Package: boostmath (via r-universe)

May 10, 2026

Title 'R' Bindings for the 'Boost' Math Functions

Version 1.4.0.9000

Description 'R' bindings for the various functions and statistical distributions provided by the 'Boost' Math library <https://www.boost.org/doc/libs/latest/libs/math/doc/html/index.html>.

License MIT + file LICENSE

URL https://github.com/andrjohns/boostmath, https://www.boost.org/doc/libs/latest/libs/math/doc/html/index.html, https://andrjohns.github.io/boostmath/

BugReports https://github.com/andrjohns/boostmath/issues

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Depends R (>= 3.0.2)

LinkingTo cpp11, BH (>= 1.90.0)

NeedsCompilation yes

Suggests knitr, rmarkdown, tinytest

VignetteBuilder knitr

Config/build/compilation-database true

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

Date/Publication 2026-05-10 12:41:03 UTC

RemoteUrl https://github.com/andrjohns/boostmath

RemoteRef HEAD

RemoteSha 2d65b18197198675e6c3628f8607dd05340a5a5d

Contents

`airy_functions` *Airy Functions*

Description

Functions to compute the Airy functions Ai and Bi, their derivatives, and their zeros.

The Airy functions are the two linearly independent solutions to the differential equation:

$$y'' - xy = 0$$

Airy Ai Function: The first solution to the Airy differential equation. For negative x values, $Ai(x)$ exhibits oscillatory behavior. For positive x values, $Ai(x)$ monotonically decreases toward zero.

Airy Bi Function: The second solution to the Airy differential equation. For negative x values, $Bi(x)$ exhibits cyclic oscillation. For positive x values, $Bi(x)$ tends toward infinity.

Airy Ai' Function: The derivative of the first solution to the Airy differential equation. For negative x values, $Ai'(x)$ displays cyclic oscillation. For positive x values, $Ai'(x)$ approaches zero asymptotically.

Airy Bi' Function: The derivative of the second solution to the Airy differential equation. For negative x values, $Bi'(x)$ oscillates cyclically. For positive x values, $Bi'(x)$ increases toward infinity.

Zeros of Airy Functions: The zeros are the values where $Ai(x) = 0$ or $Bi(x) = 0$. The zeros are indexed starting from 1. The first few zeros are approximately:

- Ai: -2.33811, -4.08795, -5.52056, ...
- Bi: -1.17371, -3.27109, -4.83074, ...

All functions are implemented using relationships to Bessel functions for numerical accuracy.

Usage

```
airy_ai(x)

airy_bi(x)

airy_ai_prime(x)

airy_bi_prime(x)
```

```
airy_ai_zero(m = NULL, start_index = NULL, number_of_zeros = NULL)

airy_bi_zero(m = NULL, start_index = NULL, number_of_zeros = NULL)
```

Arguments

x	Input numeric value
m	The index of the zero to find (1-based indexing, so m=1 returns the first zero).
start_index	The starting index for the zeros (1-based).
number_of_zeros	The number of zeros to find.

Value

Single numeric value for the Airy functions and their derivatives, or a vector of length number_of_zeros for the multiple zero functions.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
airy_ai(2)
airy_bi(2)
airy_ai_prime(2)
airy_bi_prime(2)
airy_ai_zero(1)
airy_bi_zero(1)
airy_ai_zero(start_index = 1, number_of_zeros = 5)
airy_bi_zero(start_index = 1, number_of_zeros = 5)
```

---

anderson_darling_test *Anderson-Darling Test for Normality*

Description

Performs the Anderson-Darling test for normality by computing the A^2 test statistic:

$$A^2 = n \int_{-\infty}^{\infty} \frac{(F_n(x) - F(x))^2 F'(x)}{F(x)(1 - F(x))} dx$$

The Anderson-Darling test evaluates whether a sample comes from a normal distribution by computing an integral over the empirical cumulative distribution function (ECDF) and comparing it against the normal distribution's CDF.

Interpretation:

- When A^2/n approaches zero as sample size increases, the normality hypothesis is supported
- When A^2/n converges to a positive finite value, the normality hypothesis lacks support

Important: The input data vector `x` must be sorted in ascending order. Unsorted data will trigger an error.

Usage

```
anderson_darling_normality_statistic(x, mu = 0, sd = 1)
```

Arguments

`x`	A numeric vector of sample data (must be sorted in ascending order).
`mu`	The mean of the normal distribution to test against. Default is 0.
`sd`	The standard deviation of the normal distribution to test against. Default is 1.

Value

The Anderson-Darling A^2 test statistic.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Anderson-Darling test for normality with sorted data
x <- sort(rnorm(100))
anderson_darling_normality_statistic(x, 0, 1)
```

`arcsine_distribution` *Arcsine Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the arcsine distribution on the interval $[x_{min}, x_{max}]$.

The arcsine distribution is a U-shaped distribution with infinite density at the endpoints.

For $x_{min} < x < x_{max}$:

The PDF is:

$$f(x; x_{min}, x_{max}) = \frac{1}{\pi\sqrt{(x - x_{min})(x_{max} - x)}}$$

The CDF is:

$$F(x; x_{min}, x_{max}) = \frac{2}{\pi} \arcsin\left(\sqrt{\frac{x - x_{min}}{x_{max} - x_{min}}}\right)$$

The quantile for $0 < p < 1$ is

$$F^{-1}(p; x_{min}, x_{max}) = x_{min} + (x_{max} - x_{min}) \sin^2\left(\frac{\pi p}{2}\right)$$

For the standard distribution on $[0, 1]$, these reduce to

$$f(x) = 1/(\pi\sqrt{x(1-x)})$$

$$F(x) = \frac{2}{\pi} \arcsin(\sqrt{x})$$

Usage

```
arcsine_distribution(x_min = 0, x_max = 1)

arcsine_pdf(x, x_min = 0, x_max = 1)

arcsine_lpdf(x, x_min = 0, x_max = 1)

arcsine_cdf(x, x_min = 0, x_max = 1)

arcsine_lcdf(x, x_min = 0, x_max = 1)

arcsine_quantile(p, x_min = 0, x_max = 1)
```

Arguments

x_min	Minimum value of the distribution (default is 0).
x_max	Maximum value of the distribution (default is 1).
x	Quantile value in $[x_{min}, x_{max}]$.
p	Probability in $[0, 1]$.

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Arcsine distribution with default parameters
dist <- arcsine_distribution()
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
arcsine_pdf(0.5)
arcsine_lpdf(0.5)
arcsine_cdf(0.5)
arcsine_lcdf(0.5)
arcsine_quantile(0.5)
```

---

barycentric_rational *Barycentric Rational Interpolation*

Description

Barycentric rational interpolation is a high-accuracy interpolation method for non-uniformly spaced samples.

Performance and Accuracy:

It requires O(N) time for construction and O(N) time for each evaluation. If the approximation order is d, the error is O(h^(d+1)).

Caveats:

This method is robust but can behave unexpectedly if the sample spacing at the endpoints is much larger than in the center.

Usage

```
barycentric_rational(x, y, order = 3)
```

Arguments

x	Numeric vector of data points (abscissas).
y	Numeric vector of data values (ordinates).
order	Integer representing the approximation order of the interpolator, defaults to 3.

Value

An object of class barycentric_rational_interpolator with methods:

- interpolate(xi): Evaluate the interpolator at point xi.
- prime(xi): Evaluate the derivative of the interpolator at point xi.

See Also

Boost Documentation

Examples

```
x <- c(0, 1, 2, 3)
y <- c(1, 2, 0, 2)
order <- 3
interpolator <- barycentric_rational(x, y, order)
xi <- 1.5
interpolator$interpolate(xi)
interpolator$prime(xi)
```

---

basic_functions *Basic Mathematical Functions*

Description

High-precision implementations of basic mathematical functions with enhanced numerical stability for special cases.

These functions provide numerically stable alternatives to standard operations, particularly useful when working with values near zero or when high precision is required.

Trigonometric Functions with π:

- sin_pi(x): Computes $\sin(\pi x)$
- cos_pi(x): Computes $\cos(\pi x)$

Logarithmic and Exponential Functions:

- log1p_boost(x): Computes $\log(1 + x)$ accurately for small $|x|$
- expm1_boost(x): Computes $\exp(x) - 1$ accurately for small $|x|$

Root Functions:

- `cbrt(x)`: Computes the cube root $x^{1/3}$
- `sqrt1pm1(x)`: Computes $\sqrt{1+x} - 1$ accurately for small $|x|$
- `rsqrt(x)`: Computes the reciprocal square root $\frac{1}{\sqrt{x}}$

Power Functions:

- `powm1(x, y)`: Computes $x^y - 1$ accurately

Geometric Functions:

- `hypot(x, y)`: Computes $\sqrt{x^2 + y^2}$ without overflow/underflow

Usage

```
sin_pi(x)

cos_pi(x)

log1p_boost(x)

expm1_boost(x)

cbrt(x)

sqrt1pm1(x)

powm1(x, y)

hypot(x, y)

rsqrt(x)
```

Arguments

`x`	Input numeric value
`y`	Second input numeric value (for power and hypotenuse functions)

Value

A single numeric value with the computed result of the function.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# sin(pi/2) = 1 (exact)
sin_pi(0.5)
# cos(pi/2) = 0 (exact)
cos_pi(0.5)
# log(1 + x) for small x
log1p_boost(0.001)
# exp(x) - 1 for small x
expm1_boost(0.001)
# Cube root
cbrt(8)  # Returns 2
# sqrt(1 + x) - 1 for small x
sqrt1pm1(0.001)
# x^y - 1 accurately
powm1(2, 3)  # Returns 7 (2^3 - 1)
# Euclidean distance
hypot(3, 4)  # Returns 5
# Reciprocal square root
rsqrt(4)  # Returns 0.5
```

`bernoulli_distribution`

Bernoulli Distribution Functions

Description

Functions to compute the probability mass function (pmf), cumulative distribution function, and quantile function for the Bernoulli distribution.

The Bernoulli distribution models a single trial with outcomes $k \in \{0, 1\}$ and success probability p:

$$P(X = 0) = 1 - p, \quad P(X = 1) = p$$
$$F(0) = 1 - p, \quad F(1) = 1$$

Usage

```
bernoulli_distribution(p_success)

bernoulli_pdf(x, p_success)

bernoulli_lpdf(x, p_success)

bernoulli_cdf(x, p_success)

bernoulli_lcdf(x, p_success)

bernoulli_quantile(p, p_success)
```

Arguments

p_success	Probability of success (0 <= p_success <= 1).
x	Quantile value (must be 0 or 1).
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Bernoulli distribution with p_success = 0.5
dist <- bernoulli_distribution(0.5)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
bernoulli_pdf(1, 0.5)
bernoulli_lpdf(1, 0.5)
bernoulli_cdf(1, 0.5)
bernoulli_lcdf(1, 0.5)
bernoulli_quantile(0.5, 0.5)
```

`bessel_functions` *Bessel Functions, Their Derivatives, and Zeros*

Description

Functions to compute Bessel functions of the first and second kind, their modified versions, spherical Bessel functions, and their derivatives and zeros.

Bessel functions are solutions to Bessel's ordinary differential equation and appear in many problems with cylindrical or spherical symmetry, such as wave propagation, heat conduction, and quantum mechanics.

Bessel Functions of the First and Second Kinds

- `cyl_bessel_j(v, x)`: Computes the Bessel function of the first kind $J_v(x)$:

$$J_v(x) = \left(\frac{1}{2}x\right)^v \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}x^2\right)^k}{k!\Gamma(v+k+1)}$$

- `cyl_neumann(v, x)`: Computes the Bessel function of the second kind $Y_v(x) = N_v(x)$:

$$Y_v(x) = \frac{J_v(x)\cos(v\pi) - J_{-v}(x)}{\sin(v\pi)}$$

Modified Bessel Functions of the First and Second Kinds

- `cyl_bessel_i(v, x)`: Computes the modified Bessel function of the first kind $I_v(x)$:

$$I_v(x) = \left(\frac{1}{2}x\right)^v \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}x^2\right)^k}{k!\Gamma(v+k+1)}$$

- `cyl_bessel_k(v, x)`: Computes the modified Bessel function of the second kind $K_v(x)$:

$$K_v(x) = \frac{\pi}{2}\frac{I_{-v}(x) - I_v(x)}{\sin(v\pi)}$$

Spherical Bessel Functions of the First and Second Kinds

- `sph_bessel(v, x)`: Computes the Spherical Bessel function of the first kind $j_v(x)$:

$$j_v(x) = \sqrt{\frac{\pi}{2x}} J_{v+\frac{1}{2}}(x)$$

- `sph_neumann(v, x)`: Computes the Spherical Bessel function of the second kind $y_v(x) = n_v(x)$:

$$y_v(x) = \sqrt{\frac{\pi}{2x}} Y_{v+\frac{1}{2}}(x)$$

Derivatives:

The _prime functions compute the derivatives with respect to x of the corresponding Bessel functions:

$$J'_v(x) = \frac{J_{v-1}(x) - J_{v+1}(x)}{2}$$
$$Y'_v(x) = \frac{Y_{v-1}(x) - Y_{v+1}(x)}{2}$$
$$I'_v(x) = \frac{I_{v-1}(x) - I_{v+1}(x)}{2}$$
$$K'_v(x) = \frac{K_{v-1}(x) - K_{v+1}(x)}{-2}$$
$$j'_v(x) = \left(\frac{v}{x}\right) j_v(x) - j_{v+1}(x)$$
$$y'_v(x) = \left(\frac{v}{x}\right) y_v(x) - y_{v+1}(x)$$

Zeros:

The zero functions find the zeros of J_v and Y_v (where the function equals zero), indexed starting from 1.

Usage

```
cyl_bessel_j(v, x)

cyl_neumann(v, x)

cyl_bessel_j_zero(v, m = NULL, start_index = NULL, number_of_zeros = NULL)

cyl_neumann_zero(v, m = NULL, start_index = NULL, number_of_zeros = NULL)

cyl_bessel_i(v, x)

cyl_bessel_k(v, x)

sph_bessel(v, x)

sph_neumann(v, x)

cyl_bessel_j_prime(v, x)

cyl_neumann_prime(v, x)

cyl_bessel_i_prime(v, x)
```

```
cyl_bessel_k_prime(v, x)

sph_bessel_prime(v, x)

sph_neumann_prime(v, x)
```

Arguments

v	Order of the Bessel function (can be any real number)
x	Argument of the Bessel function
m	The index of the zero to find (1-based).
start_index	The starting index for the zeros (1-based).
number_of_zeros	The number of zeros to find.

Value

Single numeric value for the Bessel functions and their derivatives, or a vector of length number_of_zeros for the multiple zero functions.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Bessel function of the first kind J_0(1)
cyl_bessel_j(0, 1)
# Bessel function of the second kind Y_0(1)
cyl_neumann(0, 1)
# Modified Bessel function of the first kind I_0(1)
cyl_bessel_i(0, 1)
# Modified Bessel function of the second kind K_0(1)
cyl_bessel_k(0, 1)
# Spherical Bessel function of the first kind j_0(1)
sph_bessel(0, 1)
# Spherical Bessel function of the second kind y_0(1)
sph_neumann(0, 1)
# Derivative of the Bessel function of the first kind J_0(1)
cyl_bessel_j_prime(0, 1)
# Derivative of the Bessel function of the second kind Y_0(1)
cyl_neumann_prime(0, 1)
# Derivative of the modified Bessel function of the first kind I_0(1)
cyl_bessel_i_prime(0, 1)
# Derivative of the modified Bessel function of the second kind K_0(1)
cyl_bessel_k_prime(0, 1)
# Derivative of the spherical Bessel function of the first kind j_0(1)
sph_bessel_prime(0, 1)
# Derivative of the spherical Bessel function of the second kind y_0(1)
```

```
sph_neumann_prime(0, 1)
# Finding the first zero of the Bessel function of the first kind J_0
cyl_bessel_j_zero(0, 1)
# Finding the first zero of the Bessel function of the second kind Y_0
cyl_neumann_zero(0, 1)
# Finding multiple zeros of the Bessel function of the first kind J_0 starting from index 1
cyl_bessel_j_zero(0, start_index = 1, number_of_zeros = 5)
# Finding multiple zeros of the Bessel function of the second kind Y_0 starting from index 1
cyl_neumann_zero(0, start_index = 1, number_of_zeros = 5)
```

`beta_distribution` *Beta Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, quantile function, and parameter estimators for the Beta distribution.

The Beta distribution is defined on $x \in [0, 1]$ with shape parameters $\alpha > 0$ and $\beta > 0$.

The PDF is:

$$f(x; \alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

Where $B(\alpha, \beta)$ is the beta function.

The CDF is:

$$F(x; \alpha, \beta) = I_x(\alpha, \beta)$$

Where I_x is the regularised incomplete beta function.

The quantile is:

$$F^{-1}(p; \alpha, \beta) = I_p^{-1}(\alpha, \beta)$$

Where I_p^{-1} is the inverse of the regularised incomplete beta function.

Usage

```
beta_distribution(alpha, beta)

beta_pdf(x, alpha, beta)

beta_lpdf(x, alpha, beta)

beta_cdf(x, alpha, beta)

beta_lcdf(x, alpha, beta)
```

```
beta_quantile(p, alpha, beta)

beta_find_alpha(mean = NULL, variance = NULL, beta = NULL, x = NULL, p = NULL)

beta_find_beta(mean = NULL, variance = NULL, alpha = NULL, x = NULL, p = NULL)
```

Arguments

alpha	Shape parameter (alpha > 0).
beta	Shape parameter (beta > 0).
x	Quantile value (0 <= x <= 1).
p	Probability (0 <= p <= 1).
mean	Mean of the Beta distribution.
variance	Variance of the Beta distribution.

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Beta distribution with shape parameters alpha = 2, beta = 5
dist <- beta_distribution(2, 5)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
beta_pdf(0.5, 2, 5)
```

```
beta_lpdf(0.5, 2, 5)
beta_cdf(0.5, 2, 5)
beta_lcdf(0.5, 2, 5)
beta_quantile(0.5, 2, 5)

## Not run:
# Find alpha given mean and variance
beta_find_alpha(mean = 0.3, variance = 0.02)
# Find alpha given beta, x, and probability
beta_find_alpha(beta = 5, x = 0.4, p = 0.6)
# Find beta given mean and variance
beta_find_beta(mean = 0.3, variance = 0.02)
# Find beta given alpha, x, and probability
beta_find_beta(alpha = 2, x = 0.4, p = 0.6)

## End(Not run)
```

`beta_functions` *Beta Functions*

Description

Functions to compute the Beta function, normalised incomplete beta function, and their complements, as well as their inverses and derivatives.

Beta Function $B(a, b)$:

- `beta_boost(a, b)`

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

Incomplete Beta Functions:

- **Normalised (Regularised) Functions:**
  - `ibeta(a, b, x)`: Normalised incomplete beta function $I_x(a, b)$

$$I_x(a,b) = \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

- `ibetac(a, b, x)`: Normalised complement, $1 - I_x(a, b) = I_{1-x}(b, a)$
- **Non-normalised Functions:**
  - `beta_boost(a, b, x)`: Full incomplete beta function $B_x(a, b)$

$$\int_0^x t^{a-1}(1-t)^{b-1} dt$$

- `betac(a, b, x)`: Full complement , $1 - B_x(a, b) = B_{1-x}(b, a)$

Inverse Functions:

- **Primary inverses (solving for x):**
  - `ibeta_inv(a, b, p)`: Returns x such that $p = I_x(a, b)$
  - `ibetac_inv(a, b, q)`: Returns x such that $q = 1 - I_x(a, b)$
- **Parameter inverses (solving for a or b):**
  - `ibeta_inva(b, x, p)`: Returns a such that $p = I_x(a, b)$
  - `ibetac_inva(b, x, q)`: Returns a such that $q = 1 - I_x(a, b)$
  - `ibeta_invb(a, x, p)`: Returns b such that $p = I_x(a, b)$
  - `ibetac_invb(a, x, q)`: Returns b such that $q = 1 - I_x(a, b)$s

Derivatives:

`ibeta_derivative(a, b, x)`: Computes the partial derivative with respect to x of the incomplete beta function

$$\frac{\partial}{\partial x} I_x(a, b) = \frac{(1-x)^{b-1} x^{a-1}}{B(a, b)}$$

Usage

```
beta_boost(a, b, x = NULL)

ibeta(a, b, x)

ibetac(a, b, x)

betac(a, b, x)

ibeta_inv(a, b, p)

ibetac_inv(a, b, q)

ibeta_inva(b, x, p)

ibetac_inva(b, x, q)

ibeta_invb(a, x, p)

ibetac_invb(a, x, q)

ibeta_derivative(a, b, x)
```

Arguments

a	First parameter of the beta function (must be positive)
b	Second parameter of the beta function (must be positive)
x	Upper limit of integration (0 <= x <= 1)

p	Probability value (0 <= p <= 1)
q	Probability value (0 <= q <= 1), where q = 1 - p

Value

A single numeric value with the computed beta function, normalised incomplete beta function, or their complements, depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
## Not run:
# Euler beta function B(2, 3)
beta_boost(2, 3)
# Normalised incomplete beta function I_x(2, 3) for x = 0.5
ibeta(2, 3, 0.5)
# Normalised complement of the incomplete beta function 1 - I_x(2, 3) for x = 0.5
ibetac(2, 3, 0.5)
# Full incomplete beta function B_x(2, 3) for x = 0.5
beta_boost(2, 3, 0.5)
# Full complement of the incomplete beta function 1 - B_x(2, 3) for x = 0.5
betac(2, 3, 0.5)
# Inverse of the normalised incomplete beta function I_x(2, 3) = 0.5
ibeta_inv(2, 3, 0.5)
# Inverse of the normalised complement of the incomplete beta function I_x(2, 3) = 0.5
ibetac_inv(2, 3, 0.5)
# Inverse of the normalised complement of the incomplete beta function I_x(a, b)
# with respect to a for x = 0.5 and q = 0.5
ibetac_inva(3, 0.5, 0.5)
# Inverse of the normalised incomplete beta function I_x(a, b)
# with respect to b for x = 0.5 and p = 0.5
ibeta_invb(0.8, 0.5, 0.5)
# Inverse of the normalised complement of the incomplete beta function I_x(a, b)
# with respect to b for x = 0.5 and q = 0.5
ibetac_invb(2, 0.5, 0.5)
# Derivative of the incomplete beta function with respect to x for a = 2, b = 3, x = 0.5
ibeta_derivative(2, 3, 0.5)

## End(Not run)
```

`bezier_polynomial` *Bezier Polynomial Interpolator*

Description

Bezier polynomials are smooth curves which approximate a set of control points. They are commonly used in computer-aided geometric design.

