on a function that operators on pairs of rows. See details.

Usage

```
outer1(X, f)
```

```
outer2(X, Y, f)
```

```
outer1_matvec(X, f, a)
```

```
outer2_matvec(X, Y, f, a)
```

Arguments

X A numerical matrix.

f	Function $f(x, y)$ that operates on a pair of rows. Depending on the context, rows x and y are both rows of X , or x is a row from X and y is a row from Y .
Y	A numerical matrix.
a	A scalar vector.

Details

The `outer1` function computes the $n \times n$ symmetric matrix

$$\text{outer1}(X, f) = \begin{bmatrix} f(x_1, x_1) & \cdots & f(x_1, x_n) \\ \vdots & \ddots & \vdots \\ f(x_n, x_1) & \cdots & f(x_n, x_n) \end{bmatrix}$$

and the `outer1_matvec` operation computes the n -dimensional vector

$$\text{outer1_matvec}(X, f, a) = \begin{bmatrix} f(x_1, x_1) & \cdots & f(x_1, x_n) \\ \vdots & \ddots & \vdots \\ f(x_n, x_1) & \cdots & f(x_n, x_n) \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}.$$

The `outer2` operation computes the $m \times n$ matrix

$$\text{outer2}(X, Y, f) = \begin{bmatrix} f(x_1, y_1) & \cdots & f(x_1, y_n) \\ \vdots & \ddots & \vdots \\ f(x_m, y_1) & \cdots & f(x_m, y_n) \end{bmatrix}$$

and the `outer2_matvec` operation computes the m -dimensional vector

$$\text{outer2_matvec}(X, Y, f, a) = \begin{bmatrix} f(x_1, y_1) & \cdots & f(x_1, y_n) \\ \vdots & \ddots & \vdots \\ f(x_m, y_1) & \cdots & f(x_m, y_n) \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}.$$

Value

`outer1` and `outer2` return a matrix. `outer1_matvec` and `outer2_matvec` return a vector. See section "Outer" of the package vignette for details.

Examples

```
set.seed(1234)
f = function(x,y) { sum( (x - y)^2 ) }
X = matrix(rnorm(12), 6, 2)
Y = matrix(rnorm(10), 5, 2)
outer1(X, f)
outer2(X, Y, f)

a = rep(1, 6)
b = rep(1, 5)
outer1_matvec(X, f, a)
```

```
outer2_matvec(X, Y, f, b)
```

```
solve_cg
```

Iteratively Solve a Linear System with Conjugate Gradient

Description

Solve the system $l(x) = b$ where $l(x)$ is a matrix-free representation of the linear operation Ax .

Usage

```
solve_cg(l, b, init, args)
```

Arguments

<code>l</code>	A linear transformation of x .
<code>b</code>	A vector.
<code>init</code>	Initial value of solution.
<code>args</code>	List of additional arguments from <code>cg_args</code> .

Value

A list with the form of a `solve_cg_result` described in section "Conjugate Gradient" of the package vignette.

Examples

```
set.seed(1234)

n = 8
idx_diag = cbind(1:n, 1:n)
idx_ldiag = cbind(2:n, 1:(n-1))
idx_udiag = cbind(1:(n-1), 2:n)
b = rep(1, n)

## Solution by explicit computation of solve(A, b)
A = matrix(0, n, n)
A[idx_diag] = 2
A[idx_ldiag] = 1
A[idx_udiag] = 1
solve(A, b)

## Solve iteratively with solve_cg
f = function(x) { A %*% x }
args = cg_args()
init = rep(0, n)
solve_cg(f, b, init, args)
```

 univariate-optimization

Univariate Optimization

Description

Univariate Optimization

Usage

```
goldensection(f, lower, upper, args)
```

```
optimize_brent(f, lower, upper, args)
```

Arguments

f	Function to optimize.
lower	Lower limit of search interval. Must be finite.
upper	Upper limit of search interval. Must be finite.
args	List of additional arguments from the function optimize_args.

Value

A list with the form of a `optimize_result` described in section "Univariate Optimization" of the package vignette.

Examples

```
f = function(x) { x^2 - 1 }
args = optimize_args()
goldensection(f, 0, 10, args)
optimize_brent(f, 0, 10, args)
```

 which0

Matrix Which Function

Description

Matrix Which Function

Usage

```
which0(X, f)
```

Arguments

<code>X</code>	A matrix
<code>f</code>	A predicate to apply to each element of <code>X</code> .

Details

The which C++ functions are intended to operate like the following call in R.

```
which(f(X), arr.ind = TRUE) - 1
```

The R functions exposed here are specific to numeric-valued matrices, but the underlying C++ functions are intended to work with any type of Rcpp Matrix.

Value

A matrix with two columns. Each row contains a row and column index corresponding to an element of `X` that matches the criteria of `f`. See section "Which" of the package vignette for details.

Examples

```
X = matrix(1:12 / 6, nrow = 4, ncol = 3)
f = function(x) { x < 1 }
which0(X, f)
```

Index

args, 2

bfgs1 (multivariate-optimization), 9
bfgs2 (multivariate-optimization), 9
bfgs_args (args), 2

cg1 (multivariate-optimization), 9
cg2 (multivariate-optimization), 9
cg_args (args), 2
col_apply (matrix_apply), 8

deriv, 3
deriv1 (deriv), 3
deriv2 (deriv), 3

fd_deriv1 (deriv), 3
fd_deriv2 (deriv), 3
findroot, 5
findroot_args (args), 2
findroot_bisect (findroot), 5
findroot_brent (findroot), 5
fntl (fntl-package), 2
fntl-package, 2

goldensection
 (univariate-optimization), 14
gradient0, 6

hessian0, 6

integrate0, 7
integrate_args (args), 2

jacobian0, 8

lbfgsb1 (multivariate-optimization), 9
lbfgsb2 (multivariate-optimization), 9
lbfgsb_args (args), 2

mat_apply (matrix_apply), 8
matrix_apply, 8

multivariate-optimization, 9

neldermead (multivariate-optimization),
 9
neldermead_args (args), 2
nlm1 (multivariate-optimization), 9
nlm2 (multivariate-optimization), 9
nlm3 (multivariate-optimization), 9
nlm_args (args), 2

optimize_args (args), 2
optimize_brent
 (univariate-optimization), 14

outer, 11
outer1 (outer), 11
outer1_matvec (outer), 11
outer2 (outer), 11
outer2_matvec (outer), 11

richardson_args (args), 2
row_apply (matrix_apply), 8

solve_cg, 13

univariate-optimization, 14

which0, 14