Package ‘fntl’

July 22, 2025

Title Numerical Tools for 'Rcpp' and Lambda Functions

Version 0.1.2

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

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

NeedsCompilation yes

Author Andrew M. Raim [aut, cre] (ORCID: <https://orcid.org/0000-0002-4440-2330>)

Maintainer Andrew M. Raim <andrew.raim@gmail.com>

Repository CRAN

Date/Publication 2024-11-12 19:00:02 UTC

Contents

`fntl-package` *fntl*

Description

Package documentation

Author(s)

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

See Also

Useful links:

- `https://github.com/andrewraim/fntl`

`args` *Arguments*

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 richardson_args.

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

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

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

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

Value

A 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

`init`	Initial value
`f`	Function f to optimize
`g`	Gradient function of f.
`args`	List of additional arguments for optimization.
`h`	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 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

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

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)
```

---

trunc *Functions for Truncated Distributions*

Description

Density, CDF, quantile, and drawing functions for a univariate distribution with density f, cdf, F, and quantile function F^{-} truncated to the interval $[a, b]$.

Usage

```
d_trunc(x, lo, hi, f, F, log = FALSE)

p_trunc(x, lo, hi, F, lower = TRUE, log = FALSE)

q_trunc(p, lo, hi, F, Finv, lower = TRUE, log = FALSE)

r_trunc(n, lo, hi, F, Finv)
```

Arguments

`x`	Vector of quantiles.
`lo`	Vector of lower limits.
`hi`	Vector of upper limits.
`f`	Density function with form `f(x, log)`.

`F`	CDF function with signature `F(x, lower, log)`, where `x` is numeric and `lower` and `log` are logical.
`log`	logical; if `TRUE`, probabilities are given on log-scale.
`lower`	logical; if `TRUE`, probabilities are $P(X \leq x)$; otherwise, $P(X > x)$.
`p`	Vector of probabilities.
`Finv`	Quantile function with signature `Finv(x, lower, log)`, where `x` is numeric and `lower` and `log` are logical.
`n`	Number of draws.

Value

Vector with results. `d_trunc` computes the density, `r_trunc` generates random deviates, `p_trunc` computes the CDF, and `q_trunc` computes quantiles.

Examples

```
library(tidyverse)

m = 100  ## Length of sequence for density, CDF, etc
shape1 = 5
shape2 = 2
lo = 0.5
hi = 0.7

# Density, CDF, and quantile function for untruncated distribution
ff = function(x, log) { dbeta(x, shape1, shape2, log = log) }
FF = function(x, lower, log) { pbeta(x, shape1, shape2, lower.tail = lower, log = log) }
FFinv = function(x, lower, log) { qbeta(x, shape1, shape2, lower.tail = lower, log = log) }

# Compare truncated and untruncated densities
xseq = seq(0, 1, length.out = m)
lo_vec = rep(lo, m)
hi_vec = rep(hi, m)
f0seq = ff(xseq, log = FALSE)
fseq = d_trunc(xseq, lo_vec, hi_vec, ff, FF)
data.frame(x = xseq, f = fseq, f0 = f0seq) %>%
    ggplot() +
    geom_line(aes(xseq, fseq)) +
    geom_line(aes(xseq, f0seq), lty = 2) +
    xlab("x") +
    ylab("Density") +
    theme_minimal()

# Compare truncated densities and empirical density of generated draws
n = 100000
lo_vec = rep(lo, n)
hi_vec = rep(hi, n)
x = r_trunc(n = n, lo_vec, hi_vec, FF, FFinv)
hist(x, probability = TRUE, breaks = 15)
points(xseq, fseq)
```

```
# Compare empirical CDF of draws with CDF function
Femp = ecdf(x)
lo_vec = rep(lo, m)
hi_vec = rep(hi, m)
Fseq = p_trunc(xseq, lo_vec, hi_vec, FF)
data.frame(x = xseq, FF = Fseq) %>%
    mutate(F0 = Femp(x)) %>%
    ggplot() +
    geom_line(aes(xseq, FF), lwd = 1.2) +
    geom_line(aes(xseq, F0), col = "orange") +
    xlab("x") +
    ylab("Probability") +
    theme_minimal()

# Compare empirical quantiles of draws with quantile function
pseq = seq(0, 1, length.out = m)
lo_vec = rep(lo, m)
hi_vec = rep(hi, m)
Finvseq = q_trunc(pseq, lo_vec, hi_vec, FF, FFinv)
Finvemp = quantile(x, prob = pseq)
data.frame(p = pseq, Finv = Finvseq, Finvemp = Finvemp) %>%
    ggplot() +
    geom_line(aes(pseq, Finv), lwd = 1.2) +
    geom_line(aes(pseq, Finvemp), col = "orange") +
    xlab("p") +
    ylab("Quantile") +
    theme_minimal()
```

---

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

`X`	A matrix
`f`	A predicate to apply to each element of X.

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