Properties:

The curve is approximating, meaning it does not necessarily pass through the control points. Passing n control points creates a polynomial of degree n-1. Evaluation is O(N^2) via de Casteljau's algorithm.

Usage

```
bezier_polynomial(control_points)
```

Arguments

`control_points`	List of control points, where each element is a numeric vector of length 3.

Value

An object of class `bezier_polynomial` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `edit_control_point(new_control_point, index)`: Insert a new control point at the specified index.

See Also

Boost Documentation

Examples

```
control_points <- list(c(0, 0, 0), c(1, 2, 0), c(2, 0, 0), c(3, 3, 0))
interpolator <- bezier_polynomial(control_points)
xi <- 1.5
interpolator$interpolate(xi)
interpolator$prime(xi)
new_control_point <- c(1.5, 1, 0)
interpolator$edit_control_point(new_control_point, 2)
```

`bilinear_uniform` *Bilinear Uniform Interpolator*

Description

The bilinear uniform interpolator takes a grid of data points specified by a linear index and interpolates between each segment using a bilinear function.

Details:

"Bilinear" means it is the product of two linear functions. The interpolant is continuous and its evaluation is constant time. The interpolator is point-centered.

Usage

```
bilinear_uniform(x, rows, cols, dx = 1, dy = 1, x0 = 0, y0 = 0)
```

Arguments

`x`	Numeric vector of all grid elements
`rows`	Integer representing the number of rows in the grid
`cols`	Integer representing the number of columns in the grid
`dx`	Numeric value representing the spacing between grid points in the x-direction, defaults to 1
`dy`	Numeric value representing the spacing between grid points in the y-direction, defaults to 1
`x0`	Numeric value representing the x-coordinate of the origin, defaults to 0
`y0`	Numeric value representing the y-coordinate of the origin, defaults to 0

Value

An object of class `bilinear_uniform` with methods:

- `interpolate(xi, yi)`: Evaluate the interpolator at point `(xi, yi)`.

See Also

Boost Documentation

Examples

```
x <- seq(0, 1, length.out = 10)
interpolator <- bilinear_uniform(x, rows = 2, cols = 5)
xi <- 0.5
yi <- 0.5
interpolator$interpolate(xi, yi)
```

`binomial_distribution` *Binomial Distribution Functions*

Description

Functions to compute the probability mass function (pmf), cumulative distribution function, quantile function, and confidence bounds for the Binomial distribution.

The Binomial distribution models the number of successes k in n independent trials with success probability p. The pmf is

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Accuracy and Implementation Notes: CDF and related functions are implemented using incomplete beta functions (`ibeta`, `ibetac`). The pmf is evaluated via `ibeta_derivative` for stability. Quantiles are obtained numerically (TOMS 748), since no closed-form inverse exists for discrete k. As a discrete distribution, quantiles are rounded outward to ensure at least the requested coverage; use complements for improved tail accuracy.

Confidence Bounds: `binomial_find_lower_bound_on_p` and `binomial_find_upper_bound_on_p` implement Clopper-Pearson exact intervals (default) or Jeffreys prior intervals, as described in the Boost documentation.

Usage

```
binomial_distribution(n, prob)

binomial_pdf(k, n, prob)

binomial_lpdf(k, n, prob)

binomial_cdf(k, n, prob)

binomial_lcdf(k, n, prob)

binomial_quantile(p, n, prob)

binomial_find_lower_bound_on_p(n, k, alpha, method = "clopper_pearson_exact")

binomial_find_upper_bound_on_p(n, k, alpha, method = "clopper_pearson_exact")

binomial_find_minimum_number_of_trials(k, prob, alpha)

binomial_find_maximum_number_of_trials(k, prob, alpha)
```

Arguments

n	Number of trials (n >= 0).
prob	Probability of success on each trial (0 <= prob <= 1).
k	Number of successes (0 <= k <= n).
p	Probability (0 <= p <= 1).
alpha	Largest acceptable probability that the true value of the success fraction is less than the value returned (by binomial_find_lower_bound_on_p) or greater than the value returned (by binomial_find_upper_bound_on_p).
method	Method to use for calculating the confidence bounds. Options are "clopper_pearson_exact" (default) and "jeffreys_prior".

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Binomial distribution with n = 10, prob = 0.5
dist <- binomial_distribution(10, 0.5)
# Apply generic functions
cdf(dist, 2)
logcdf(dist, 2)
pdf(dist, 2)
logpdf(dist, 2)
hazard(dist, 2)
chf(dist, 2)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
binomial_pdf(3, 10, 0.5)
binomial_lpdf(3, 10, 0.5)
binomial_cdf(3, 10, 0.5)
binomial_lcdf(3, 10, 0.5)
binomial_quantile(0.5, 10, 0.5)
```

```
## Not run:
# Find lower bound on p given k = 3 successes in n = 10 trials with 95% confidence
binomial_find_lower_bound_on_p(10, 3, 0.05)
# Find upper bound on p given k = 3 successes in n = 10 trials with 95% confidence
binomial_find_upper_bound_on_p(10, 3, 0.05)
# Find minimum number of trials n to observe k = 3 successes with p = 0.5 at 95% confidence
binomial_find_minimum_number_of_trials(3, 0.5, 0.05)
# Find maximum number of trials n to observe k = 3 successes with p = 0.5 at 95% confidence
binomial_find_maximum_number_of_trials(3, 0.5, 0.05)

## End(Not run)
```

`bivariate_statistics` *Bivariate Statistics Functions*

Description

Functions to compute bivariate statistics including covariance and the Pearson correlation coefficient.

Covariance: The population covariance is

$$\mathrm{cov}(x, y) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$

Correlation Coefficient: The Pearson correlation coefficient is

$$\rho_{x,y} = \frac{\mathrm{cov}(x, y)}{\sigma_x \sigma_y}$$

Usage

```
covariance(x, y)

means_and_covariance(x, y)

correlation_coefficient(x, y)
```

Arguments

x	A numeric vector.
y	A numeric vector.

Value

A numeric value (or tuple for `means_and_covariance`) with the computed statistic.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Covariance
covariance(c(1, 2, 3), c(4, 5, 6))
# Means and Covariance
means_and_covariance(c(1, 2, 3), c(4, 5, 6))
# Correlation Coefficient
correlation_coefficient(c(1, 2, 3), c(4, 5, 6))
```

`cardinal_cubic_b_spline`

Cardinal Cubic B-Spline Interpolator

Description

The cardinal cubic B-spline interpolator allows for fast and accurate interpolation of a function which is known at equally spaced points.

Mathematical Properties:

It uses compactly supported basis functions constructed via iterative convolution, ensuring numerical stability. The interpolant is O(h^4) accurate for values and O(h^3) accurate for derivatives, where h is the step size.

Conditions:

Ideally, the function being interpolated should be four-times continuously differentiable. If the derivatives at the endpoints are not provided, they are estimated using one-sided finite-difference formulas.

Usage

```
cardinal_cubic_b_spline(
  y,
  t0,
  h,
  left_endpoint_derivative = NULL,
  right_endpoint_derivative = NULL
)
```

Arguments

`y`	Numeric vector of data points to interpolate.
`t0`	Numeric scalar representing the starting point of the data.
`h`	Numeric scalar representing the spacing between data points.

`left_endpoint_derivative`
: Optional numeric scalar for the derivative at the left endpoint.

`right_endpoint_derivative`
: Optional numeric scalar for the derivative at the right endpoint.

Value

An object of class `cardinal_cubic_b_spline` with methods:

- `interpolate(x)`: Evaluate the spline at point `x`.
- `prime(x)`: Evaluate the first derivative of the spline at point `x`.
- `double_prime(x)`: Evaluate the second derivative of the spline at point `x`.

See Also

Boost Documentation

Examples

```
y <- c(1, 2, 0, 2, 1)
t0 <- 0
h <- 1
spline_obj <- cardinal_cubic_b_spline(y, t0, h)
x <- 0.5
spline_obj$interpolate(x)
spline_obj$prime(x)
spline_obj$double_prime(x)
```

`cardinal_cubic_hermite` *Cardinal Cubic Hermite Interpolator*

Description

The cardinal cubic Hermite interpolator is similar to the cubic Hermite interpolator but optimised for equispaced data.

Performance:

This allows for constant-time evaluation.

Usage

```
cardinal_cubic_hermite(y, dydx, x0, dx)
```

Arguments

`y`	Numeric vector of ordinates (y-coordinates).
`dydx`	Numeric vector of derivatives (slopes) at each point.
`x0`	Numeric value of the first abscissa (x-coordinate).
`dx`	Numeric value of the spacing between abscissas.

Value

An object of class `cardinal_cubic_hermite` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `domain()`: Get the domain of the interpolator.

See Also

Boost Documentation

Examples

```
y <- c(0, 1, 0)
dydx <- c(1, 0, -1)
interpolator <- cardinal_cubic_hermite(y, dydx, 0, 1)
xi <- 0.5
interpolator$interpolate(xi)
interpolator$prime(xi)
interpolator$domain()
```

`cardinal_quadratic_b_spline`

Cardinal Quadratic B-Spline Interpolator

Description

The cardinal quadratic B-spline interpolator is very nearly the same as the cubic B-spline interpolator, but uses quadratic basis functions.

Use Cases:

Basis functions are constructed by convolving a box function with itself twice. Since the basis functions are less smooth than the cubic B-spline, this is primarily useful for approximating functions of reduced smoothness. It is appropriate for functions which are two or three times continuously differentiable.

Usage

```
cardinal_quadratic_b_spline(
  y,
  t0,
  h,
  left_endpoint_derivative = NULL,
  right_endpoint_derivative = NULL
)
```

Arguments

`y`	Numeric vector of data points to interpolate.
`t0`	Numeric scalar representing the starting point of the data.
`h`	Numeric scalar representing the spacing between data points.
`left_endpoint_derivative`	Optional numeric scalar for the derivative at the left endpoint.
`right_endpoint_derivative`	Optional numeric scalar for the derivative at the right endpoint.

Value

An object of class `cardinal_quadratic_b_spline` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.

See Also

Boost Documentation

Examples

```
y <- c(0, 1, 0, 1)
t0 <- 0
h <- 1
interpolator <- cardinal_quadratic_b_spline(y, t0, h)
xi <- 0.5
interpolator$interpolate(xi)
interpolator$prime(xi)
```

`cardinal_quintic_b_spline` *Cardinal Quintic B-Spline Interpolator*

Description

The cardinal quintic B-spline interpolator is similar to the cubic B-spline but uses basis functions constructed by convolving a box function with itself five times.

Properties:

The basis functions are more smooth (C4) than the cubic B-spline (C2), making this useful for computing second derivatives. The second derivative of the quintic B-spline is a cubic spline.

Usage

```
cardinal_quintic_b_spline(
  y,
  t0,
  h,
  left_endpoint_derivatives = NULL,
  right_endpoint_derivatives = NULL
)
```

Arguments

`y`	Numeric vector of data points to interpolate.
`t0`	Numeric scalar representing the starting point of the data.
`h`	Numeric scalar representing the spacing between data points.
`left_endpoint_derivatives`	Optional two-element numeric vector for the derivative at the left endpoint.
`right_endpoint_derivatives`	Optional two-element numeric vector for the derivative at the right endpoint.

Value

An object of class `cardinal_quintic_b_spline` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `double_prime(xi)`: Evaluate the second derivative of the interpolator at point `xi`.

See Also

Boost Documentation

Examples

```
y <- seq(0, 1, length.out = 20)
t0 <- 0
h <- 1
interpolator <- cardinal_quintic_b_spline(y, t0, h)
xi <- 0.5
interpolator$interpolate(xi)
interpolator$prime(xi)
interpolator$double_prime(xi)
```

`cardinal_quintic_hermite` *Cardinal Quintic Hermite Interpolator*

Description

The cardinal quintic Hermite interpolator is similar to the quintic Hermite interpolator but optimised for equispaced data.

Performance:

This allows for constant-time evaluation.

Usage

```
cardinal_quintic_hermite(y, dydx, d2ydx2, x0, dx)
```

Arguments

`y`	Numeric vector of ordinates (y-coordinates).
`dydx`	Numeric vector of first derivatives (slopes) at each point.
`d2ydx2`	Numeric vector of second derivatives at each point.
`x0`	Numeric value of the first abscissa (x-coordinate).
`dx`	Numeric value of the spacing between abscissas.

Value

An object of class `cardinal_quintic_hermite` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `double_prime(xi)`: Evaluate the second derivative of the interpolator at point `xi`.
- `domain()`: Get the domain of the interpolator.

See Also

Boost Documentation

Examples

```
y <- c(0, 1, 0)
dydx <- c(1, 0, -1)
d2ydx2 <- c(0, -1, 0)
x0 <- 0
dx <- 1
interpolator <- cardinal_quintic_hermite(y, dydx, d2ydx2, x0, dx)
xi <- 0.5
interpolator$interpolate(xi)
```

```
interpolator$prime(xi)
interpolator$double_prime(xi)
interpolator$domain()
```

`catmull_rom` *Catmull-Rom Interpolation*

Description

Catmull-Rom splines are a family of interpolating curves which are commonly used in computer graphics and animation.

Properties:

They enjoy affine invariance, local support, C2-smoothness, and interpolation of control points. The curve is internally closed, however the user specifies if it should be treated as open or closed via the parameters. Evaluation is O(log N).

Usage

```
catmull_rom(control_points, closed = FALSE, alpha = 0.5)
```

Arguments

`control_points`	List of control points, where each element is a numeric vector of length 3.
`closed`	Logical indicating whether the spline is closed (i.e., the first and last control points are connected), defaults to false
`alpha`	Numeric scalar for the tension parameter, defaults to 0.5

Value

An object of class `catmull_rom` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `max_parameter()`: Get the maximum parameter value of the spline.
- `parameter_at_point(i)`: Get the parameter value at index `i`.

See Also

Boost Documentation

Examples

```
control_points <- list(c(0, 0, 0), c(1, 1, 0), c(2, 0, 0), c(3, 1, 0))
interpolator <- catmull_rom(control_points)
xi <- 1.5
interpolator$interpolate(xi)
interpolator$prime(xi)
interpolator$max_parameter()
interpolator$parameter_at_point(2)
```

`cauchy_distribution` *Cauchy Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Cauchy distribution.

The PDF is:

$$f(x; x_0, \gamma) = \frac{1}{\pi}\left(\frac{\gamma}{(x - x_0)^2 + \gamma^2}\right)$$

The CDF:

$$F(x; x_0, \gamma) = \frac{1}{\pi}\arctan\left(\frac{x - x_0}{\gamma}\right) + \frac{1}{2}$$

Usage

```
cauchy_distribution(location = 0, scale = 1)

cauchy_pdf(x, location = 0, scale = 1)

cauchy_lpdf(x, location = 0, scale = 1)

cauchy_cdf(x, location = 0, scale = 1)

cauchy_lcdf(x, location = 0, scale = 1)

cauchy_quantile(p, location = 0, scale = 1)
```

Arguments

`location`	location parameter (default is 0)
`scale`	scale parameter (default is 1)
`x`	quantile
`p`	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Cauchy distribution with location = 0, scale = 1
dist <- cauchy_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
support(dist)

# Convenience functions
cauchy_pdf(0)
cauchy_lpdf(0)
cauchy_cdf(0)
cauchy_lcdf(0)
cauchy_quantile(0.5)
```

---

chatterjee_correlation

Chatterjee Correlation

Description

Computes the Chatterjee correlation coefficient, a robust measure of dependence. Unlike classical correlation coefficients (Pearson, Spearman), Chatterjee's coefficient measures the degree to which y is a function of x (functional dependence), capturing non-linear relationships.

Characteristics:

- **Functional Dependence:** Value is 1 if and only if y is a measurable function of x.
- **Independence:** Value is 0 if x and y are independent.
- **Range:** The coefficient is theoretically in $[0, 1]$.
- **Asymmetry:** The measure is asymmetric; $C(X, Y) \neq C(Y, X)$. It specifically tests if $Y = f(X)$.

Usage

```
chatterjee_correlation(x, y)
```

Arguments

x	A numeric vector (the predictor/independent variable).
y	A numeric vector (the response/dependent variable).

Details

The coefficient is calculated using the ranks of y when sorted by x. This implementation computes the sample version of the coefficient as described by Chatterjee (2021).

Formula: Given pairs (X_i, Y_i), sort them such that $X_{(1)} \leq \cdots \leq X_{(n)}$. Let r_i be the rank of $Y_{(i)}$. The coefficient is:

$$\xi_n(X, Y) = 1 - \frac{3 \sum_{i=1}^{n-1} |r_{i+1} - r_i|}{n^2 - 1}$$

Value

A numeric value representing the Chatterjee correlation coefficient.

A numeric vector containing:

1. **Correlation Coefficient:** The Chatterjee correlation estimate.

References

Chatterjee, S. (2021). A new coefficient of correlation. Journal of the American Statistical Association, 116(536), 2009-2022.

See Also

Boost Documentation

Examples

```
# Functional dependence (Y = X^2)
x <- runif(50, -1, 1)
y <- x^2
chatterjee_correlation(x, y) # Should be high (near 1)

# Independence
x <- runif(50)
y <- runif(50)
chatterjee_correlation(x, y) # Should be low (near 0)

# Asymmetry check
chatterjee_correlation(x, y)
chatterjee_correlation(y, x)
```

---

chebyshev_polynomials *Chebyshev Polynomials and Related Functions*

Description

Functions to compute Chebyshev polynomials of the first and second kind, and efficiently evaluate Chebyshev series using Clenshaw's recurrence algorithm.

Chebyshev polynomials are orthogonal polynomials used extensively in approximation theory, numerical analysis, and spectral methods. They minimize the Runge phenomenon in polynomial interpolation.

Chebyshev Polynomials of the First Kind T_n(x):

- `chebyshev_t(n, x)`: Evaluates $T_n(x)$
- Recurrence relation: $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$ for n > 0
- Initial conditions: T_0(x) = 1, T_1(x) = x
- `chebyshev_t_prime(n, x)`: Derivative of T_n(x)
- Stable evaluation for x in $[-1, 1]$ (mixed forward-backward stable)

Chebyshev Polynomials of the Second Kind U_n(x):

- `chebyshev_u(n, x)`: Evaluates U_n(x)
- Related to T_n by differentiation

Recurrence Relation:

- `chebyshev_next(x, Tn, Tn_1)`: Computes $T_{n+1}(x)$ from T_n and T_{n-1}

Clenshaw's Recurrence Algorithm:

Efficient O(n) evaluation of Chebyshev series (alternative to O(n^2) direct computation):

- `chebyshev_clenshaw_recurrence(c, x)`: Evaluates Chebyshev series with coefficients c at point x on standard interval $[-1, 1]$
- `chebyshev_clenshaw_recurrence_ab(c, a, b, x)`: Evaluates Chebyshev series on arbitrary interval $[a, b]$ using Reinsch's modification

Stability:

Evaluation by three-term recurrence is known to be mixed forward-backward stable for x in $[-1, 1]$. Stability outside this interval is not established.

Usage

```
chebyshev_next(x, Tn, Tn_1)

chebyshev_t(n, x)

chebyshev_u(n, x)

chebyshev_t_prime(n, x)

chebyshev_clenshaw_recurrence(c, x)

chebyshev_clenshaw_recurrence_ab(c, a, b, x)
```

Arguments

x	Argument of the polynomial (typically in $[-1, 1]$ for stability)
Tn	Value of the Chebyshev polynomial T_n(x)
Tn_1	Value of the Chebyshev polynomial $T_{n-1}(x)$
n	Degree of the polynomial
c	Vector of coefficients for the Chebyshev series
a	Lower bound of the interval (for interval transformation)
b	Upper bound of the interval (for interval transformation)

Value

A single numeric value with the computed Chebyshev polynomial or series evaluation.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Chebyshev polynomial of the first kind T_2(0.5)
chebyshev_t(2, 0.5)
# Chebyshev polynomial of the second kind U_2(0.5)
chebyshev_u(2, 0.5)
# Derivative of the Chebyshev polynomial of the first kind T_2'(0.5)
chebyshev_t_prime(2, 0.5)
# Next Chebyshev polynomial of the first kind T_3(0.5) using T_2(0.5) and T_1(0.5)
chebyshev_next(0.5, chebyshev_t(2, 0.5), chebyshev_t(1, 0.5))
# Evaluate Chebyshev series with Clenshaw's algorithm
chebyshev_clenshaw_recurrence(c(1, 0, -1), 0.5)
# Evaluate Chebyshev series on interval [0, 1]
chebyshev_clenshaw_recurrence_ab(c(1, 0, -1), 0, 1, 0.5)
```

---

chi_squared_distribution

Chi-Squared Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, quantile function, and sample-size estimation for the Chi-Squared distribution.

With degrees of freedom

$$\nu > 0$$

, the PDF is:

$$f(x;\nu) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} x^{\nu/2-1} e^{-x/2}, \quad x \geq 0$$

and the CDF is given by the regularised incomplete gamma function

$$F(x;\nu) = P(\nu/2, x/2)$$

Accuracy and Implementation Notes: The CDF and quantiles are implemented via incomplete gamma functions. Specifically, the PDF uses `gamma_p_derivative(\nu/2, x/2)/2`, the CDF uses `gamma_p`, the complement uses `gamma_q`, and quantiles use `gamma_p_inv`/`gamma_q_inv`. Accuracy therefore follows that of the incomplete gamma functions.

Sample Size Estimation: `chi_squared_find_degrees_of_freedom` estimates the sample size required to detect a difference from a nominal variance. The sign of `difference_from_variance` determines whether the test is for higher or lower variance.

Usage

```
chi_squared_distribution(df)

chi_squared_pdf(x, df)

chi_squared_lpdf(x, df)

chi_squared_cdf(x, df)

chi_squared_lcdf(x, df)

chi_squared_quantile(p, df)

chi_squared_find_degrees_of_freedom(
  difference_from_variance,
  alpha,
  beta,
  variance,
  hint = 100
)
```

Arguments

`df`	Degrees of freedom (df > 0).
`x`	Quantile value (x >= 0).
`p`	Probability (0 <= p <= 1).
`difference_from_variance`	The difference from the assumed nominal variance that is to be detected.
`alpha`	The acceptable probability of a Type I error (false positive).
`beta`	The acceptable probability of a Type II error (false negative).
`variance`	The assumed nominal variance.
`hint`	An initial guess for the degrees of freedom to start the search from (current sample size is a good start).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Chi-Squared distribution with 3 degrees of freedom
dist <- chi_squared_distribution(3)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
chi_squared_pdf(2, 3)
chi_squared_lpdf(2, 3)
chi_squared_cdf(2, 3)
chi_squared_lcdf(2, 3)
chi_squared_quantile(0.5, 3)

# Find degrees of freedom needed to detect a difference from variance of 2.0
# with alpha = 0.05 and beta = 0.2 when the nominal variance is 5.0
chi_squared_find_degrees_of_freedom(2.0, 0.05, 0.2, 5.0)
```

---

condition_numbers *Condition Numbers*

Description

Functions to compute condition numbers for summation operations.

Usage

```
summation_condition_number(x = 0, kahan = TRUE)

evaluation_condition_number(f, x)
```

Arguments

x	A numeric value.
kahan	A logical value indicating whether to use Kahan summation.
f	A function for which to compute the condition number.

Value

For summation_condition_number, an object with methods to compute the condition number, sum, L1 norm, and to add or subtract values. For evaluation_condition_number, a numeric value representing the condition number of the function evaluation at x.

Examples

```
# Create a summation condition number object
scn <- summation_condition_number(kahan = TRUE)
# Add some values
scn$add(1.0)
scn$add(2.0)
scn$add(3.0)
# Compute sum, condition number, and L1 norm
print(scn$sum())
print(scn$condition_number())
print(scn$l1_norm())
# Compute evaluation condition number for a function
f <- function(x) { x^2 + 3*x + 2 }
print(evaluation_condition_number(f, 1.0))
```

---

constants *Boost Math Constants*

Description

Provides access to mathematical constants used in the Boost Math library.

The available constants include rational fractions,

$$\pi$$

-related values,

$$e$$

-related values, and assorted special constants (e.g., Euler-Mascheroni, Catalan). Integer values are intentionally omitted since they can be constructed exactly from literals.

Accuracy and Implementation Notes: The constants are provided at high precision by Boost.Math; refer to the Boost constants table for names and values.

Usage

```
constants(constant = NULL)
```

Arguments

constant	A string specifying the name of the constant to retrieve. If NULL, returns a list of all constants (see documentation for the full list).

Value

Requested constant value if constant is specified, otherwise a list of all available constants.

See Also

Boost Documentation for more details on the constants.

Examples

```
constants()
```

---

cubic_hermite *Cubic Hermite Interpolator*

Description

The cubic Hermite interpolant takes non-equispaced data and interpolates between them via cubic Hermite polynomials whose slopes must be provided.

Applications:

The interpolant is C1 and evaluation has O(log N) complexity. This interpolator is useful for solution skeletons of ODE steppers.

Usage

```
cubic_hermite(x, y, dydx)
```

Arguments

x	Numeric vector of abscissas (x-coordinates).
y	Numeric vector of ordinates (y-coordinates).
dydx	Numeric vector of derivatives (slopes) at each point.

Value

An object of class `cubic_hermite` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `push_back(x, y, dydx)`: Add a new control point to the interpolator.
- `domain()`: Get the domain of the interpolator.

See Also

Boost Documentation

Examples

```
x <- c(0, 1, 2)
y <- c(0, 1, 0)
dydx <- c(1, 0, -1)
interpolator <- cubic_hermite(x, y, dydx)
xi <- 0.5
interpolator$interpolate(xi)
interpolator$prime(xi)
interpolator$push_back(3, 0, 1)
interpolator$domain()
```

`double_exponential_quadrature` *Double Exponential Quadrature*

Description

Numerical integration using double exponential quadrature methods (Tanh-Sinh, Sinh-Sinh, Exp-Sinh). These methods use variable transformations to achieve high-order convergence, often optimal for functions in the Hardy space (holomorphic in the unit disk).

Tanh-Sinh Quadrature: Best for integration over a finite interval (a, b). Can handle singularities at the endpoints of the integration domain. Converges rapidly for holomorphic integrands.

Sinh-Sinh Quadrature: Designed for integration over the entire real line $(-\infty, \infty)$. Handles integrands with large features or decay properties.

Exp-Sinh Quadrature: Designed for integration over a semi-infinite interval, typically $(0, \infty)$, or ranges like (a, ∞) or $(-\infty, b)$. Supports endpoint singularities.

Usage

```
tanh_sinh(f, a, b, tol = sqrt(.Machine$double.eps), max_refinements = 15)

sinh_sinh(f, tol = sqrt(.Machine$double.eps), max_refinements = 9)

exp_sinh(f, a, b, tol = sqrt(.Machine$double.eps), max_refinements = 9)
```

Arguments

f	A function to integrate. It should accept a single numeric value and return a single numeric value.
a	The lower limit of integration.
b	The upper limit of integration.
tol	The tolerance for the approximation. Default is sqrt(.Machine$double.eps).
max_refinements	The maximum number of refinements to apply. Default is 15 for tanh-sinh and 9 for sinh-sinh and exp-sinh.

Value

A single numeric value with the computed integral.

See Also

numerical_integration

Examples

```
# Tanh-sinh quadrature of log(x) from 0 to 1 (Endpoint singularity)
tanh_sinh(function(x) { log(x) * log1p(-x) }, a = 0, b = 1)

# Sinh-sinh quadrature of exp(-x^2) over (-Inf, Inf)
sinh_sinh(function(x) { exp(-x * x) })

# Exp-sinh quadrature of exp(-3*x) from 0 to Inf
exp_sinh(function(x) { exp(-3 * x) }, a = 0, b = Inf)
```

---

elliptic_integrals *Elliptic Integrals*

Description

Functions to compute various elliptic integrals, including Carlson's elliptic integrals and incomplete elliptic integrals.

Elliptic Integrals - Carlson Form

- ellint_rf(x, y, z): Carlson's Elliptic Integral R_F

$$R_F(x, y, z) = \frac{1}{2} \int_0^\infty [(t + x)(t + y)(t + z)]^{-\frac{1}{2}} dt$$

- ellint_rd(x, y, z): Carlson's Elliptic Integral R_D

$$R_D(x, y, z) = \frac{3}{2} \int_0^\infty [(t + x)(t + y)]^{-\frac{1}{2}} (t + z)^{-\frac{3}{2}} dt$$

- `ellint_rj(x, y, z, p)`: Carlson's Elliptic Integral R_J

$$R_J(x,y,z) = \frac{3}{2}\int_0^\infty (t+p)^{-1}[(t+x)(t+y)(t+z)]^{-\frac{1}{2}}dt$$

- `ellint_rc(x, y)`: Carlson's Elliptic Integral R_C

$$R_C(x,y) = \frac{1}{2}\int_0^\infty (t+x)^{-\frac{1}{2}}(t+y)^{-1}dt$$

- `ellint_rg(x, y, z)`: Carlson's Elliptic Integral R_G

$$R_G(x,y,z) = \frac{1}{4\pi}\int_0^{2\pi}\int_0^{\pi}\sqrt{\left(x\sin^2\theta\cos^2\phi + y\sin^2\theta\sin^2\phi + z\cos^2\theta\right)}\sin\theta d\theta\phi$$

Elliptic Integrals of the First Kind - Legendre Form

- `ellint_1(k, phi)`: Incomplete elliptic integral of the first kind: $F(\phi, k)$:

$$F(\phi,k) = \int_0^\phi \frac{d\theta}{\sqrt{1-k^2\sin^2\theta}}d\theta$$

Elliptic Integrals of the Second Kind - Legendre Form

- `ellint_2(k, phi)`: Incomplete elliptic integral of the second kind: $E(\phi, k)$:

$$E(\phi,k) = \int_0^\phi \sqrt{1-k^2\sin^2\theta}d\theta$$

Elliptic Integrals of the Third Kind - Legendre Form

- `ellint_3(k, n, phi)`: Incomplete elliptic integral of the third kind: $\Pi(n, \phi, k)$:

$$\Pi(n,\phi,k) = \int_0^\phi \frac{d\theta}{\left(1-n\sin^2\theta\right)\sqrt{1-k^2\sin^2\theta}}d\theta$$

Elliptic Integral D - Legendre Form

- `ellint_d(k, phi)`: Incomplete elliptic integral $D(\phi, k)$:

$$D(\phi,k) = \frac{(F(\phi,k) - E(\phi,k))}{k^2}$$

Jacobi Zeta Function

- `jacobi_zeta(k, phi)`: Jacobi Zeta function $Z(\phi, k)$:

$$Z(\phi, k) = E(\phi, k) - \frac{E(\frac{\pi}{2}, k)F(\phi, k)}{F(\frac{\pi}{2}, k)}$$

Heuman Lambda Function

- `heuman_lambda(k, phi)`: Heuman Lambda function $\Lambda_0(\phi, k)$:

$$\Lambda_0(\phi, k) = \frac{F(\phi, \sqrt{1-k^2})}{F(\frac{\pi}{2}, \sqrt{1-k^2})} + \frac{2}{\pi}F(\frac{\pi}{2}, k)Z(\phi, \sqrt{1-k^2})$$

Usage

```
ellint_rf(x, y, z)

ellint_rd(x, y, z)

ellint_rj(x, y, z, p)

ellint_rc(x, y)

ellint_rg(x, y, z)

ellint_1(k, phi = NULL)

ellint_2(k, phi = NULL)

ellint_3(k, n, phi = NULL)

ellint_d(k, phi = NULL)

jacobi_zeta(k, phi)

heuman_lambda(k, phi)
```

Arguments

x	First parameter of Carlson's integral (must be non-negative)
y	Second parameter of Carlson's integral
z	Third parameter of Carlson's integral
p	Fourth parameter of the integral (for Rj, must be non-zero)
k	Elliptic modulus for Legendre-form integrals
phi	Amplitude (angle) for incomplete elliptic integrals
n	Characteristic for elliptic integrals of the third kind

Value

A single numeric value with the computed elliptic integral.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Carlson's elliptic integral Rf with parameters x = 1, y = 2, z = 3
ellint_rf(1, 2, 3)
# Carlson's elliptic integral Rd with parameters x = 1, y = 2, z = 3
ellint_rd(1, 2, 3)
# Carlson's elliptic integral Rj with parameters x = 1, y = 2, z = 3, p = 4
ellint_rj(1, 2, 3, 4)
# Carlson's elliptic integral Rc with parameters x = 1, y = 2
ellint_rc(1, 2)
# Carlson's elliptic integral Rg with parameters x = 1, y = 2, z = 3
ellint_rg(1, 2, 3)
# Incomplete elliptic integral of the first kind with k = 0.5, phi = pi/4
ellint_1(0.5, pi / 4)
# Complete elliptic integral of the first kind
ellint_1(0.5)
# Incomplete elliptic integral of the second kind with k = 0.5, phi = pi/4
ellint_2(0.5, pi / 4)
# Complete elliptic integral of the second kind
ellint_2(0.5)
# Incomplete elliptic integral of the third kind with k = 0.5, n = 0.5, phi = pi/4
ellint_3(0.5, 0.5, pi / 4)
# Complete elliptic integral of the third kind with k = 0.5, n = 0.5
ellint_3(0.5, 0.5)
# Incomplete elliptic integral D with k = 0.5, phi = pi/4
ellint_d(0.5, pi / 4)
# Complete elliptic integral D
ellint_d(0.5)
# Jacobi zeta function with k = 0.5, phi = pi/4
jacobi_zeta(0.5, pi / 4)
# Heuman's lambda function with k = 0.5, phi = pi/4
heuman_lambda(0.5, pi / 4)
```

---

empirical_cumulative_distribution_function

Empirical Cumulative Distribution Function (ECDF)

Description

Create an empirical cumulative distribution function (ECDF) from a given vector.

Usage

```
empirical_cumulative_distribution_function(data, sorted = FALSE)
```

Arguments

`data`	A numeric vector of data points.
`sorted`	A logical indicating whether the data is already sorted. Default is FALSE.

Details

The ECDF is a step function constructed from observed data that converges to the true CDF as sample size grows. It is commonly used in goodness-of-fit workflows that compare the empirical CDF to a hypothesised distribution.

Implementation Notes: Data must be sorted; by default the constructor sorts at

$$O(n \log n)$$

cost. If the data is already sorted, set `sorted = TRUE` to avoid the sort. Evaluation uses binary search (`upper_bound`) and runs in

$$O(\log n)$$

time.

Value

An object representing the ECDF, with member function `$ecdf(x)` to evaluate the ECDF at point(s) `x`.

Examples

```
data <- c(1.2, 2.3, 3.1, 4.5, 5.0)
ecdf_obj <- empirical_cumulative_distribution_function(data)
ecdf_obj$ecdf(3.0)  # Evaluate ECDF at x = 3.0
```

`error_functions` *Error Functions and Inverses*

Description

Functions to compute the error function, complementary error function, and their inverses.

Error functions appear frequently in probability, statistics, and partial differential equations, particularly in the study of normal distributions and diffusion processes.

Error Function:

The error function is defined by the integral:

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

The error function is an odd function (erf(-z) = -erf(z)). Implementation uses rational approximations optimised for absolute error, particularly for |z| <= 0.5.

Complementary Error Function:

The complementary error function is defined as:

$$erfc(z) = 1 - erf(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2} dt$$

Key reflection formulas:

- erfc(-z) = 2 - erfc(z) (preferred when -z < -0.5)
- erfc(-z) = 1 + erf(z) (preferred when -0.5 <= -z < 0)

For large z, uses exponential scaling to maintain numerical stability.

Inverse Functions:

- `erf_inv(p)`: Returns x such that p = erf(x), where -1 <= p <= 1
- `erfc_inv(p)`: Returns x such that p = erfc(x), where 0 <= p <= 2

Inverse functions use rational approximations with different formulas for different ranges of p, achieving accuracy to less than ~2 epsilon for standard precision types.

Usage

```
erf(x)

erfc(x)

erf_inv(p)

erfc_inv(p)
```

Arguments

x	Input numeric value
p	Probability value for the inverse functions

Value

A single numeric value with the computed error function, complementary error function, or their inverses.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Error function
erf(0.5)
# Complementary error function
erfc(0.5)
# Inverse error function
erf_inv(0.5)
```

```
# Inverse complementary error function
erfc_inv(0.5)
```

`exponential_distribution` *Exponential Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Exponential distribution.

With rate parameter $\lambda > 0$, the PDF and CDF are

$$f(x; \lambda) = \lambda e^{-\lambda x}, \quad x \geq 0$$
$$F(x; \lambda) = 1 - e^{-\lambda x}$$

and the quantile is

$$F^{-1}(p; \lambda) = -\frac{\log(1-p)}{\lambda}$$

Usage

```
exponential_distribution(lambda = 1)

exponential_pdf(x, lambda = 1)

exponential_lpdf(x, lambda = 1)

exponential_cdf(x, lambda = 1)

exponential_lcdf(x, lambda = 1)

exponential_quantile(p, lambda = 1)
```

Arguments

`lambda`	Rate parameter (lambda > 0).
`x`	Quantile value (x >= 0).
`p`	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Exponential distribution with rate parameter lambda = 2
dist <- exponential_distribution(2)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
exponential_pdf(1, 2)
exponential_lpdf(1, 2)
exponential_cdf(1, 2)
exponential_lcdf(1, 2)
exponential_quantile(0.5, 2)
```

---

exponential_integrals *Exponential Integrals*

Description

Functions to compute various exponential integrals, including En and Ei.

Exponential Integral En:

Defined by the integral:

$$E_n(z) = \int_1^\infty \frac{e^{-zt}}{t^n} dt$$

Exponential Integral Ei:

Defined by the integral:

$$Ei(z) = -\int_{-z}^\infty \frac{e^{-t}}{t} dt$$

Usage

```
expint_en(n, z)

expint_ei(z)
```

Arguments

n	Order of the integral (for En), must be a non-negative integer
z	Argument of the integral (for En and Ei)

Value

A single numeric value with the computed exponential integral.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Exponential integral En with n = 1 and z = 2
expint_en(1, 2)
# Exponential integral Ei with z = 2
expint_ei(2)
```

---

extreme_value_distribution

Extreme Value Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Extreme Value (Gumbel) distribution.

With location a and scale $b > 0$, the PDF and CDF are

$$f(x; a, b) = \frac{1}{b} \exp\left(\frac{a - x}{b}\right) \exp\left(-\exp\left(\frac{a - x}{b}\right)\right)$$

$$F(x; a, b) = \exp\left(-\exp\left(\frac{a - x}{b}\right)\right)$$

and the quantile is

$$F^{-1}(p; a, b) = a - b \log\left(-\log(p)\right)$$

.

Usage

```
extreme_value_distribution(location = 0, scale = 1)

extreme_value_pdf(x, location = 0, scale = 1)

extreme_value_lpdf(x, location = 0, scale = 1)

extreme_value_cdf(x, location = 0, scale = 1)

extreme_value_lcdf(x, location = 0, scale = 1)

extreme_value_quantile(p, location = 0, scale = 1)
```

Arguments

location	Location parameter (default is 0).
scale	Scale parameter (default is 1).
x	Quantile value.
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Extreme Value distribution with location = 0, scale = 1
dist <- extreme_value_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
```

```
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
extreme_value_pdf(0)
extreme_value_lpdf(0)
extreme_value_cdf(0)
extreme_value_lcdf(0)
extreme_value_quantile(0.5)
```

`factorials_and_binomial_coefficients`
Factorials and Binomial Coefficients

Description

Functions to compute factorials, double factorials, rising and falling factorials (Pochhammer symbols), and binomial coefficients.

These fundamental combinatorial functions appear in counting problems, probability distributions, and series expansions of special functions.

Factorial Functions:

- `factorial_boost(i)`: Computes $i! = 1 * 2 * 3 * ... * i$
  - Standard factorial with overflow checking
  - Returns error for $i >$ max_factorial()
- `unchecked_factorial(i)`: Fast table lookup for small factorials
  - No overflow checking, assumes i is valid
  - Use when performance is critical and i is known to be small
- `max_factorial()`: Returns the largest i for which $factorial_boost(i)$ fits in the return type without overflow

Double Factorial:

- `double_factorial(i)`: Computes $i!! = i * (i-2) * (i-4) * ...$
  - For even i: $i!! = i * (i-2) * ... * 4 * 2$
  - For odd i: $i!! = i * (i-2) * ... * 3 * 1$

Rising and Falling Factorials (Pochhammer Symbols):

- `rising_factorial(x, i)`: Computes $x^{(i)} = x(x+1)(x+2)...(x+i-1)$
  - Also called Pochhammer symbol or ascending factorial
  - Used in hypergeometric functions and series expansions
  - For integer x, equals $(x+i-1)!/(x-1)!$
- `falling_factorial(x, i)`: Computes $(x)_i = x(x-1)(x-2)...(x-i+1)$
  - Also called descending factorial

  - Counts permutations: number of ways to arrange i items from x items
  - For integer x, $equals x!/(x-i)!$

Binomial Coefficients:

- `binomial_coefficient(n, k)`: Computes $C(n,k) = n!/(k!(n-k)!)$
  - "n choose k": number of ways to choose k items from n items

Usage

```
factorial_boost(i)

unchecked_factorial(i)

max_factorial()

double_factorial(i)

rising_factorial(x, i)

falling_factorial(x, i)

binomial_coefficient(n, k)
```

Arguments

i	Non-negative integer input for factorials and double factorials
x	Base value for rising and falling factorials (can be real-valued)
n	Total number of elements for binomial coefficients
k	Number of elements to choose for binomial coefficients (0 <= k <= n)

Value

A single numeric value with the computed factorial, double factorial, rising factorial, falling factorial, or binomial coefficient.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Factorial of 5: 5! = 120
factorial_boost(5)
# Unchecked factorial of 5 (fast table lookup)
unchecked_factorial(5)
# Maximum factorial value that can be computed
max_factorial()
# Double factorial: 6!! = 6*4*2 = 48
double_factorial(6)
```

```
# Rising factorial: 3^(2) = 3*4 = 12
rising_factorial(3, 2)
# Falling factorial: 3^[2] = 3*2 = 6
falling_factorial(3, 2)
# Binomial coefficient: C(5,2) = 10
binomial_coefficient(5, 2)
```

`filters` *Filters*

Description

Functions to compute Daubechies scaling and wavelet filter coefficients.

The returned coefficients correspond to the compactly supported Daubechies wavelets indexed by the number of vanishing moments p, returning $2p$ taps.

Conventions: Boost indexes filters by vanishing moments (as in PyWavelets and Mathematica), normalizes coefficients to unit ℓ_2 norm, and uses the convolutional ordering shown in Daubechies (1988). Other libraries may index by number of taps, use a $\sqrt{2}$ scaling, or reverse coefficient order.

Usage

```
daubechies_scaling_filter(order)

daubechies_wavelet_filter(order)
```

Arguments

`order`	An integer specifying the number of vanishing moments (1 to 19).

Value

A numeric vector of size 2*order containing the filter coefficients.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Daubechies Scaling Filter of order 4
daubechies_scaling_filter(4)
# Daubechies Wavelet Filter of order 4
daubechies_wavelet_filter(4)
```

`fisher_f_distribution` *Fisher F Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Fisher F distribution.

The PDF is:

$$f(x; \nu_1, \nu_2) = \frac{\sqrt{\frac{(\nu_1 x)^{\nu_1} \nu_2^{\nu_2}}{(\nu_1 x + \nu_2)^{\nu_1 + \nu_2}}}}{x\, B(\nu_1/2, \nu_2/2)}$$

The CDF is:

$$F(x; \nu_1, \nu_2) = I_{\nu_1 x/(\nu_1 x + \nu_2))(\frac{\nu_1}{2}, \frac{\nu_2}{2})}$$

Where $I_{.}(.,.)$ is the regularised incomplete beta function

Usage

```
fisher_f_distribution(df1, df2)

fisher_f_pdf(x, df1, df2)

fisher_f_lpdf(x, df1, df2)

fisher_f_cdf(x, df1, df2)

fisher_f_lcdf(x, df1, df2)

fisher_f_quantile(p, df1, df2)
```

Arguments

`df1`	Degrees of freedom for the numerator (df1 > 0).
`df2`	Degrees of freedom for the denominator (df2 > 0).
`x`	Quantile value (x >= 0).
`p`	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Fisher F distribution with df1 = 5, df2 = 10
dist <- fisher_f_distribution(5, 10)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
fisher_f_pdf(1, 5, 10)
fisher_f_lpdf(1, 5, 10)
fisher_f_cdf(1, 5, 10)
fisher_f_lcdf(1, 5, 10)
fisher_f_quantile(0.5, 5, 10)
```

---

fp_utilities *Floating Point Utilities*

Description

Utilities for floating-point number manipulation and analysis, including adjacent representable values, ULP distances, and condition numbers.

Floating-Point Navigation:

- `float_next(x)` / `float_prior(x)` move to the next greater/smaller representable value.
- `float_distance(x, y)` returns the representation distance in ULPs.
- `float_advance(x, n)` advances by n ULPs.
- `ulp(x)` returns the size of one unit in the last place at `x`.

Comparisons:

- `relative_difference(x, y)` and `epsilon_difference(x, y)` provide scale-aware measures of deviation.

Condition Numbers: `summation_condition_number` and `evaluation_condition_number` help quantify numerical sensitivity to perturbations.

Usage

```
float_next(x)

float_prior(x)

float_distance(x, y)

float_advance(x, distance)

ulp(x)

relative_difference(x, y)

epsilon_difference(x, y)
```

Arguments

`x`	A numeric value.
`y`	A numeric value.
`distance`	Integer number of ULPs to advance by.

Value

A numeric value after performing the specified floating point operation.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
print(float_next(1.0), digits = 20)
print(float_distance(1.0, 2.0), digits = 20)
print(float_prior(1.0), digits = 20)
print(float_advance(1.0, 10), digits = 20)
print(ulp(1.0), digits = 20)
print(relative_difference(1.1, 1.1000009), digits = 20)
print(epsilon_difference(1.1, 1.1000009), digits = 20)
```

`gamma_distribution` *Gamma Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Gamma distribution.

The PDF is

$$f(x; k, \lambda) = \frac{1}{\Gamma(k)\,\theta^k} x^{k-1} e^{-x/\theta}, \quad x \geq 0$$

The CDF is

$$F(x; k, \lambda) = P(k, x/\theta)$$

Where $P(.,.)$ is the regularised incomplete gamma function.

Usage

```
gamma_distribution(shape, scale = 1)

gamma_pdf(x, shape, scale = 1)

gamma_lpdf(x, shape, scale = 1)

gamma_cdf(x, shape, scale = 1)

gamma_lcdf(x, shape, scale = 1)

gamma_quantile(p, shape, scale = 1)
```

Arguments

`shape`	Shape parameter (shape > 0).
`scale`	Scale parameter (scale > 0).
`x`	Quantile value (x >= 0).
`p`	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Gamma distribution with shape = 3, scale = 4
dist <- gamma_distribution(3, 4)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
gamma_pdf(2, 3, 4)
gamma_lpdf(2, 3, 4)
gamma_cdf(2, 3, 4)
gamma_lcdf(2, 3, 4)
gamma_quantile(0.5, 3, 4)
```

`gamma_functions` *Gamma Functions*

Description

Functions to compute the gamma function, its logarithm, digamma, trigamma, polygamma, and various incomplete gamma functions.

Gamma Function Gamma(z):

- `tgamma(z)`: Computes $\Gamma(z)$, the true gamma function
- `tgamma1pm1(z)`: Computes $\Gamma(1+z)-1$ with enhanced numerical stability for very small z values, avoiding precision loss
- `lgamma_boost(z)`: Returns $\log(\Gamma(z))$, the logarithm of the gamma function

Derivative Functions:

- `digamma_boost(z)`: The digamma function, the first derivative of the logarithm of the Gamma function $\psi(z) = \frac{d}{dz}\log\Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}$,

- `trigamma_boost(z)`: The trigamma function, the second derivative of the logarithm of the Gamma function $\psi_1(z) = \frac{d^2}{dz^2} \log \Gamma(z)$
- `polygamma(n, z)`: The nth derivative of the digamma function (n-th order polygamma) $\psi^{(m)}(z) = \frac{d^m}{dz^m} \psi(z)$

Ratios:

- `tgamma_ratio(a, b)`: Computes $\Gamma(a)/\Gamma(b)$
- `tgamma_delta_ratio(a, delta)`: Computes $\Gamma(a)/\Gamma(a + \delta)$

Incomplete Gamma Functions:

These functions require a > 0 and z >= 0.

- **Normalised (Regularised) Functions** (return values in $[0, 1]$):
  - `gamma_p(a, z)`: Normalised lower incomplete gamma $P(a, z) = \frac{\gamma(a,z)}{\Gamma(z)}$
  - `gamma_q(a, z)`: Normalised upper incomplete gamma $Q(a, z) = \frac{\Gamma(a,z)}{\Gamma(a)}$
- **Non-normalised Functions** (return values in $[0, Gamma(a)]$):
  - `tgamma_lower(a, z)`: Full lower incomplete gamma function $\gamma(a, z) = \int_0^z t^{a-1} e^{-t} dt$
  - `tgamma_upper(a, z)`: Full upper incomplete gamma function $\Gamma(a, z) = \int_z^\infty t^{a-1} e^{-t} dt$

Inverse Functions:

- `gamma_p_inv(a, p)`: Returns z such that $p = P(a, z)$
- `gamma_q_inv(a, q)`: Returns z such that $q = Q(a, z)$
- `gamma_p_inva(z, p)`: Returns a such that $p = P(a, z)$
- `gamma_q_inva(z, q)`: Returns a such that $q = Q(a, z)$

Derivative:

- `gamma_p_derivative(a, z)`: Computes the derivative of the normalised lower incomplete gamma function with respect to z: $\frac{\partial}{\partial z} P(a, z) = \frac{e^{-z} z^{a-1}}{\Gamma(a)}$

Usage

```
tgamma(z)

tgamma1pm1(z)

lgamma_boost(z)

digamma_boost(z)

trigamma_boost(z)

polygamma(n, z)

tgamma_ratio(a, b)
```

```
tgamma_delta_ratio(a, delta)

gamma_p(a, z)

gamma_q(a, z)

tgamma_lower(a, z)

tgamma_upper(a, z)

gamma_q_inv(a, q)

gamma_p_inv(a, p)

gamma_q_inva(z, q)

gamma_p_inva(z, p)

gamma_p_derivative(a, z)
```

Arguments

z	Input numeric value for the gamma function
n	Order of the polygamma function (non-negative integer)
a	Argument for the incomplete gamma functions (must be positive)
b	Denominator argument for the ratio of gamma functions
delta	Increment for the ratio of gamma functions
q	Probability value for the incomplete gamma functions (0 <= q <= 1)
p	Probability value for the incomplete gamma functions (0 <= p <= 1)

Value

A single numeric value with the computed gamma function, logarithm, digamma, trigamma, polygamma, or incomplete gamma functions.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
## Not run:
# Gamma function for z = 5
tgamma(5)
# Gamma function for (1 + z) - 1, where z = 5
tgamma1pm1(5)
# Logarithm of the gamma function for z = 5
```

```
lgamma_boost(5)
# Digamma function for z = 5
digamma_boost(5)
# Trigamma function for z = 5
trigamma_boost(5)
# Polygamma function of order 1 for z = 5
polygamma(1, 5)
# Ratio of gamma functions for a = 5, b = 3
tgamma_ratio(5, 3)
# Ratio of gamma functions with delta for a = 5, delta = 2
tgamma_delta_ratio(5, 2)
# Normalised lower incomplete gamma function P(a, z) for a = 5, z = 2
gamma_p(5, 2)
# Normalised upper incomplete gamma function Q(a, z) for a = 5, z = 2
gamma_q(5, 2)
# Full lower incomplete gamma function for a = 5, z = 2
tgamma_lower(5, 2)
# Full upper incomplete gamma function for a = 5, z = 2
tgamma_upper(5, 2)
# Inverse of the normalised upper incomplete gamma function for a = 5, q = 0.5
gamma_q_inv(5, 0.5)
# Inverse of the normalised lower incomplete gamma function for a = 5, p = 0.5
gamma_p_inv(5, 0.5)
# Inverse of the normalised upper incomplete gamma function with respect to a for z = 2, q = 0.5
gamma_q_inva(2, 0.5)
# Inverse of the normalised lower incomplete gamma function with respect to a for z = 2, p = 0.5
gamma_p_inva(2, 0.5)
# Derivative of the normalised lower incomplete gamma function for a = 5, z = 2
gamma_p_derivative(5, 2)

## End(Not run)
```

`gegenbauer_polynomials`

Gegenbauer Polynomials and Related Functions

Description

Functions to compute Gegenbauer polynomials, their derivatives, and related functions.

Usage

```
gegenbauer(n, lambda, x)

gegenbauer_prime(n, lambda, x)

gegenbauer_derivative(n, lambda, x, k)
```

Arguments

n	Degree of the polynomial
lambda	Parameter of the polynomial
x	Argument of the polynomial
k	Order of the derivative

Value

A single numeric value with the computed Gegenbauer polynomial, its derivative, or k-th derivative.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Gegenbauer polynomial C_2^(1)(0.5)
gegenbauer(2, 1, 0.5)
# Derivative of the Gegenbauer polynomial C_2^(1)'(0.5)
gegenbauer_prime(2, 1, 0.5)
# k-th derivative of the Gegenbauer polynomial C_2^(1)''(0.5)
gegenbauer_derivative(2, 1, 0.5, 2)
```

---

generic_distribution_functions

Generic Distribution Functions

Description

Generic functions for computing various properties of statistical distributions.

Usage

```
cdf(x, ...)

logcdf(x, ...)

pdf(x, ...)

logpdf(x, ...)

hazard(x, ...)

chf(x, ...)

mode(x, ...)
```

```
standard_deviation(x, ...)

support(x, ...)

variance(x, ...)

skewness(x, ...)

kurtosis(x, ...)

kurtosis_excess(x, ...)
```

Arguments

x	A distribution object created by a distribution constructor function.
...	Additional arguments passed to specific methods.

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, quantile, mean, median, mode, range, standard deviation, support, variance, skewness, kurtosis, or excess kurtosis depending on the function called.

Examples

```
# Create a Weibull distribution
weibull_dist <- weibull_distribution(shape = 2, scale = 1)
# Compute the CDF at a specific point
cdf(weibull_dist, 1)
# Check support
support(weibull_dist)
```

---

geometric_distribution

Geometric Distribution Functions

Description

Functions to compute the probability mass function (pmf), cumulative distribution function, quantile function, and confidence bounds for the Geometric distribution.

The geometric distribution models the number of failures k before the first success in Bernoulli trials with success probability p. The pmf is

$$P(X = k) = (1 - p)^k p, \quad k \in \{0, 1, 2, \dots\}$$

Confidence Bounds: The bound and trial-estimation functions are implemented as in the negative binomial distribution (successes = 1), using Clopper-Pearson style bounds and numeric inversion.

Usage

```
geometric_distribution(prob)

geometric_pdf(x, prob)

geometric_lpdf(x, prob)

geometric_cdf(x, prob)

geometric_lcdf(x, prob)

geometric_quantile(p, prob)

geometric_find_lower_bound_on_p(trials, alpha)

geometric_find_upper_bound_on_p(trials, alpha)

geometric_find_minimum_number_of_trials(failures, prob, alpha)

geometric_find_maximum_number_of_trials(failures, prob, alpha)
```

Arguments

`prob`	Probability of success (0 < prob < 1).
`x`	Quantile value (non-negative integer).
`p`	Probability (0 <= p <= 1).
`trials`	Number of trials.
`alpha`	Largest acceptable probability that the true value of the success fraction is less than the value returned (by `geometric_find_lower_bound_on_p`) or greater than the value returned (by `geometric_find_upper_bound_on_p`).
`failures`	Number of failures.

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Geometric distribution with probability of success prob = 0.5
dist <- geometric_distribution(0.5)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
```

```
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
geometric_pdf(3, 0.5)
geometric_lpdf(3, 0.5)
geometric_cdf(3, 0.5)
geometric_lcdf(3, 0.5)
geometric_quantile(0.5, 0.5)
## Not run:
# Find lower bound on p given 5 trials with 95% confidence
geometric_find_lower_bound_on_p(5, 0.05)
# Find upper bound on p given 5 trials with 95% confidence
geometric_find_upper_bound_on_p(5, 0.05)
# Find minimum number of trials to observe 3 failures with p = 0.5 at 95% confidence
geometric_find_minimum_number_of_trials(3, 0.5, 0.05)
# Find maximum number of trials to observe 3 failures with p = 0.5 at 95% confidence
geometric_find_maximum_number_of_trials(3, 0.5, 0.05)

## End(Not run)
```

`hankel_functions` *Hankel Functions*

Description

Functions to compute cylindrical and spherical Hankel functions of the first and second kinds.

Cyclic Hankel Functions

- `cyl_hankel_1(v, x)`: The Hankel function of the first kind: $H_v^{(1)}(x) = J_v(x) + iY_v(x)$
- `cyl_hankel_2(v, x)`: The Hankel function of the second kind: $H_v^{(2)}(x) = J_v(x) - iY_v(x)$

Where $J_v(x)$ is the Bessel function of the first kind and $Y_v(x)$ is the Bessel function of the second kind.

Spherical Hankel Functions:

- `sph_hankel_1(v, x)`: The spherical Hankel function of the first kind: $h_v^{(1)}(x) = \sqrt{\frac{\pi}{2}} \frac{1}{\sqrt{\pi}} H^{(1)}_{v+\frac{1}{2}}(x)$
- `sph_hankel_2(v, x)`: The spherical Hankel function of the second kind: $h_v^{(2)}(x) = \sqrt{\frac{\pi}{2}} \frac{1}{\sqrt{\pi}} H^{(2)}_{v+\frac{1}{2}}(x)$

Usage

```
cyl_hankel_1(v, x)

cyl_hankel_2(v, x)

sph_hankel_1(v, x)

sph_hankel_2(v, x)
```

Arguments

v	Order of the Hankel function (can be any real number)
x	Argument of the Hankel function (can be any real number)

Value

A single complex value with the computed Hankel function.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
cyl_hankel_1(2, 0.5)
cyl_hankel_2(2, 0.5)
sph_hankel_1(2, 0.5)
sph_hankel_2(2, 0.5)
```

`hermite_polynomials` *Hermite Polynomials and Related Functions*

Description

Functions to compute Hermite polynomials using three-term recurrence relations.

Hermite polynomials are orthogonal polynomials that appear in probability theory (as derivatives of the Gaussian function), quantum mechanics (quantum harmonic oscillator), and numerical analysis.

Hermite Polynomials H_n(x):

- `hermite(n, x)`: Evaluates the Hermite polynomial of degree n at point x

- Orthogonal with respect to the weight function

$$e^{-x^2}$$

  on (-Inf, Inf)
- Appear as eigenfunctions of the quantum harmonic oscillator

Recurrence Relation:

- `hermite_next(n, x, Hn, Hnm1)`: Computes

$$H_{n+1}(x)$$

  from H_n and

$$H_{n-1}$$

- Uses stable three-term recurrence for sequential computation

Implementation Notes:

- Guarantees low absolute error but not low relative error near polynomial roots
- Values greater than ~120 for n are unlikely to produce sensible results
- Relative errors may grow arbitrarily large when the function is very close to a root

Usage

```
hermite(n, x)

hermite_next(n, x, Hn, Hnm1)
```

Arguments

`n`	Degree of the polynomial (practical limit ~120)
`x`	Argument of the polynomial
`Hn`	Value of the Hermite polynomial H_n(x)
`Hnm1`	Value of the Hermite polynomial $H_{n-1}(x)$

Value

A single numeric value with the computed Hermite polynomial.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Hermite polynomial H_2(0.5)
hermite(2, 0.5)
# Next Hermite polynomial H_3(0.5) using H_2(0.5) and H_1(0.5)
hermite_next(2, 0.5, hermite(2, 0.5), hermite(1, 0.5))
```

`holtsmark_distribution` *Holtsmark Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Holtsmark distribution.

The PDF is:

$$f(x;\mu,c) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \exp\left(it\mu - |ct|^{3/2}\right) e^{-ixt}dt$$

Usage

```
holtsmark_distribution(location = 0, scale = 1)

holtsmark_pdf(x, location = 0, scale = 1)

holtsmark_lpdf(x, location = 0, scale = 1)

holtsmark_cdf(x, location = 0, scale = 1)

holtsmark_lcdf(x, location = 0, scale = 1)

holtsmark_quantile(p, location = 0, scale = 1)
```

Arguments

`location`	Location parameter (default is 0).
`scale`	Scale parameter (default is 1).
`x`	Quantile value.
`p`	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Holtsmark distribution with location 0 and scale 1
dist <- holtsmark_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)

# Convenience functions
holtsmark_pdf(3)
holtsmark_lpdf(3)
holtsmark_cdf(3)
holtsmark_lcdf(3)
holtsmark_quantile(0.5)
```

---

hyperexponential_distribution

Hyperexponential Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Hyperexponential distribution.

A k-phase hyperexponential distribution is a mixture of k exponential distributions with phase probabilities α_i and rates λ_i. The PDF and CDF are

$$f(x; \alpha, \lambda) = \sum_{i=1}^{k} \alpha_i \lambda_i e^{-\lambda_i x}$$

$$F(x; \alpha, \lambda) = 1 - \sum_{i=1}^{k} \alpha_i e^{-\lambda_i x}$$

.

Usage

```
hyperexponential_distribution(probabilities, rates)

hyperexponential_pdf(x, probabilities, rates)

hyperexponential_lpdf(x, probabilities, rates)

hyperexponential_cdf(x, probabilities, rates)

hyperexponential_lcdf(x, probabilities, rates)

hyperexponential_quantile(p, probabilities, rates)
```

Arguments

probabilities	Vector of non-negative probabilities (will be normalised to sum to 1).
rates	Vector of positive rates (all rates must be > 0).
x	Quantile value (x >= 0).
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Hyperexponential distribution with probabilities = c(0.5, 0.5) and rates = c(1, 2)
dist <- hyperexponential_distribution(c(0.5, 0.5), c(1, 2))
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
```

```
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
hyperexponential_pdf(2, c(0.5, 0.5), c(1, 2))
hyperexponential_lpdf(2, c(0.5, 0.5), c(1, 2))
hyperexponential_cdf(2, c(0.5, 0.5), c(1, 2))
hyperexponential_lcdf(2, c(0.5, 0.5), c(1, 2))
hyperexponential_quantile(0.5, c(0.5, 0.5), c(1, 2))
```

`hypergeometric_distribution`

Hypergeometric Distribution Functions

Description

Functions to compute the probability mass function (pmf), cumulative distribution function, and quantile function for the Hypergeometric distribution.

Usage

```
hypergeometric_distribution(r, n, N)

hypergeometric_pdf(x, r, n, N)

hypergeometric_lpdf(x, r, n, N)

hypergeometric_cdf(x, r, n, N)

hypergeometric_lcdf(x, r, n, N)

hypergeometric_quantile(p, r, n, N)
```

Arguments

`r`	Number of successes in the population (r >= 0).
`n`	Number of draws (n >= 0).
`N`	Population size (N >= r).
`x`	Quantile value (non-negative integer).
`p`	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Hypergeometric distribution with r = 5, n = 10, N = 20
dist <- hypergeometric_distribution(5, 10, 20)
# Apply generic functions
cdf(dist, 4)
logcdf(dist, 4)
pdf(dist, 4)
logpdf(dist, 4)
hazard(dist, 4)
chf(dist, 4)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
hypergeometric_pdf(3, 5, 10, 20)
hypergeometric_lpdf(3, 5, 10, 20)
hypergeometric_cdf(3, 5, 10, 20)
hypergeometric_lcdf(3, 5, 10, 20)
hypergeometric_quantile(0.5, 5, 10, 20)
```

`hypergeometric_functions` *Hypergeometric Functions*

Description

Functions to compute various hypergeometric functions, which are solutions to hypergeometric differential equations and generalize many special functions.

Specialised Hypergeometric Functions:

- `hypergeometric_1F0(a, z)`:

$$ {}_1F_0(a; z) = \sum_{n=0}^{\infty} \frac{(a)_n z^n}{n!} $$

- `hypergeometric_0F1(b, z)`:

$$ {}_0F_1(;b;z) = \sum_{n=0}^{\infty} \frac{z^n}{(b)_n!} $$

- `hypergeometric_2F0(a1, a2, z)`:

$$ {}_2F_0(a_1, a_2; z) = \sum_{n=0}^{\infty} \frac{(a_1)_n (a_2)_n z^n}{n!} $$

- `hypergeometric_1F1(a, b, z)`:

$$ {}_1F_1(a, b; z) = \sum_{n=0}^{\infty} \frac{(a)_n z^n}{(b)_n!} $$

- `hypergeometric_1F1_regularized(a, b, z)`:

$$ {}_1\tilde{F}_1(a, b; z) = \frac{{}_1F_1(a, b; z)}{\Gamma(b)} $$

- `log_hypergeometric_1F1(a, b, z)`: Numerically stable implementation of $\ln_1 F_1(a, b; z)$

Generalised Hypergeometric Function pFq:

- `hypergeometric_pFq(a, b, z)`: General form with p numerator parameters and q denominator parameters:

$$ {}_pF_q\left(\{a_1, ..., a_p\}, \{b_1, ..., b_q\}; z\right) = \sum_{n=0}^{\infty} \frac{\prod_{j=1}^{p} (a_j)_n}{\prod_{j=1}^{q} (b_j)_n} \frac{z^n}{n!} $$

Usage

```
hypergeometric_1F0(a, z)

hypergeometric_0F1(b, z)

hypergeometric_2F0(a1, a2, z)

hypergeometric_1F1(a, b, z)

hypergeometric_1F1_regularized(a, b, z)

log_hypergeometric_1F1(a, b, z)

hypergeometric_pFq(a, b, z)
```

Arguments

a	Parameter of the hypergeometric function (numerator parameter)
z	Argument of the hypergeometric function
b	Denominator parameter of the hypergeometric function
a1	First numerator parameter (for 2F0)
a2	Second numerator parameter (for 2F0)

Value

A single numeric value with the computed hypergeometric function.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Hypergeometric Function 1F0
hypergeometric_1F0(2, 0.2)
# Hypergeometric Function 0F1
hypergeometric_0F1(1, 0.8)
# Hypergeometric Function 2F0
hypergeometric_2F0(0.1, -1, 0.1)
# Hypergeometric Function 1F1 (Kummer's function)
hypergeometric_1F1(2, 3, 1)
# Regularised Hypergeometric Function 1F1
hypergeometric_1F1_regularized(2, 3, 1)
# Logarithm of the Hypergeometric Function 1F1
log_hypergeometric_1F1(2, 3, 1)
# Generalised Hypergeometric Function pFq (3F4 example)
hypergeometric_pFq(c(2, 3, 4), c(5, 6, 7, 8), 0.5)
```

---

inverse_chi_squared_distribution

Inverse Chi-Squared Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Inverse Chi-Squared distribution.

For degrees of freedom ν and scale ξ, the PDF is

$$f(x;\nu,\xi) = \frac{(\nu\xi/2)^{\nu/2}}{\Gamma(\nu/2)} x^{-1-\nu/2} \exp\left(-\frac{\nu\xi}{2x}\right)$$

The CDF is:

$$F(x;\nu,\xi) = \frac{\Gamma(\nu/2, \nu\xi/2x)}{\Gamma(\nu/2)}$$

The unscaled case corresponds to $\xi = 1/\nu$.

Usage

```
inverse_chi_squared_distribution(df = 1, scale = 1/df)

inverse_chi_squared_pdf(x, df = 1, scale = 1/df)

inverse_chi_squared_lpdf(x, df = 1, scale = 1/df)

inverse_chi_squared_cdf(x, df = 1, scale = 1/df)

inverse_chi_squared_lcdf(x, df = 1, scale = 1/df)

inverse_chi_squared_quantile(p, df = 1, scale = 1/df)
```

Arguments

df	Degrees of freedom (df > 0; default is 1).
scale	Scale parameter (default is 1/df).
x	Quantile value (x >= 0).
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Inverse Chi-Squared distribution with 10 degrees of freedom, scale = 1
dist <- inverse_chi_squared_distribution(10, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
```

```
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
inverse_chi_squared_pdf(2, 10, 1)
inverse_chi_squared_lpdf(2, 10, 1)
inverse_chi_squared_cdf(2, 10, 1)
inverse_chi_squared_lcdf(2, 10, 1)
inverse_chi_squared_quantile(0.5, 10, 1)
```

`inverse_gamma_distribution`

Inverse Gamma Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Inverse Gamma distribution.

With shape α and scale β, the PDF is

$$f(x; \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} \exp\left(-\frac{\beta}{x}\right)$$

The CDF is

$$F(x; \alpha, \beta) = \Gamma(\alpha, \beta/x)/\Gamma(\alpha)$$

Usage

```
inverse_gamma_distribution(shape, scale = 1)

inverse_gamma_pdf(x, shape, scale = 1)

inverse_gamma_lpdf(x, shape, scale = 1)

inverse_gamma_cdf(x, shape, scale = 1)

inverse_gamma_lcdf(x, shape, scale = 1)

inverse_gamma_quantile(p, shape, scale = 1)
```

Arguments

shape	Shape parameter (shape > 0).
scale	Scale parameter (scale > 0; default is 1).
x	Quantile value (x >= 0).
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Inverse Gamma distribution with shape = 5, scale = 4
dist <- inverse_gamma_distribution(5, 4)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
inverse_gamma_pdf(2, 5, 4)
inverse_gamma_lpdf(2, 5, 4)
inverse_gamma_cdf(2, 5, 4)
inverse_gamma_lcdf(2, 5, 4)
inverse_gamma_quantile(0.5, 5, 4)
```

`inverse_gaussian_distribution` *Inverse Gaussian Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Inverse Gaussian (Inverse Normal) distribution.

With mean μ and scale λ, the PDF is

$$f(x;\mu,\lambda) = \sqrt{\frac{\lambda}{2\pi x^3}} \exp\left(-\frac{\lambda(x-\mu)^2}{2\mu^2 x}\right)$$

and the CDF is

$$F(x;\mu,\lambda) = \Phi\left(\sqrt{\lambda x}(x\mu - 1)\right) + e^{2\mu/\lambda}\Phi\left(-\sqrt{\lambda/\mu}(1 + x/\mu)\right)$$

where Φ is the standard normal CDF.

Accuracy and Implementation Notes: Implemented using the standard normal distribution and the exponential function. `logpdf` is specialised for numerical accuracy. Quantiles have no closed form and are computed via Newton-Raphson refinement.

Usage

```
inverse_gaussian_distribution(mu = 1, lambda = 1)

inverse_gaussian_pdf(x, mu = 1, lambda = 1)

inverse_gaussian_lpdf(x, mu = 1, lambda = 1)

inverse_gaussian_cdf(x, mu = 1, lambda = 1)

inverse_gaussian_lcdf(x, mu = 1, lambda = 1)

inverse_gaussian_quantile(p, mu = 1, lambda = 1)
```

Arguments

`mu`	Mean parameter (mu > 0; default is 1).
`lambda`	Scale parameter (lambda > 0; default is 1).
`x`	Quantile value (x >= 0).
`p`	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Inverse Gaussian distribution with mu = 3, lambda = 4
dist <- inverse_gaussian_distribution(3, 4)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
inverse_gaussian_pdf(2, 3, 4)
inverse_gaussian_lpdf(2, 3, 4)
inverse_gaussian_cdf(2, 3, 4)
inverse_gaussian_lcdf(2, 3, 4)
inverse_gaussian_quantile(0.5, 3, 4)
```

---

inverse_hyperbolic_functions

Inverse Hyperbolic Functions

Description

Functions to compute the inverse hyperbolic functions with high precision and proper handling of edge cases.

- `acosh_boost(x)`: Inverse hyperbolic cosine, $\cosh^{-1}(\cosh(x)) = x$
- `asinh_boost(x)`: Inverse hyperbolic sine, $\sinh^{-1}(\sinh(x)) = x$
- `atanh_boost(x)`: Inverse hyperbolic tangent, $\tanh^{-1}(\tanh(x)) = x$

Usage

```
acosh_boost(x)

asinh_boost(x)

atanh_boost(x)
```

Arguments

x	Input numeric value (domain depends on the function)

Value

A single numeric value with the computed inverse hyperbolic function.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Inverse Hyperbolic Cosine (x >= 1)
acosh_boost(2)  # Returns approximately 1.317
# Inverse Hyperbolic Sine (all real x)
asinh_boost(1)  # Returns approximately 0.881
# Inverse Hyperbolic Tangent (|x| < 1)
atanh_boost(0.5)  # Returns approximately 0.549
```

---

jacobi_elliptic_functions

Jacobi Elliptic Functions

Description

Functions to compute the Jacobi elliptic functions, which are doubly periodic generalizations of trigonometric and hyperbolic functions.

Jacobi Elliptic SN, CN and DN Given:

$$u = \int_0^{\phi} \frac{d\psi}{\sqrt{1 - k^2 \sin^2 \psi}}$$

- `jacobi_sn(k, u)`: $\sin\psi$
- `jacobi_cn(k, u)`: $\cos\psi$
- `jacobi_dn(k, u)`: $\sqrt{(1 - k^2 \sin^2 \psi)}$
- `jacobi_cd(k, u)`: $\frac{cn(k,u)}{dn(k,u)}$

- `jacobi_cs(k, u)`: $\frac{cn(k,u)}{sn(k,u)}$
- `jacobi_dc(k, u)`: $\frac{dn(k,u)}{cn(k,u)}$
- `jacobi_ds(k, u)`: $\frac{dn(k,u)}{sn(k,u)}$
- `jacobi_nc(k, u)`: $\frac{1}{cn(k,u)}$
- `jacobi_nd(k, u)`: $\frac{1}{dn(k,u)}$
- `jacobi_ns(k, u)`: $\frac{1}{sn(k,u)}$
- `jacobi_sc(k, u)`: $\frac{sn(k,u)}{cn(k,u)}$
- `jacobi_sd(k, u)`: $\frac{sn(k,u)}{dn(k,u)}$

Usage

```
jacobi_elliptic(k, u)

jacobi_cd(k, u)

jacobi_cn(k, u)

jacobi_cs(k, u)

jacobi_dc(k, u)

jacobi_dn(k, u)

jacobi_ds(k, u)

jacobi_nc(k, u)

jacobi_nd(k, u)

jacobi_ns(k, u)

jacobi_sc(k, u)

jacobi_sd(k, u)

jacobi_sn(k, u)
```

Arguments

k	Elliptic modulus (typically 0 <= k <= 1, but k > 1 uses complementary relations)
u	Argument of the elliptic functions

Value

For `jacobi_elliptic`, a list containing the values of the Jacobi elliptic functions: `sn`, `cn`, `dn`. For individual functions, a single numeric value is returned.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# All three principal Jacobi Elliptic Functions at once
k <- 0.5
u <- 2
jacobi_elliptic(k, u)
# Individual Jacobi Elliptic Functions
jacobi_cd(k, u)
jacobi_cn(k, u)
jacobi_cs(k, u)
jacobi_dc(k, u)
jacobi_dn(k, u)
jacobi_ds(k, u)
jacobi_nc(k, u)
jacobi_nd(k, u)
jacobi_ns(k, u)
jacobi_sc(k, u)
jacobi_sd(k, u)
jacobi_sn(k, u)
```

---

jacobi_polynomials *Jacobi Polynomials and Related Functions*

Description

Functions to compute Jacobi polynomials, their derivatives, and related functions.

Usage

```
jacobi(n, alpha, beta, x)

jacobi_prime(n, alpha, beta, x)

jacobi_double_prime(n, alpha, beta, x)

jacobi_derivative(n, alpha, beta, x, k)
```

Arguments

n	Degree of the polynomial
alpha	First parameter of the polynomial
beta	Second parameter of the polynomial
x	Argument of the polynomial
k	Order of the derivative

Value

A single numeric value with the computed Jacobi polynomial, its derivative, or k-th derivative.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Jacobi polynomial P_2^(1, 2)(0.5)
jacobi(2, 1, 2, 0.5)
# Derivative of the Jacobi polynomial P_2^(1, 2)'(0.5)
jacobi_prime(2, 1, 2, 0.5)
# Second derivative of the Jacobi polynomial P_2^(1, 2)''(0.5)
jacobi_double_prime(2, 1, 2, 0.5)
# 3rd derivative of the Jacobi polynomial P_2^(1, 2)^(k)(0.5)
jacobi_derivative(2, 1, 2, 0.5, 3)
```

---

jacobi_theta_functions

Jacobi Theta Functions

Description

Functions to compute the four Jacobi theta functions theta_1, theta_2, theta_3, theta_4, which are inter-related periodic functions parameterised by either q (nome) or tau.

Jacobi Theta Function θ_1

- `jacobi_theta1(x, q)`: First Jacobi theta function, nome parameterisation:

$$\theta_1(x, q) = 2\sum_{n=0}^{\infty}(-1)^n q^{(n+\frac{1}{2})^2}\sin((2n+1)x)$$

- `jacobi_theta1tau(x, tau)`: First Jacobi theta function, tau parameterisation:

$$\theta_1(x|\tau) = 2\sum_{n=0}^{\infty}(-1)^n \exp(i\pi\tau(n+0.5)^2)\sin((2n+1)x)$$

Jacobi Theta Function θ_2

- `jacobi_theta2(x, q)`: Second Jacobi theta function, nome parameterisation:

$$\theta_2(x, q) = 2\sum_{n=0}^{\infty} q^{(n+\frac{1}{2})^2}\cos((2n+1)x)$$

- `jacobi_theta2tau(x, tau)`: Second Jacobi theta function, tau parameterisation:

$$\theta_2(x|\tau) = 2\sum_{n=0}^{\infty} \exp(i\pi\tau(n+0.5)^2)\cos((2n+1)x)$$

Jacobi Theta Function θ_3

- `jacobi_theta3(x, q)`: Third Jacobi theta function, nome parameterisation:

$$\theta_3(x,q) = 1 + 2\sum_{n=0}^{\infty} q^{n^2}\cos((2nx)$$

- `jacobi_theta3tau(x, tau)`: Third Jacobi theta function, tau parameterisation:

$$\theta_3(x|\tau) = 1 + 2\sum_{n=0}^{\infty} \exp(i\pi\tau n^2)\cos(2nx)$$

Jacobi Theta Function θ_4

- `jacobi_theta4(x, q)`: Fourth Jacobi theta function, nome parameterisation:

$$\theta_4(x,q) = 1 + 2\sum_{n=0}^{\infty} (-1)^n q^{n^2}\cos((2nx)$$

- `jacobi_theta4tau(x, tau)`: Fourth Jacobi theta function, tau parameterisation:

$$\theta_4(x|\tau) = 1 + 2\sum_{n=0}^{\infty} (-1)^n \exp(i\pi\tau n^2)\cos(2nx)$$

Usage

```
jacobi_theta1(x, q)

jacobi_theta1tau(x, tau)

jacobi_theta2(x, q)

jacobi_theta2tau(x, tau)

jacobi_theta3(x, q)

jacobi_theta3tau(x, tau)

jacobi_theta3m1(x, q)

jacobi_theta3m1tau(x, tau)

jacobi_theta4(x, q)
```

```
jacobi_theta4tau(x, tau)

jacobi_theta4m1(x, q)

jacobi_theta4m1tau(x, tau)
```

Arguments

x	Input value (argument of the theta function)
q	The nome parameter of the Jacobi theta function ($0 < q < 1$)
tau	The nome parameter in tau-form (real-valued, implicitly multiplied by i)

Value

A single numeric value with the computed Jacobi theta function.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Jacobi Theta Functions with q parametrization
x <- 0.5
q <- 0.3  # Note: q should be in (0, 1)
tau <- 1.5
jacobi_theta1(x, q)
jacobi_theta1tau(x, tau)
jacobi_theta2(x, q)
jacobi_theta2tau(x, tau)
jacobi_theta3(x, q)
jacobi_theta3tau(x, tau)
# Special "minus 1" variants for improved accuracy when q is small
jacobi_theta3m1(x, q)
jacobi_theta3m1tau(x, tau)
jacobi_theta4(x, q)
jacobi_theta4tau(x, tau)
jacobi_theta4m1(x, q)
jacobi_theta4m1tau(x, tau)
```

---

kolmogorov_smirnov_distribution

Kolmogorov-Smirnov Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Kolmogorov-Smirnov distribution.

Boost implements the limiting (first-order) Kolmogorov distribution.

The CDF is:

$$\lim_{n\to\infty} F_n(x/\sqrt{n}) = 1 + 2\sum_{k=1}^{\infty}(-1)^k e^{-2k^2x^2}$$

The PDF is obtained by differentiating the CDF, and quantiles are computed via Newton-Raphson iteration.

Accuracy and Implementation Notes: The CDF uses the Jacobi theta function and inherits its accuracy.

Usage

```
kolmogorov_smirnov_distribution(n)

kolmogorov_smirnov_pdf(x, n)

kolmogorov_smirnov_lpdf(x, n)

kolmogorov_smirnov_cdf(x, n)

kolmogorov_smirnov_lcdf(x, n)

kolmogorov_smirnov_quantile(p, n)
```

Arguments

n	Sample size (n > 0).
x	Quantile value (x >= 0).
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Kolmogorov-Smirnov distribution with sample size n = 10
dist <- kolmogorov_smirnov_distribution(10)
# Apply generic functions
cdf(dist, 2)
logcdf(dist, 2)
pdf(dist, 2)
logpdf(dist, 2)
hazard(dist, 2)
chf(dist, 2)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
kolmogorov_smirnov_pdf(0.5, 10)
kolmogorov_smirnov_lpdf(0.5, 10)
kolmogorov_smirnov_cdf(0.5, 10)
kolmogorov_smirnov_lcdf(0.5, 10)
kolmogorov_smirnov_quantile(0.5, 10)
```

---

laguerre_polynomials *Laguerre Polynomials and Related Functions*

Description

Functions to compute Laguerre polynomials and associated Laguerre polynomials.

Laguerre polynomials are orthogonal polynomials that appear in the solution of the quantum harmonic oscillator, hydrogen atom wavefunctions, and various problems in mathematical physics and probability theory.

Standard Laguerre Polynomials L_n(x):

- `laguerre(n, x)`: Evaluates the Laguerre polynomial of degree n at point x
- Solutions to Laguerre's differential equation
- Orthogonal with respect to the weight function e^{-x} on [0, Inf)

Associated Laguerre Polynomials L_n^m(x):

- `laguerre_m(n, m, x)`: Evaluates the associated Laguerre polynomial of degree n and order m at point x

- Generalizations of the standard Laguerre polynomials

Recurrence Relations:

Three-term recurrence relations enable efficient sequential computation:

- `laguerre_next(n, x, Ln, Lnm1)`: Computes $L_{n+1}(x)$ from L_n and L_{n-1}
- `laguerre_next_m(n, m, x, Ln, Lnm1)`: Computes $L^m_{n+1}(x)$ from previous values

Implementation Notes:

Functions use stable three-term recurrence relations that guarantee low absolute error but cannot guarantee low relative error near polynomial roots.

Usage

```
laguerre(n, x)

laguerre_m(n, m, x)

laguerre_next(n, x, Ln, Lnm1)

laguerre_next_m(n, m, x, Ln, Lnm1)
```

Arguments

`n`	Degree of the polynomial
`x`	Argument of the polynomial
`m`	Order of the polynomial (for associated Laguerre polynomials)
`Ln`	Value of the Laguerre polynomial $L_n(x)$
`Lnm1`	Value of the Laguerre polynomial $L_{n-1}(x)$

Value

A single numeric value with the computed Laguerre polynomial.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Laguerre polynomial L_2(0.5)
laguerre(2, 0.5)
# Associated Laguerre polynomial L_2^1(0.5)
laguerre_m(2, 1, 0.5)
# Next Laguerre polynomial L_3(0.5) using L_2(0.5) and L_1(0.5)
laguerre_next(2, 0.5, laguerre(2, 0.5), laguerre(1, 0.5))
# Next associated Laguerre polynomial L_3^1(0.5) using L_2^1(0.5) and L_1^1(0.5)
laguerre_next_m(2, 1, 0.5, laguerre_m(2, 1, 0.5), laguerre_m(1, 1, 0.5))
```

`lambert_w_function` *Lambert W Function and Its Derivatives*

Description

Functions to compute the Lambert W function and its derivatives for the principal branch (W_0) and the branch -1 (W_-_1).

The Lambert W function is the solution of:

$$W(z) \cdot e^{W(z)} = z$$

Branches:

The function has two real branches:

- **W_0 (Principal Branch):**
  - `lambert_w0(z)`: Returns the principal branch value
  - `lambert_w0_prime(z)`: Returns the derivative of W_0
  - For z >= 0, there is a single real solution
- **W_-_1 (Secondary Branch):**
  - `lambert_wm1(z)`: Returns the -1 branch value
  - `lambert_wm1_prime(z)`: Returns the derivative of W_-_1
  - Exists where two real solutions occur on (-1/e, 0)
  - As z approaches 0, W_-_1(z) approaches -Inf

Usage

```
lambert_w0(z)

lambert_wm1(z)

lambert_w0_prime(z)

lambert_wm1_prime(z)
```

Arguments

z	Argument of the Lambert W function

Value

A single numeric value with the computed Lambert W function or its derivative.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Lambert W Function (Principal Branch)
lambert_w0(0.3)
# Lambert W Function (Branch -1)
lambert_wm1(-0.3)
# Derivative of the Lambert W Function (Principal Branch)
lambert_w0_prime(0.3)
# Derivative of the Lambert W Function (Branch -1)
lambert_wm1_prime(-0.3)
```

---

landau_distribution *Landau Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Landau distribution.

The PDF is:

$$f(x;\mu,c) = \frac{1}{\pi c}\int_0^\infty \exp(-t)\cos\left(t\left(\frac{x-\mu}{c}\right) + \frac{2t}{\pi}\log\left(\frac{t}{c}\right)\right)dt$$

Usage

```
landau_distribution(location = 0, scale = 1)

landau_pdf(x, location = 0, scale = 1)

landau_lpdf(x, location = 0, scale = 1)

landau_cdf(x, location = 0, scale = 1)

landau_lcdf(x, location = 0, scale = 1)

landau_quantile(p, location = 0, scale = 1)
```

Arguments

location	Location parameter (default is 0).
scale	Scale parameter (default is 1).
x	Quantile value.
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Landau distribution with location 0 and scale 1
dist <- landau_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
support(dist)

# Convenience functions
landau_pdf(3)
landau_lpdf(3)
landau_cdf(3)
landau_lcdf(3)
landau_quantile(0.5)
```

---

laplace_distribution *Laplace Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Laplace (double exponential) distribution.

The PDF is

$$f(x;\mu,\sigma) = \frac{1}{2\sigma}\exp\left(-\frac{|x-\mu|}{\sigma}\right)$$

and the CDF is

$$F(x) = \begin{cases} \exp\left(\frac{x-\mu}{\sigma}\right)/\sigma, & x < \mu, \\ 1 - \exp\left(\frac{x-\mu}{\sigma}\right)/\sigma, & x \geq \mu. \end{cases}$$

Usage

```
laplace_distribution(location = 0, scale = 1)

laplace_pdf(x, location = 0, scale = 1)

laplace_lpdf(x, location = 0, scale = 1)

laplace_cdf(x, location = 0, scale = 1)

laplace_lcdf(x, location = 0, scale = 1)

laplace_quantile(p, location = 0, scale = 1)
```

Arguments

location	Location parameter (default is 0).
scale	Scale parameter (default is 1).
x	Quantile value.
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Laplace distribution with location = 0, scale = 1
dist <- laplace_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
```

```
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
laplace_pdf(0)
laplace_lpdf(0)
laplace_cdf(0)
laplace_lcdf(0)
laplace_quantile(0.5)
```

`legendre_polynomials` *Legendre Polynomials and Related Functions*

Description

Functions to compute Legendre polynomials of the first and second kind, their derivatives, zeros, and related functions.

Legendre polynomials are orthogonal polynomials that are solutions to Legendre's differential equation. They appear in physics (multipole expansions, solutions to Laplace's equation in spherical coordinates) and numerical analysis (Gaussian quadrature).

Legendre Polynomials of the First Kind P_n(x):

Standard solutions to the Legendre differential equation.

- Domain: -1 <= x <= 1 (domain error outside this range)
- Reflection formula: $P_{-l-1}(x) = P_l(x)$
- `legendre_p(n, x)`: Evaluates P_n(x)
- `legendre_p_prime(n, x)`: Derivative of P_n(x)
- `legendre_p_zeros(n)`: Returns zeros in increasing order. For odd n, first zero is 0. Computed using Newton's method with Tricomi's initial estimates (O(n^2) complexity)

Associated Legendre Polynomials P_n^m(x):

- `legendre_p_m(n, m, x)`: Evaluates P_n^m(x)
- Includes Condon-Shortley phase term (-1)^m matching Abramowitz & Stegun definition
- Negative values of n and m handled through identity relations

Legendre Polynomials of the Second Kind Q_n(x):

Second solution to the Legendre differential equation.

- `legendre_q(n, x)`: Evaluates Q_n(x)
- Domain: -1 <= x <= 1 (domain error otherwise)

Recurrence Relations:

Efficient computation using three-term recurrence at fixed x with increasing degree:

- `legendre_next(n, x, Pl, Plm1)`: Computes $P_{n+1}(x)$ from P_n and P_{n-1}
- `legendre_next_m(n, m, x, Pl, Plm1)`: Computes $P^m_{n+1}(x)$ from previous values

Recurrence relations guarantee low absolute error but cannot guarantee low relative error near polynomial roots.

Usage

```
legendre_p(n, x)

legendre_p_prime(n, x)

legendre_p_zeros(n)

legendre_p_m(n, m, x)

legendre_q(n, x)

legendre_next(n, x, Pl, Plm1)

legendre_next_m(n, m, x, Pl, Plm1)
```

Arguments

n	Degree of the polynomial
x	Argument of the polynomial (must be in $[-1, 1]$)
m	Order of the polynomial (for associated Legendre polynomials)
Pl	Value of the Legendre polynomial P_l(x)
Plm1	Value of the Legendre polynomial $P_{l-1}(x)$

Value

A single numeric value with the computed Legendre polynomial, its derivative, or zeros (as a vector).

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Legendre polynomial of the first kind P_2(0.5)
legendre_p(2, 0.5)
# Derivative of the Legendre polynomial of the first kind P_2'(0.5)
legendre_p_prime(2, 0.5)
# Zeros of the Legendre polynomial of the first kind P_2
legendre_p_zeros(2)
# Associated Legendre polynomial P_2^1(0.5)
legendre_p_m(2, 1, 0.5)
# Legendre polynomial of the second kind Q_2(0.5)
legendre_q(2, 0.5)
# Next Legendre polynomial of the first kind P_3(0.5) using P_2(0.5) and P_1(0.5)
legendre_next(2, 0.5, legendre_p(2, 0.5), legendre_p(1, 0.5))
# Next associated Legendre polynomial P_3^1(0.5) using P_2^1(0.5) and P_1^1(0.5)
legendre_next_m(2, 1, 0.5, legendre_p_m(2, 1, 0.5), legendre_p_m(1, 1, 0.5))
```

`linear_regression` *Linear Regression Functions*

Description

Functions to perform simple ordinary least squares (OLS) linear regression.

The OLS fit finds c_0 and c_1 that minimize

$$\mathcal{L}(c_0, c_1) := \sum_{i=0}^{n-1} (y_i - c_0 - c_1 x_i)^2$$

producing the model $f(x) = c_0 + c_1 x$. The optional R^2 output uses

$$R^2 = 1 - \frac{\sum_i (y_i - c_0 - c_1 x_i)^2}{\sum_i (y_i - \bar{y})^2}$$

Usage

```
simple_ordinary_least_squares(x, y)

simple_ordinary_least_squares_with_R_squared(x, y)
```

Arguments

`x`	A numeric vector.
`y`	A numeric vector.

Value

A two-element numeric vector containing the intercept and slope of the regression line, or a three-element vector containing the intercept, slope, and R-squared value if applicable.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Simple Ordinary Least Squares
x <- c(1, 2, 3, 4, 5)
y <- c(2, 3, 5, 7, 11)
simple_ordinary_least_squares(x, y)
# Simple Ordinary Least Squares with R-squared
simple_ordinary_least_squares_with_R_squared(x, y)
```

`ljung_box_test` *Ljung-Box Test for Autocorrelation*

Description

Functions to perform the Ljung-Box test for autocorrelation in residuals.

The test statistic is

$$Q := n(n+2)\sum_{k=1}^{\ell}\frac{\hat{r}_k^2}{n-k}$$

Where:

$$\hat{r}_k := \frac{\sum_{i=k}^{n-1}(v_i-\bar{v})(v_{i-k}-\bar{v})}{\sum_{i=0}^{n-1}(v_i-\bar{v})^2}$$

Where: n is the sample size (length of v) and ℓ is the number of lags

Usage

```
ljung_box(v, lags = -1, fit_dof = 0)
```

Arguments

`v`	A numeric vector.
`lags`	A single integer value (default uses $\log(n)$).
`fit_dof`	A single integer value.

Value

A two-element numeric vector containing the test statistic and the p-value.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Ljung-Box test for autocorrelation
ljung_box(c(1, 2, 3, 4, 5), lags = 2, fit_dof = 0)
```

`logistic_distribution` *Logistic Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Logistic distribution.

With location u and scale $s>0$, the PDF and CDF are

$$f(x;\mu,s) = \frac{e^{-(x-u)/s}}{s\left(1+e^{-(x-u)/s}\right)^2}$$

$$F(x;\mu,s) = \frac{1}{1+e^{-(x-u)/s}}$$

and the quantile is

$$F^{-1}(p;\mu,s) = \mu + s\log\left(\frac{p}{1-p}\right)$$

Usage

```
logistic_distribution(location = 0, scale = 1)

logistic_pdf(x, location = 0, scale = 1)

logistic_lpdf(x, location = 0, scale = 1)

logistic_cdf(x, location = 0, scale = 1)

logistic_lcdf(x, location = 0, scale = 1)

logistic_quantile(p, location = 0, scale = 1)
```

Arguments

`location`	Location parameter (default is 0).
`scale`	Scale parameter (default is 1).
`x`	Quantile value.
`p`	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Logistic distribution with location = 0, scale = 1
dist <- logistic_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
logistic_pdf(0)
logistic_lpdf(0)
logistic_cdf(0)
logistic_lcdf(0)
logistic_quantile(0.5)
```

---

logistic_functions *Logistic Functions*

Description

Functions to compute the logistic sigmoid function and its inverse, the logit function.

These functions are fundamental in statistics, machine learning, and probability theory, particularly in logistic regression and neural networks.

Logistic Sigmoid Function:

- `logistic_sigmoid(x)`: $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$

Logit Function:

- `logit(x)`:

$$logit(x) = \log\left(\frac{x}{1-x}\right)$$

Usage

```
logistic_sigmoid(x)

logit(x)
```

Arguments

x	Numeric value for which to compute the functions

Value

A single numeric value with the computed logit or logistic sigmoid function.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Logistic Sigmoid Function
logistic_sigmoid(0)     # Returns 0.5
logistic_sigmoid(2)     # Returns ~0.881
logistic_sigmoid(-2)    # Returns ~0.119

# Logit Function (inverse of sigmoid)
logit(0.5)              # Returns 0
logit(0.7)              # Returns ~0.847
logit(0.881)            # Returns ~2 (inverse of sigmoid(2))
```

---

lognormal_distribution

Log Normal Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Log Normal distribution.

The PDF is:

$$f(x;\mu,\sigma^2) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right), \quad x > 0$$

The CDF is:

$$F(x;\mu,\sigma^2) = \Phi\left(\frac{\ln x - \mu}{\sigma}\right)$$

The Quantile is:

$$F^{-1}(p;\mu,\sigma^2) = \exp(\mu + \sigma\Phi^{-1}(p))$$

Usage

```
lognormal_distribution(location = 0, scale = 1)

lognormal_pdf(x, location = 0, scale = 1)

lognormal_lpdf(x, location = 0, scale = 1)

lognormal_cdf(x, location = 0, scale = 1)

lognormal_lcdf(x, location = 0, scale = 1)

lognormal_quantile(p, location = 0, scale = 1)
```

Arguments

location	Location parameter (default is 0).
scale	Scale parameter (default is 1).
x	Quantile value (x > 0).
p	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Log Normal distribution with location = 0, scale = 1
dist <- lognormal_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
```

```
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
lognormal_pdf(0)
lognormal_lpdf(0)
lognormal_cdf(0)
lognormal_lcdf(0)
lognormal_quantile(0.5)
```

`makima` *Modified Akima Interpolator*

Description

The modified Akima interpolant takes non-equispaced data and interpolates between them via cubic Hermite polynomials whose slopes are chosen significantly.

Properties:

The slopes are chosen by a modification of a geometric construction proposed by Akima. The interpolant is C1 and evaluation has O(log N) complexity. It oscillates less than the cubic spline but has less smoothness. The modification is given by Cosmin Ionita and agrees with Matlab's version.

Usage

```
makima(x, y, left_endpoint_derivative = NULL, right_endpoint_derivative = NULL)
```

Arguments

`x`	Numeric vector of abscissas (x-coordinates).
`y`	Numeric vector of ordinates (y-coordinates).
`left_endpoint_derivative`	Optional numeric value of the derivative at the left endpoint.
`right_endpoint_derivative`	Optional numeric value of the derivative at the right endpoint.

Value

An object of class `makima` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `push_back(x, y)`: Add a new control point

Examples

```
x <- c(0, 1, 2, 3)
y <- c(0, 1, 0, 1)
interpolator <- makima(x, y)
xi <- 0.5
interpolator$interpolate(xi)
interpolator$prime(xi)
interpolator$push_back(4, 1)
```

`mapairy_distribution` *Map-Airy Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Map-Airy distribution.

The PDF is:

$$f(x; \mu, c) = 2 \exp\left(\frac{2}{3}x^3\right) \left(-x\mathrm{Ai}(x^2) - \mathrm{Ai}'(x^2)\right)$$

Usage

```
mapairy_distribution(location = 0, scale = 1)

mapairy_pdf(x, location = 0, scale = 1)

mapairy_lpdf(x, location = 0, scale = 1)

mapairy_cdf(x, location = 0, scale = 1)

mapairy_lcdf(x, location = 0, scale = 1)

mapairy_quantile(p, location = 0, scale = 1)
```

Arguments

`location`	Location parameter (default is 0).
`scale`	Scale parameter (default is 1).
`x`	Quantile value.
`p`	Probability (0 <= p <= 1).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Map-Airy distribution with location 0 and scale 1
dist <- mapairy_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)

# Convenience functions
mapairy_pdf(3)
mapairy_lpdf(3)
mapairy_cdf(3)
mapairy_lcdf(3)
mapairy_quantile(0.5)
```

---

negative_binomial_distribution

Negative Binomial Distribution Functions

Description

Functions to compute the probability mass function (pmf), cumulative distribution function, and quantile function for the Negative Binomial distribution.

The PDF is:

$$f(k; r, p) = \frac{\Gamma(r+k)}{k!\Gamma(r)} p^r (1-p)^k$$

The CDF is:

$$F(k; r, p) = I_p(r, k+1)$$

Where I_p is the regularised incomplete beta function.

Usage

```
negative_binomial_distribution(successes, success_fraction)

negative_binomial_pdf(x, successes, success_fraction)

negative_binomial_lpdf(x, successes, success_fraction)

negative_binomial_cdf(x, successes, success_fraction)

negative_binomial_lcdf(x, successes, success_fraction)

negative_binomial_quantile(p, successes, success_fraction)

negative_binomial_find_lower_bound_on_p(trials, successes, alpha)

negative_binomial_find_upper_bound_on_p(trials, successes, alpha)

negative_binomial_find_minimum_number_of_trials(
  failures,
  success_fraction,
  alpha
)

negative_binomial_find_maximum_number_of_trials(
  failures,
  success_fraction,
  alpha
)
```

Arguments

`successes`	Number of successes (successes > 0).
`success_fraction`	Probability of success on each trial (0 <= success_fraction <= 1).
`x`	Quantile value.
`p`	Probability (0 <= p <= 1).
`trials`	Number of trials.
`alpha`	Significance level (0 < alpha < 1).
`failures`	Number of failures (failures >= 0).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Negative Binomial distribution with successes = 5, success_fraction = 0.5
dist <- negative_binomial_distribution(5, 0.5)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
negative_binomial_pdf(3, 5, 0.5)
negative_binomial_lpdf(3, 5, 0.5)
negative_binomial_cdf(3, 5, 0.5)
negative_binomial_lcdf(3, 5, 0.5)
negative_binomial_quantile(0.5, 5, 0.5)

## Not run:
# Find lower bound on p given 10 trials and 5 successes with 95% confidence
negative_binomial_find_lower_bound_on_p(10, 5, 0.05)
# Find upper bound on p given 10 trials and 5 successes with 95% confidence
negative_binomial_find_upper_bound_on_p(10, 5, 0.05)
# Find minimum number of trials to observe 3 failures with success fraction 0.5 at 95% confidence
negative_binomial_find_minimum_number_of_trials(3, 0.5, 0.05)
# Find maximum number of trials to observe 3 failures with success fraction 0.5 at 95% confidence
negative_binomial_find_maximum_number_of_trials(3, 0.5, 0.05)

## End(Not run)
```

---

non_central_beta_distribution

Noncentral Beta Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Noncentral Beta distribution.

The noncentral beta distribution is a generalization of the Beta Distribution.

The PDF is:

$$f(x;\alpha,\beta,\lambda) = \sum_{i=0}^{\infty} P(i;\lambda/2) I'_x(\alpha+i,\beta)$$

where $P(i;\lambda/2)$ is the discrete Poisson probability at i, with mean $\lambda/2$, and $I'_x(\alpha,\beta)$ is the derivative of the incomplete beta function.

The CDF is:

$$F(x;\alpha,\beta,\lambda) = \sum_{i=0}^{\infty} P(i;\lambda/2) I_x(\alpha+i,\beta)$$

where $I_x(\alpha,\beta)$ is the incomplete beta function.

Usage

```
non_central_beta_distribution(alpha, beta, lambda)

non_central_beta_pdf(x, alpha, beta, lambda)

non_central_beta_lpdf(x, alpha, beta, lambda)

non_central_beta_cdf(x, alpha, beta, lambda)

non_central_beta_lcdf(x, alpha, beta, lambda)

non_central_beta_quantile(p, alpha, beta, lambda)
```

Arguments

`alpha`	first shape parameter (alpha > 0)
`beta`	second shape parameter (beta > 0)
`lambda`	noncentrality parameter (lambda >= 0)
`x`	quantile (0 <= x <= 1)
`p`	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Noncentral Beta distribution with shape parameters alpha = 2, beta = 3
# and noncentrality parameter lambda = 1
dist <- non_central_beta_distribution(2, 3, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)

# Convenience functions
non_central_beta_pdf(0.5, 2, 3, 1)
non_central_beta_lpdf(0.5, 2, 3, 1)
non_central_beta_cdf(0.5, 2, 3, 1)
non_central_beta_lcdf(0.5, 2, 3, 1)
non_central_beta_quantile(0.5, 2, 3, 1)
```

---

non_central_chi_squared_distribution

Noncentral Chi-Squared Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Noncentral Chi-Squared distribution.

The PDF is:

$$f(x; \nu, \lambda) = \sum_{k=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^k}{k!} f(x; \nu + 2k)$$

where $f(x; \nu)$ is the central chi-squared distribution PDF.

The CDF is:

$F(x; \nu, \lambda) = e^{-\lambda/2} \sum_{k=0}^{\infty} \frac{(\lambda/2)^k}{k!} F(x; \nu + 2k)$

Where $F(x; \nu)$ is the central chi-squared distribution CDF.

Usage

```
non_central_chi_squared_distribution(df, lambda)

non_central_chi_squared_pdf(x, df, lambda)

non_central_chi_squared_lpdf(x, df, lambda)

non_central_chi_squared_cdf(x, df, lambda)

non_central_chi_squared_lcdf(x, df, lambda)

non_central_chi_squared_quantile(p, df, lambda)

non_central_chi_squared_find_degrees_of_freedom(lambda, x, alpha)

non_central_chi_squared_find_non_centrality(df, x, alpha)
```

Arguments

df	degrees of freedom (df > 0)
lambda	noncentrality parameter (lambda >= 0)
x	quantile
p	probability (0 <= p <= 1)
alpha	The acceptable probability of a Type I error (false positive).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
## Not run:
# Noncentral Chi-Squared distribution with 3 degrees of freedom and noncentrality
# parameter 1
dist <- non_central_chi_squared_distribution(3, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
```

```
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
non_central_chi_squared_pdf(2, 3, 1)
non_central_chi_squared_lpdf(2, 3, 1)
non_central_chi_squared_cdf(2, 3, 1)
non_central_chi_squared_lcdf(2, 3, 1)
non_central_chi_squared_quantile(0.5, 3, 1)

# Find degrees of freedom needed for CDF at 2.0 with noncentrality parameter 1.0 = 0.05
non_central_chi_squared_find_degrees_of_freedom(1.0, 2.0, 0.05)
# Find noncentrality parameter needed for CDF at 2.0 with 3 degrees of freedom = 0.05
non_central_chi_squared_find_non_centrality(3, 2.0, 0.05)

## End(Not run)
```

`non_central_f_distribution`

Noncentral F Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Noncentral F distribution.

The noncentral F distribution is a generalization of the Fisher F Distribution.

Thje PDF is:

$$f(x;\nu_1,\nu_2,\lambda) = \sum_{k=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^k}{B\left(\frac{\nu_2}{2},\frac{\nu_1}{2}+k\right)k!} \left(\frac{\nu_1}{\nu_2}\right)^{\frac{\nu_1}{2}+k} \left(\frac{\nu_2}{\nu_2+\nu_1 x}\right)^{\frac{\nu_1+\nu_2}{2}+k} x^{\nu_1/2-1+k}$$

The CDF is:

$$F(x;d_1,d_2,\lambda) = \sum_{j=0}^{\infty} \left(\frac{\left(\frac{1}{2}\lambda\right)^j}{j!}e^{-\lambda/2}\right) I\left(\frac{d_1 x}{d_2+d_1 x}\middle|\frac{d_1}{2}+j,\frac{d_2}{2}\right)$$

Usage

```
non_central_f_distribution(df1, df2, lambda)

non_central_f_pdf(x, df1, df2, lambda)

non_central_f_lpdf(x, df1, df2, lambda)

non_central_f_cdf(x, df1, df2, lambda)

non_central_f_lcdf(x, df1, df2, lambda)

non_central_f_quantile(p, df1, df2, lambda)
```

Arguments

df1	degrees of freedom for the numerator (df1 > 0)
df2	degrees of freedom for the denominator (df2 > 0)
lambda	noncentrality parameter (lambda >= 0)
x	quantile
p	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Noncentral F distribution with df1 = 10, df2 = 10 and noncentrality
# parameter 1
dist <- non_central_f_distribution(10, 10, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
```

```
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
non_central_f_pdf(1, 5, 2, 1)
non_central_f_lpdf(1, 5, 2, 1)
non_central_f_cdf(1, 5, 2, 1)
non_central_f_lcdf(1, 5, 2, 1)
non_central_f_quantile(0.5, 5, 2, 1)
```

`non_central_t_distribution`

Noncentral T Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Noncentral T distribution.

The noncentral T distribution is a generalization of the Student's t Distribution.

The PDF is:

$$f(x;\nu;\delta) = \frac{\nu^{\nu/2}\,\nu!}{2^{\nu}e^{\delta^2/2}(\nu+x^2)^{\nu/2}\Gamma\left(\frac{\nu}{2}\right)}\left(\frac{\sqrt{2}\,\delta x\,{}_1F_1\left(\frac{\nu}{2}+1;\frac{3}{2};\frac{\delta^2x^2}{2(\nu+x^2)}\right)}{(\nu+x^2)\Gamma\left(\frac{\nu+1}{2}\right)} + \frac{{}_1F_1\left(\frac{\nu+1}{2};\frac{1}{2};\frac{\delta^2x^2}{2(\nu+x^2)}\right)}{\sqrt{\nu+x^2}\Gamma\left(\frac{\nu}{2}+1\right)}\right)$$

The CDF is:

$$F(t;\nu;\delta) = \Phi(-\delta) + \frac{1}{2}\sum_{i=0}^{\infty}\left(P_i\,I_x\left(i+\frac{1}{2},\frac{\nu}{2}\right) + \frac{\delta}{\sqrt{2}}\,Q_i\,I_x\left(i+1,\frac{\nu}{2}\right)\right)$$

Where:

$$P_i = e^{-\delta^2/2}\frac{\left(\delta^2/2\right)^i}{i!}, \qquad Q_i = e^{-\delta^2/2}\frac{\left(\delta^2/2\right)^i}{\Gamma\left(i+\frac{3}{2}\right)}, \qquad x = \frac{t^2}{\nu+t^2}$$

Usage

```
non_central_t_distribution(df, delta)

non_central_t_pdf(x, df, delta)

non_central_t_lpdf(x, df, delta)
```

```
non_central_t_cdf(x, df, delta)

non_central_t_lcdf(x, df, delta)

non_central_t_quantile(p, df, delta)
```

Arguments

df	degrees of freedom (df > 0)
delta	noncentrality parameter (delta >= 0)
x	quantile
p	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Noncentral T distribution with 5 degrees of freedom and noncentrality parameter 1
dist <- non_central_t_distribution(5, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
non_central_t_pdf(0, 5, 1)
non_central_t_lpdf(0, 5, 1)
non_central_t_cdf(0, 5, 1)
non_central_t_lcdf(0, 5, 1)
non_central_t_quantile(0.5, 5, 1)
```

`normal_distribution` *Normal Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Normal distribution.

The normal distribution is probably the most well known statistical distribution: it is also known as the Gaussian Distribution. A normal distribution with mean zero and standard deviation one is known as the Standard Normal Distribution.

Given mean μ and standard deviation σ, it has the PDF:

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The cumulative distribution function is given by:

$$F(x; \mu, \sigma) = \int_{-\infty}^{x} f(t; \mu, \sigma) dt = \frac{1}{2} \text{erfc}\left(\frac{-(x-\mu)}{\sigma\sqrt{2}}\right)$$

Usage

```
normal_distribution(mean = 0, sd = 1)

normal_pdf(x, mean = 0, sd = 1)

normal_lpdf(x, mean = 0, sd = 1)

normal_cdf(x, mean = 0, sd = 1)

normal_lcdf(x, mean = 0, sd = 1)

normal_quantile(p, mean = 0, sd = 1)
```

Arguments

`mean`	mean parameter (default is 0)
`sd`	standard deviation parameter (default is 1)
`x`	quantile
`p`	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Normal distribution with mean = 0, sd = 1
dist <- normal_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
normal_pdf(0)
normal_lpdf(0)
normal_cdf(0)
normal_lcdf(0)
normal_quantile(0.5)
```

---

number_series *Number Series*

Description

Functions to compute Bernoulli numbers, Tangent numbers, Prime numbers, and Fibonacci numbers. The library provides efficient implementations using table lookups for smaller indices and advanced algorithms for larger values.

Bernoulli Numbers B(2n):

The Bernoulli numbers are a sequence of rational numbers useful for Taylor series expansions, the Euler-Maclaurin formula, and the Riemann zeta function.

- `bernoulli_b2n(n)`: Returns the (2n)-th Bernoulli number B_{2n}. Note that odd Bernoulli numbers are 0 (except B_1 = -1/2).
- `max_bernoulli_b2n()`: Returns the largest n such that B_{2n} can be represented in the return type.

- `unchecked_bernoulli_b2n(n)`: A faster version without overflow checks.
- `bernoulli_b2n(start_index, number_of_bernoullis_b2n)`: Computes a range of Bernoulli numbers.

Tangent Numbers T(n):

Tangent numbers (or zag functions) appear in the Maclaurin series of tan(x).

- `tangent_t2n(n)`: Returns the n-th Tangent number.
- `tangent_t2n(start_index, number_of_tangent_t2n)`: Computes a range of Tangent numbers.

Prime Numbers:

Fast table lookup for the first 10,000 prime numbers.

- `prime(n)`: Returns the n-th prime number (0-indexed, so prime(0) = 2).
- `max_prime()`: Returns the maximum index n supported (currently 10,000).

Fibonacci Numbers F(n):

Computes Fibonacci numbers defined by $F_n = F_{n-1} + F_{n-2}$ with $F_0 = 0, F_1 = 1$.

- `fibonacci(n)`: Returns the n-th Fibonacci number.
- `unchecked_fibonacci(n)`: A faster version without overflow checks.

Implementation uses table lookup for small values and iterative algorithms for larger values.

Usage

```
bernoulli_b2n(n = NULL, start_index = NULL, number_of_bernoullis_b2n = NULL)

max_bernoulli_b2n()

unchecked_bernoulli_b2n(n)

tangent_t2n(n = NULL, start_index = NULL, number_of_tangent_t2n = NULL)

prime(n)

max_prime()

fibonacci(n)

unchecked_fibonacci(n)
```

Arguments

`n`	Index of the number to compute (must be a non-negative integer).
`start_index`	The starting index for computing a range of numbers.
`number_of_bernoullis_b2n`	The number of Bernoulli numbers to compute in the range.
`number_of_tangent_t2n`	The number of Tangent numbers to compute in the range.

Value

A single numeric or integer value for scalar inputs, or a vector for range computations. For `max_` functions, returns the maximum supported index.

See Also

Boost Documentation

Examples

```
## Not run:
# 10th Bernoulli number B_20 (index is doubled)
bernoulli_b2n(10)
# Maximum supported index for Bernoulli numbers
max_bernoulli_b2n()
# Range of Bernoulli numbers B_0, B_2, ..., B_18 (10 numbers)
bernoulli_b2n(start_index = 0, number_of_bernoullis_b2n = 10)

# 10th Tangent number
tangent_t2n(10)

# 10th Prime number (0-indexed)
prime(10)

# 10th Fibonacci number
fibonacci(10)

## End(Not run)
```

`numerical_differentiation` *Numerical Differentiation*

Description

Functions for numerical differentiation using finite difference and complex step methods.

Finite Difference Derivative: Calculates a finite-difference approximation to the derivative of a function f at point x. This problem is ill-conditioned: truncation error ($O(h^k)$) decreases with h, but roundoff error increases. The function balances these errors automatically. The default order is 6. Requires the function to be differentiable (up to the order requested).

Complex Step Derivative: Computes the derivative of a real-valued function $f(x)$ using the complex step approximation:

$$f'(x) \approx \frac{\Im(f(x+ih))}{h}$$

This method avoids the subtractive cancellation error inherent in finite differences and is extremely accurate. However, it requires f to be a holomorphic function (complex-differentiable) that takes real values at real arguments. Ideally, the function `f` should be able to accept a complex argument.

Usage

```
finite_difference_derivative(f, x, order = 1)

complex_step_derivative(f, x)
```

Arguments

f	A function to differentiate. It should accept a single numeric/complex value and return a single numeric/complex value.
x	The point at which to evaluate the derivative.
order	The order of accuracy of the finite difference method. Can be 1, 2, 4, 6, or 8. Default is 1.

Value

The approximate value of the derivative at the point x.

See Also

numerical_integration

Examples

```
# Finite difference derivative of sin(x) at pi/4
finite_difference_derivative(sin, pi / 4)

# Complex step derivative of exp(x) at 1.7 (Requires f to handle complex input ideally)
# Note: In pure R, `exp` handles complex numbers automatically.
complex_step_derivative(exp, 1.7)
```

---

numerical_integration *Numerical Integration*

Description

Functions for numerical integration using Trapezoidal, Gauss-Legendre, and Gauss-Kronrod methods.

Trapezoidal Quadrature: Calculates the integral of a function f using the trapezoidal rule. If the integrand is periodic and integrated over a full period, the trapezoidal rule converges faster than any power of the step size h (exponential convergence). For non-periodic twice continuously differentiable functions, the error is $O(h^2)$. Checks for convergence by halving the interval until the tolerance is met or max_refinements is reached. Useful for periodic functions, bump functions, and bell-shaped integrals over infinite intervals.

Gauss-Legendre Quadrature: Performs "one-shot" non-adaptive integration on (a, b) using a fixed number of points. Very efficient for smooth "bell-like" functions and functions with rapidly convergent power series. Does not provide an error estimate.

Gauss-Kronrod Quadrature: An adaptive extension of Gaussian quadrature. Adds $n + 1$ nodes (Kronrod points) to an n-point Gaussian quadrature to provide an *a posteriori* error estimate ($O(h^{3n+1})$ vs $O(h^{2n-1})$). Preserves exponential convergence for smooth functions. Best suited for smooth functions with no end-point singularities.

Usage

```
trapezoidal(f, a, b, tol = sqrt(.Machine$double.eps), max_refinements = 12)

gauss_legendre(f, a, b, points = 7)

gauss_kronrod(
  f,
  a,
  b,
  points = 15,
  max_depth = 15,
  tol = sqrt(.Machine$double.eps)
)
```

Arguments

`f`	A function to integrate. It should accept a single numeric value and return a single numeric value.
`a`	The lower limit of integration.
`b`	The upper limit of integration.
`tol`	The tolerance for the approximation. For trapezoidal, default is `sqrt(.Machine$double.eps)`. For Gauss-Kronrod, default is `sqrt(.Machine$double.eps)`.
`max_refinements`	The maximum number of refinements to apply. Default is 12.
`points`	The number of evaluation points to use in the Gauss-Legendre or Gauss-Kronrod quadrature.
`max_depth`	Sets the maximum number of interval splittings for Gauss-Kronrod permitted before stopping. Set this to zero for non-adaptive quadrature.

Value

A single numeric value with the computed integral.

See Also

numerical_differentiation, double_exponential_quadrature

Examples

```
# Trapezoidal rule integration of sin(x) from 0 to pi (Periodic over 0 to 2*pi)
trapezoidal(sin, 0, pi)
```

```
# Gauss-Legendre integration of exp(x) from 0 to 1
gauss_legendre(exp, 0, 1, points = 7)

# Adaptive Gauss-Kronrod integration of log(x) from 1 to 2
gauss_kronrod(log, 1, 2, points = 15, max_depth = 10)
```

`ooura_fourier_integrals` *Ooura Fourier Integrals*

Description

Computes Fourier sine and cosine integrals using Ooura's robust double exponential method (1999). These methods are designed to handle oscillatory integrals that are problematic for standard quadrature.

Ooura Fourier Sine: Computes the integral:

$$\int_0^\infty f(t)\sin(\omega t)dt$$

Ooura Fourier Cosine: Computes the integral:

$$\int_0^\infty f(t)\cos(\omega t)dt$$

Usage

```
ooura_fourier_sin(
  f,
  omega = 1,
  relative_error_tolerance = sqrt(.Machine$double.eps),
  levels = 8
)

ooura_fourier_cos(
  f,
  omega = 1,
  relative_error_tolerance = sqrt(.Machine$double.eps),
  levels = 8
)
```

Arguments

`f`	A function to integrate. It should accept a single numeric value and return a single numeric value.
`omega`	The frequency parameter ω for the sine or cosine term.
`relative_error_tolerance`	The relative error tolerance for the approximation. Default is `sqrt(.Machine$double.eps)`.
`levels`	The number of levels of refinement to apply. Default is 8.

Details

The method precomputes nodes and weights for efficiency. Convergence depends on the position of the poles of the integrand in the complex plane. If poles are too close to the real axis, convergence may be slow.

Value

A single numeric value with the computed Fourier sine or cosine integral.

References

Ooura, Takuya, and Masatake Mori. "A robust double exponential formula for Fourier-type integrals." Journal of computational and applied mathematics 112.1-2 (1999): 229-241.

See Also

double_exponential_quadrature

Examples

```
# Fourier sine integral of 1/x -> integral convergent to pi/2 approx
# sin(x)/x from 0 to Inf is pi/2. Here we integrate 1/x * sin(1*x).
ooura_fourier_sin(function(x) { 1 / x }, omega = 1)

# Fourier cosine integral of 1/(x^2 + 1) * cos(x)
# Expected value is pi/(2*e) approx 0.57786
ooura_fourier_cos(function(x) { 1/ (x * x + 1) }, omega = 1)
```

---

owens_t *Owens T Function*

Description

Computes Owen's T function T(h, a), which gives the probability of the event (X > h and 0 < Y < a*X) where X and Y are independent standard normal random variables.

$$T(h,a) = \frac{1}{2\pi}\int_0^a \frac{\exp\{-\frac{1}{2}h^2(1+x^2)\}}{1+x^2}\,dx, \quad (-\infty < h, a < +\infty)$$

Usage

```
owens_t(h, a)
```

Arguments

h	The first argument of the Owens T function (boundary parameter)
a	The second argument of the Owens T function (slope parameter)

Value

The value of the Owens T function at (h, a).

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Owens T Function
owens_t(1, 0.5)
```

`pareto_distribution` *Pareto Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Pareto distribution.

The PDF is:

$$f(x; \alpha, \beta) = \frac{\alpha \beta^{\alpha}}{x^{\alpha+1}}$$

The CDF is:

$$F(x; \alpha, \beta) = 1 - (\beta/x)^{\alpha}$$

The Quantile is:

$$F^{-1}(p; \alpha, \beta) = \frac{\beta}{(1-p)^{1/\alpha}}$$

Usage

```
pareto_distribution(scale = 1, shape = 1)

pareto_pdf(x, scale = 1, shape = 1)

pareto_lpdf(x, scale = 1, shape = 1)

pareto_cdf(x, scale = 1, shape = 1)

pareto_lcdf(x, scale = 1, shape = 1)

pareto_quantile(p, scale = 1, shape = 1)
```

Arguments

scale	scale parameter (default is 1)
shape	shape parameter (default is 1)
x	quantile
p	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Pareto distribution with scale = 10, shape = 5
dist <- pareto_distribution(10, 5)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
pareto_pdf(1)
pareto_lpdf(1)
pareto_cdf(1)
pareto_lcdf(1)
pareto_quantile(0.5)
```

pchip *PCHIP Interpolator*

Description

The PCHIP (Piecewise Cubic Hermite Interpolating Polynomial) interpolant takes non-equispaced data and interpolates between them via cubic Hermite polynomials whose slopes are chosen to preserve monotonicity.

Details:

The interpolant is C1 and evaluation has O(log N) complexity. See Fritsch and Carlson for details.

Usage

```
pchip(x, y, left_endpoint_derivative = NULL, right_endpoint_derivative = NULL)
```

Arguments

`x`	Numeric vector of abscissas (x-coordinates).
`y`	Numeric vector of ordinates (y-coordinates).
`left_endpoint_derivative`	Optional numeric value of the derivative at the left endpoint.
`right_endpoint_derivative`	Optional numeric value of the derivative at the right endpoint.

Value

An object of class `pchip` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `push_back(x, y)`: Add a new control point

See Also

Boost Documentation

Examples

```
x <- c(0, 1, 2, 3)
y <- c(0, 1, 0, 1)
interpolator <- pchip(x, y)
xi <- 0.5
interpolator$interpolate(xi)
interpolator$prime(xi)
interpolator$push_back(4, 1)
```

`poisson_distribution` *Poisson Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Poisson distribution.

The Poisson distribution expresses the probability of a number of events (or failures, arrivals, occurrences ...) occurring in a fixed period of time, provided these events occur with a known mean rate λ (events/time), and are independent of the time since the last event.

It has the Probability Mass Function:

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

for k events, with an expected number of events λ.

Accuracy and Implementation Notes: The Poisson distribution is implemented in terms of the incomplete gamma functions (`gamma_p` and `gamma_q`) and as such should have low error rates. The quantile function will by default return an integer result that has been rounded outwards to ensure that if an X% quantile is requested, then at least the requested coverage will be present in the central region, and no more than the requested coverage will be present in the tails.

Usage

```
poisson_distribution(lambda = 1)

poisson_pdf(x, lambda = 1)

poisson_lpdf(x, lambda = 1)

poisson_cdf(x, lambda = 1)

poisson_lcdf(x, lambda = 1)

poisson_quantile(p, lambda = 1)
```

Arguments

`lambda`	rate parameter (default is 1)
`x`	quantile
`p`	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Poisson distribution with lambda = 1
dist <- poisson_distribution(1)
# Apply generic functions
cdf(dist, 5)
logcdf(dist, 5)
pdf(dist, 5)
logpdf(dist, 5)
hazard(dist, 5)
chf(dist, 5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
poisson_pdf(0, 1)
poisson_lpdf(0, 1)
poisson_cdf(0, 1)
poisson_lcdf(0, 1)
poisson_quantile(0.5, 1)
```

---

polynomial_root_finding

Polynomial Root-Finding

Description

Functions for finding roots of polynomials of degree 2, 3, and 4 (quadratic, cubic, quartic).

Quadratic Roots: Solves $ax^2 + bx + c = 0$. Returns real roots. If roots are complex, behavior depends on implementation (typically NaN for this real-valued interface).

Cubic Roots: Solves $ax^3 + bx^2 + cx + d = 0$. Returns real roots.

Quartic Roots: Solves $ax^4 + bx^3 + cx^2 + dx + e = 0$. Returns real roots.

Usage

```
quadratic_roots(a, b, c)

cubic_roots(a, b, c, d)

cubic_root_residual(a, b, c, d, root)

cubic_root_condition_number(a, b, c, d, root)

quartic_roots(a, b, c, d, e)
```

Arguments

a	Coefficient of the highest degree term.
b	Coefficient of the second highest degree term.
c	Coefficient of the third highest degree term (or constant for quadratic).
d	Coefficient of the fourth highest degree term (or constant for cubic).
root	The root to evaluate the residual or condition number at.
e	Constant term for quartic.

Details

These functions use analytic formulas where possible and numerically stable implementations to avoid catastrophic cancellation.

Value

A numeric vector containing the real roots of the polynomial.

Examples

```
# Quadratic: x^2 - 3x + 2 = 0 -> Roots: 1, 2
quadratic_roots(1, -3, 2)

# Cubic: x^3 - 6x^2 + 11x - 6 = 0 -> Roots: 1, 2, 3
cubic_roots(1, -6, 11, -6)

# Quartic: x^4 - 10x^3 + 35x^2 - 50x + 24 = 0 -> Roots: 1, 2, 3, 4
quartic_roots(1, -10, 35, -50, 24)

# Residual and Condition Number
cubic_root_residual(1, -6, 11, -6, 1)
cubic_root_condition_number(1, -6, 11, -6, 1)
```

quintic_hermite *Quintic Hermite Interpolator*

Description

The quintic Hermite interpolator takes a list of possibly non-uniformly spaced abscissas, ordinates, and their velocities and accelerations.

Applications:

It constructs a quintic interpolating polynomial between segments. This is useful for taking solution skeletons from ODE steppers and turning them into a continuous function. The interpolant is C2 and its evaluation has O(log N) complexity.

Usage

```
quintic_hermite(x, y, dydx, d2ydx2)
```

Arguments

x	Numeric vector of abscissas (x-coordinates).
y	Numeric vector of ordinates (y-coordinates).
dydx	Numeric vector of first derivatives (slopes) at each point.
d2ydx2	Numeric vector of second derivatives at each point.

Value

An object of class `quintic_hermite` with methods:

- `interpolate(xi)`: Evaluate the interpolator at point `xi`.
- `prime(xi)`: Evaluate the derivative of the interpolator at point `xi`.
- `double_prime(xi)`: Evaluate the second derivative of the interpolator at point `xi`.
- `push_back(x, y, dydx, d2ydx2)`: Add a new control point to the interpolator.
- `domain()`: Get the domain of the interpolator.

See Also

Boost Documentation

Examples

```
x <- c(0, 1, 2)
y <- c(0, 1, 0)
dydx <- c(1, 0, -1)
d2ydx2 <- c(0, -1, 0)
interpolator <- quintic_hermite(x, y, dydx, d2ydx2)
xi <- 0.5
interpolator$interpolate(xi)
```

```
interpolator$prime(xi)
interpolator$double_prime(xi)
interpolator$push_back(3, 0, 1, 0)
interpolator$domain()
```

`rayleigh_distribution` *Rayleigh Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Rayleigh distribution.

The Rayleigh distribution is a continuous distribution. It is often used where two orthogonal components have an absolute value, for example, wind velocity and direction may be combined to yield a wind speed, or real and imaginary components may have absolute values that are Rayleigh distributed.

It has the probability density function (PDF):

$$f(x;\sigma) = \frac{x}{\sigma^2} e^{-x^2/(2\sigma^2)}$$

for $\sigma > 0$ and $x > 0$.

Accuracy and Implementation Notes: The Rayleigh distribution is implemented in terms of the standard library `sqrt` and `exp` and as such should have very low error rates.

Usage

```
rayleigh_distribution(sigma = 1)

rayleigh_pdf(x, sigma = 1)

rayleigh_lpdf(x, sigma = 1)

rayleigh_cdf(x, sigma = 1)

rayleigh_lcdf(x, sigma = 1)

rayleigh_quantile(p, sigma = 1)
```

Arguments

`sigma`	scale parameter (default is 1)
`x`	quantile
`p`	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Rayleigh distribution with sigma = 1
dist <- rayleigh_distribution(1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
rayleigh_pdf(1)
rayleigh_lpdf(1)
rayleigh_cdf(1)
rayleigh_lcdf(1)
rayleigh_quantile(0.5)
```

---

rootfinding_and_minimisation

Root-Finding and Minimisation

Description

Functions for finding roots of equations and minimizing functions using various numerical methods.

Root Finding Without Derivatives: These methods require a bracket (an interval $[a, b]$ where the function has opposite signs) or a guess.

- **Bisection (**`bisect`**):** A robust method that repeatedly subdivides the interval. Guaranteed to converge but slowly (linear convergence).
- **TOMS 748 (**`toms748_solve`**):** An asymptotically efficient algorithm (Alefeld, Potra, and Shi) that combines interpolation and bisection. It has higher-order convergence and is often optimal for smooth functions.
- **Bracket and Solve (**`bracket_and_solve_root`**):** A convenience wrapper that attempts to find a bracket around a guess and then solves using TOMS 748.

Root Finding With Derivatives: These methods require the user to provide derivatives of the function.

- **Newton-Raphson (**`newton_raphson_iterate`**):** Second-order convergence. Requires $f(x)$ and $f'(x)$.
- **Halley's Method (**`halley_iterate`**):** Third-order convergence. Requires $f(x)$, $f'(x)$, and $f''(x)$.
- **Schroder's Method (**`schroder_iterate`**):** Third-order convergence. Similar to Halley's method but more robust ensuring quadratic convergence for multiple roots.

Minimization:

- **Brent's Method (**`brent_find_minima`**):** Finds the minimum of a function in a given interval. It is a hybrid method using a combination of the golden section search and quadratic interpolation.

Usage

```
bisect(
  f,
  lower,
  upper,
  digits = .Machine$double.digits,
  max_iter = .Machine$integer.max
)

bracket_and_solve_root(
  f,
  guess,
  factor,
  rising,
  digits = .Machine$double.digits,
  max_iter = .Machine$integer.max
)

toms748_solve(
  f,
  lower,
  upper,
  digits = .Machine$double.digits,
  max_iter = .Machine$integer.max
```

```
)

newton_raphson_iterate(
  f,
  guess,
  lower,
  upper,
  digits = .Machine$double.digits,
  max_iter = .Machine$integer.max
)

halley_iterate(
  f,
  guess,
  lower,
  upper,
  digits = .Machine$double.digits,
  max_iter = .Machine$integer.max
)

schroder_iterate(
  f,
  guess,
  lower,
  upper,
  digits = .Machine$double.digits,
  max_iter = .Machine$integer.max
)

brent_find_minima(
  f,
  lower,
  upper,
  digits = .Machine$double.digits,
  max_iter = .Machine$integer.max
)
```

Arguments

`f`	A function to find the root of or to minimise. • For **no-derivative** methods: A function returning a single numeric value. • For **Newton-Raphson**: A function returning a vector `c(f(x), f'(x))`. • For **Halley/Schroder**: A function returning a vector `c(f(x), f'(x), f''(x))`. • For **Minimization**: A function returning a single numeric value.
`lower`	The lower bound of the interval to search.
`upper`	The upper bound of the interval to search.
`digits`	The number of significant digits to which the root or minimum should be found. Default is double precision.

max_iter	The maximum number of iterations to perform.
guess	A numeric value that is a guess for the root.
factor	Size of steps to take when searching for the root (for bracket_and_solve_root).
rising	If TRUE, the function is assumed to be rising (for bracket_and_solve_root).

Value

A list containing the root or minimum value, the value of the function at that point, and the number of iterations performed.

See Also

polynomial_root_finding

Examples

```
# --- Root Finding Without Derivatives ---
# Bisection for x^2 - 2 = 0
f_bi <- function(x) x^2 - 2
bisect(f_bi, lower = 0, upper = 2)

# TOMS 748 for x^2 - 2 = 0
toms748_solve(f_bi, lower = 0, upper = 2)

# Bracket and Solve
bracket_and_solve_root(f_bi, guess = 1, factor = 2, rising = TRUE)

# --- Root Finding With Derivatives ---
# Newton-Raphson: Need f(x) and f'(x)
# x^2 - 2 = 0  => f(x) = x^2 - 2, f'(x) = 2x
f_newton <- function(x) c(x^2 - 2, 2 * x)
newton_raphson_iterate(f_newton, guess = 1, lower = 0, upper = 2)

# Halley/Schroder: Need f(x), f'(x), f''(x)
# x^2 - 2 = 0  => f''(x) = 2
f_halley <- function(x) c(x^2 - 2, 2 * x, 2)
halley_iterate(f_halley, guess = 1, lower = 0, upper = 2)
schroder_iterate(f_halley, guess = 1, lower = 0, upper = 2)

# --- Minimization ---
# Find minimum of (x-2)^2 + 1
f_min <- function(x) (x - 2)^2 + 1
brent_find_minima(f_min, lower = 0, upper = 4)
```

`runs_tests` *Runs Tests*

Description

The runs test is a statistical test that checks a randomness hypothesis for a two-valued data sequence. It can be used to test the hypothesis that the elements of the sequence are mutually independent.

Runs Above and Below Median: Determines if a sequence is random by observing the number of consecutive values which exceed (or are below) the median of the sequence. Values equal to the median are ignored. The expected number of runs and variance are calculated according to NIST standards to derive a test statistic and p-value. This function calculates the median internally.

Runs Above and Below Threshold: similar to the median test, but uses a user-specified `threshold` instead of the calculated median. This is more efficient if the median is already known or if a different threshold is required.

Usage

```
runs_above_and_below_threshold(v, threshold)

runs_above_and_below_median(v)
```

Arguments

`v`	A numeric vector containing the sequence to test.
`threshold`	A single numeric value to serve as the threshold for the test (for `runs_above_and_below_threshold`).

Details

The test statistic is approximated as a standard normal distribution to extract the p-value. If the p-value is small (e.g., < 0.05), the null hypothesis of randomness is rejected.

Value

A two-element numeric vector:

- The first element is the **t-statistic**.
- The second element is the **p-value** (two-sided).

References

NIST/SEMATECH e-Handbook of Statistical Methods, "Runs Test for Detecting Non-randomness", https://www.itl.nist.gov/div898/handbook/eda/section3/eda35d.htm

See Also

Boost Documentation

Examples

```
# Runs Above and Below Threshold with a known threshold
v <- c(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
runs_above_and_below_threshold(v, threshold = 3)

# Runs Above and Below Median (calculates median = 3 internally)
runs_above_and_below_median(v)

# Example of a non-random sequence (fewer runs than expected)
v_non_random <- c(rep(1, 5), rep(10, 5))
runs_above_and_below_median(v_non_random)
```

`saspoint5_distribution`

SαS Point5 Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the SαS Point5 distribution.

The SαS Point5 distribution is a special case of a stable distribution with shape parameter $\alpha = 1/2$, $\beta = 0$.

It has the probability density function (PDF):

$$f(x; \mu, \gamma) = \frac{1}{\sqrt{2\pi}} x^{-3/2} e^{-\frac{1}{2x}}$$

(Note: The boost documentation reference shows a standard form, generalised by location μ and scale γ).

This distribution has heavier tails than the Cauchy distribution. Note that the SαS Point5 distribution does not have a defined mean or standard deviation.

Accuracy and Implementation Notes: The error is within 4 epsilon.

Usage

```
saspoint5_distribution(location = 0, scale = 1)

saspoint5_pdf(x, location = 0, scale = 1)

saspoint5_lpdf(x, location = 0, scale = 1)

saspoint5_cdf(x, location = 0, scale = 1)

saspoint5_lcdf(x, location = 0, scale = 1)

saspoint5_quantile(p, location = 0, scale = 1)
```

Arguments

location	location parameter (default is 0)
scale	scale parameter (default is 1)
x	quantile
p	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# SaS Point5 distribution with location 0 and scale 1
dist <- saspoint5_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
support(dist)

# Convenience functions
saspoint5_pdf(3)
saspoint5_lpdf(3)
saspoint5_cdf(3)
saspoint5_lcdf(3)
saspoint5_quantile(0.5)
```

---

signal_statistics *Signal Statistics*

Description

Functions for computing statistics commonly used in signal analysis, such as sparsity measures and Signal-to-Noise Ratio (SNR) estimators.

Absolute Gini Coefficient: The Gini coefficient is a measure of the sparsity of a signal (expansion in a basis).

- `absolute_gini_coefficient`: Computes the population Gini coefficient.
- `sample_absolute_gini_coefficient`: Computes the sample Gini coefficient. A Gini coefficient of 0 implies every element has equal magnitude (least sparse). A Gini coefficient of 1 implies only one element is non-zero (most sparse).

Hoyer Sparsity: A measure of sparsity based on the ratio of the L1 and L2 norms:

$$H(v) = \frac{\sqrt{N} - \frac{\|v\|_1}{\|v\|_2}}{\sqrt{N} - 1}$$

Returns 1 for maximum sparsity (one non-zero element) and 0 for minimum sparsity (all elements equal).

Oracle SNR: Computes the Signal-to-Noise Ratio (SNR) when the true signal is known (Oracle).

- `oracle_snr`: $\frac{\|s\|^2}{\|s-x\|^2}$ where s is the signal and x is the noisy signal.
- `oracle_snr_db`: Returns the SNR in decibels (dB): $10 \log_{10}(\text{SNR})$.

M2M4 SNR Estimator: A "blind" estimator (requires no clean signal reference) using the M2M4 property (uses 2nd and 4th moments). Useful for "in-service" estimation.

- `m2m4_snr_estimator`: Returns the estimated SNR ratio.
- `m2m4_snr_estimator_db`: Returns the estimated SNR in dB. Works best for SNR between -3 dB and 15 dB. Requires assumptions about signal and noise kurtosis (default signal=1, noise=3 for Gaussian).

Usage

```
absolute_gini_coefficient(x)

sample_absolute_gini_coefficient(x)

hoyer_sparsity(x)

oracle_snr(signal, noisy_signal)

oracle_snr_db(signal, noisy_signal)

m2m4_snr_estimator(noisy_signal, signal_kurtosis = 1, noise_kurtosis = 3)

m2m4_snr_estimator_db(noisy_signal, signal_kurtosis = 1, noise_kurtosis = 3)
```

Arguments

`x`	A numeric vector representing the signal or coefficients.
`signal`	A numeric vector representing the true (clean) signal.
`noisy_signal`	A numeric vector representing the signal with noise.
`signal_kurtosis`	Kurtosis of the signal (for M2M4). Default is 1 (e.g., constant amplitude).
`noise_kurtosis`	Kurtosis of the noise (for M2M4). Default is 3 (Gaussian noise).

Value

A numeric value representing the computed statistic.

References

Hurley, N., & Rickard, S. (2009). Comparing measures of sparsity. IEEE Transactions on Information Theory, 55(10), 4723-4741. Pauluzzi, D. R., & Beaulieu, N. C. (2000). A comparison of SNR estimation techniques for the AWGN channel. IEEE Transactions on Communications, 48(10), 1681-1691.

See Also

Boost Documentation

Examples

```
# --- Sparsity Measures ---

# Gini Coefficient
vec <- c(1, 0, 0, 0)
# High sparsity -> High Gini coefficient
sample_absolute_gini_coefficient(vec)

vec_uniform <- c(1, 1, 1, 1)
# Low sparsity -> Low Gini coefficient
absolute_gini_coefficient(vec_uniform)

# Hoyer Sparsity
hoyer_sparsity(vec)      # Returns 1
hoyer_sparsity(vec_uniform) # Returns 0

# --- SNR Estimation ---

s <- sin(seq(0, 10, length.out = 100))
n <- rnorm(100, sd = 0.5)
x <- s + n

# Oracle SNR (Known signal)
oracle_snr_db(s, x)

# M2M4 Blind SNR Estimation
# Assuming signal kurtosis = 1.5 (sinusoid) and Gaussian noise (kurtosis = 3)
m2m4_snr_estimator_db(x, signal_kurtosis = 1.5, noise_kurtosis = 3)
```

---

sinus_cardinal_hyperbolic_functions

Sinus Cardinal and Hyperbolic Sinus Cardinal Functions

Description

Functions to compute the sinus cardinal (sinc) and hyperbolic sinus cardinal (sinhc) functions.

These functions appear in signal processing, Fourier analysis, and various mathematical applications. The implementations avoid numerical instability near x = 0.

Sinus Cardinal Function:

The sinc function is defined as:

$$sinc(\pi x) = \frac{\sin(\pi x)}{\pi x}$$

- `sinc_pi(x)`: Computes sinc(pix) = sin(pix)/(pix)
- Special value: sinc_pi(0) = 1 (by L'Hopital's rule or Taylor series)
- The function oscillates with decreasing amplitude as |x| increases
- Used extensively in signal processing (ideal low-pass filter impulse response)
- Appears in the Whittaker-Shannon interpolation formula

Hyperbolic Sinus Cardinal Function:

The hyperbolic sinc function is defined as:

$$sinhc(\pi x) = \frac{\sinh(\pi x)}{\pi x}$$

- `sinhc_pi(x)`: Computes sinhc(pix) = sinh(pix)/(pix)
- Special value: sinhc_pi(0) = 1 (by L'Hopital's rule or Taylor series)
- The function grows exponentially for large |x|
- Analogous to sinc but using hyperbolic sine instead of circular sine

Numerical Stability:

Both functions use Taylor series expansions near x = 0 to avoid division by zero and loss of precision. For x away from 0, direct evaluation is used.

Usage

```
sinc_pi(x)

sinhc_pi(x)
```

Arguments

`x`	Input value

Value

A single numeric value with the computed sinus cardinal or hyperbolic sinus cardinal function.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Sinus cardinal function at x = 0.5: sinc(pi/2)
sinc_pi(0.5)
# Sinus cardinal at zero (returns exactly 1)
sinc_pi(0)
# Hyperbolic sinus cardinal function
sinhc_pi(0.5)
# Hyperbolic sinus cardinal at zero (returns exactly 1)
sinhc_pi(0)
```

---

skew_normal_distribution

Skew Normal Distribution Functions

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Skew Normal distribution.

The skew normal distribution is a variant of the most well known Gaussian statistical distribution. If the standard (mean = 0, scale = 1) normal distribution probability density function is $\phi(x)$ and the cumulative distribution function is $\Phi(x)$, then the PDF of the skew normal distribution with shape parameter α is:

$$f(x;\alpha) = 2\phi(x)\Phi(\alpha x)$$

Given location ξ, scale ω, and shape α, it can be transformed to:

$$f(x) = \frac{2}{\omega}\phi\left(\frac{x-\xi}{\omega}\right)\Phi\left(\alpha\frac{x-\xi}{\omega}\right)$$

Accuracy and Implementation Notes: The skew_normal distribution with shape = zero is equivalent to the normal distribution and uses the error function for excellent accuracy. The CDF requires Owen's T function, which is evaluated using algoritms by Patefield and Tandy. The median and mode are calculated by iterative root finding and may be less accurate than other estimates.

Usage

```
skew_normal_distribution(location = 0, scale = 1, shape = 0)

skew_normal_pdf(x, location = 0, scale = 1, shape = 0)

skew_normal_lpdf(x, location = 0, scale = 1, shape = 0)

skew_normal_cdf(x, location = 0, scale = 1, shape = 0)

skew_normal_lcdf(x, location = 0, scale = 1, shape = 0)

skew_normal_quantile(p, location = 0, scale = 1, shape = 0)
```

Arguments

location	location parameter (default is 0)
scale	scale parameter (default is 1)
shape	shape parameter (default is 0)
x	quantile
p	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Skew Normal distribution with location = 0, scale = 1, shape = 0
dist <- skew_normal_distribution(0, 1, 0)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
skew_normal_pdf(0)
skew_normal_lpdf(0)
skew_normal_cdf(0)
skew_normal_lcdf(0)
skew_normal_quantile(0.5)
```

spherical_harmonics *Spherical Harmonics*

Description

Functions to compute spherical harmonics and related functions.

Usage

```
spherical_harmonic(n, m, theta, phi)

spherical_harmonic_r(n, m, theta, phi)

spherical_harmonic_i(n, m, theta, phi)
```

Arguments

n	Degree of the spherical harmonic
m	Order of the spherical harmonic
theta	Polar angle (colatitude)
phi	Azimuthal angle (longitude)

Value

A single complex value with the computed spherical harmonic function, or its real and imaginary parts.

See Also

Boost Documentation

Examples

```
# Spherical harmonic function Y_2^1(0.5, 0.5)
spherical_harmonic(2, 1, 0.5, 0.5)
# Real part of the spherical harmonic function Y_2^1(0.5, 0.5)
spherical_harmonic_r(2, 1, 0.5, 0.5)
# Imaginary part of the spherical harmonic function Y_2^1(0.5, 0.5)
spherical_harmonic_i(2, 1, 0.5, 0.5)
```

`students_t_distribution` *Student's T Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Student's t distribution.

Student's t-distribution is defined as the distribution of the random variable t which is (very loosely) the "best" that we can do while not knowing the true standard deviation of the sample.

Given N independent measurements, let

$$t = \frac{\mu - M}{s/\sqrt{N}}$$

where M is the population mean, μ is the sample mean, and s is the sample variance.

It has the PDF:

$$f(x;\nu) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\nu\pi}\Gamma(\nu/2)}\left(1+\frac{x^2}{\nu}\right)^{-(\nu+1)/2}$$

where ν is the degrees of freedom.

Accuracy and Implementation Notes: The Student's t distribution is implemented in terms of the incomplete beta function and its inverses.

Usage

```
students_t_distribution(df = 1)

students_t_pdf(x, df = 1)

students_t_lpdf(x, df = 1)

students_t_cdf(x, df = 1)

students_t_lcdf(x, df = 1)

students_t_quantile(p, df = 1)

students_t_find_degrees_of_freedom(
  difference_from_mean,
  alpha,
  beta,
  sd,
  hint = 100
)
```

Arguments

df	degrees of freedom (default is 1)
x	quantile
p	probability (0 <= p <= 1)
difference_from_mean	The difference from the assumed nominal mean that is to be detected.
alpha	The acceptable probability of a Type I error (false positive).
beta	The acceptable probability of a Type II error (false negative).
sd	The assumed standard deviation.
hint	An initial guess for the degrees of freedom to start the search from (current sample size is a good start).

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Student's t distribution with 5 degrees of freedom
dist <- students_t_distribution(5)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
students_t_pdf(0, 5)
students_t_lpdf(0, 5)
students_t_cdf(0, 5)
students_t_lcdf(0, 5)
```

```
students_t_quantile(0.5, 5)

# Find degrees of freedom needed to detect a difference from mean of 2.0
# with alpha = 0.05 and beta = 0.2 when the standard deviation is 3.0
students_t_find_degrees_of_freedom(2.0, 0.05, 0.2, 3.0)
```

`t_tests` *Student's T-Tests*

Description

Functions for performing various Student's t-tests to compare means of populations.

One-Sample T-Test: Tests if the population mean differs from a specified `assumed_mean`.

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

Available in two forms:

- `one_sample_t_test`: Takes a vector of data.
- `one_sample_t_test_params`: Takes summary statistics (mean, variance, sample size) directly.

Two-Sample T-Test (`two_sample_t_test`): Tests if the means of two independent samples differ. Automatically handles unequal variances (Welch's t-test) or equal variances based on the data.

Paired Samples T-Test (`paired_samples_t_test`): Tests if the means of two dependent (paired) samples differ. Equivalent to a one-sample t-test on the differences between pairs.

Usage

```
one_sample_t_test_params(
  sample_mean,
  sample_variance,
  num_samples,
  assumed_mean
)

one_sample_t_test(u, assumed_mean)

two_sample_t_test(u, v)

paired_samples_t_test(u, v)
```

Arguments

sample_mean	Sample mean (for one_sample_t_test_params).
sample_variance	Sample variance (for one_sample_t_test_params).
num_samples	Number of samples (for one_sample_t_test_params).
assumed_mean	The hypothesised population mean to compare against.
u	A numeric vector of data values for the first sample.
v	A numeric vector of data values for the second sample.

Value

A two-element numeric vector containing:

- The **t-statistic**.
- The **p-value** (two-sided).

See Also

Boost Documentation

Examples

```
# --- One Sample T-Test ---
# Using raw data:
data <- c(5, 6, 7, 5, 6)
one_sample_t_test(data, assumed_mean = 4)

# Using summary statistics:
# Mean = 5.8, Variance = 0.7, N = 5
one_sample_t_test_params(sample_mean = 5.8, sample_variance = 0.7,
                         num_samples = 5, assumed_mean = 4)

# --- Two Sample T-Test ---
sample1 <- c(5, 6, 7, 5, 6)
sample2 <- c(4, 5, 6, 4, 5)
two_sample_t_test(sample1, sample2)

# --- Paired Samples T-Test ---
# Pre-test vs Post-test
pre <- c(5, 6, 7, 5, 6)
post <- c(6, 7, 8, 6, 7)
paired_samples_t_test(pre, post)
```

`triangular_distribution` *Triangular Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Triangular distribution.

The triangular distribution is a continuous probability distribution with a lower limit a, mode c, and upper limit b. It is often used where the distribution is only vaguely known, but upper and lower limits are known, and a "best guess" (mode) is added.

It has the probability density function (PDF):

$$f(x) = \frac{2(x-a)}{(b-a)(c-a)} \quad \text{for } a \leq x \leq c$$

$$f(x) = \frac{2(b-x)}{(b-a)(b-c)} \quad \text{for } c < x \leq b$$

Accuracy and Implementation Notes: The triangular distribution is implemented with simple arithmetic operators and so should have errors within an epsilon or two.

Usage

```
triangular_distribution(lower = -1, mode = 0, upper = 1)

triangular_pdf(x, lower = -1, mode = 0, upper = 1)

triangular_lpdf(x, lower = -1, mode = 0, upper = 1)

triangular_cdf(x, lower = -1, mode = 0, upper = 1)

triangular_lcdf(x, lower = -1, mode = 0, upper = 1)

triangular_quantile(p, lower = -1, mode = 0, upper = 1)
```

Arguments

`lower`	lower limit of the distribution (default is -1)
`mode`	mode of the distribution (default is 0)
`upper`	upper limit of the distribution (default is 1)
`x`	quantile
`p`	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Triangular distribution with lower = -1, mode = 0, upper = 1
dist <- triangular_distribution(-1, 0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
triangular_pdf(1)
triangular_lpdf(1)
triangular_cdf(1)
triangular_lcdf(1)
triangular_quantile(0.5)
```

---

uniform_distribution *Uniform Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Uniform distribution.

The uniform distribution, also known as a rectangular distribution, is a probability distribution that has constant probability.

The continuous uniform distribution has the probability density function (PDF):

$$f(x) = \frac{1}{\text{upper} - \text{lower}} \quad \text{for lower} \leq x \leq \text{upper}$$

$$f(x) = 0 \quad \text{for } x < \text{lower or } x > \text{upper}$$

Accuracy and Implementation Notes: The uniform distribution is implemented with simple arithmetic operators and so should have errors within an epsilon or two.

Usage

```
uniform_distribution(lower = 0, upper = 1)

uniform_pdf(x, lower = 0, upper = 1)

uniform_lpdf(x, lower = 0, upper = 1)

uniform_cdf(x, lower = 0, upper = 1)

uniform_lcdf(x, lower = 0, upper = 1)

uniform_quantile(p, lower = 0, upper = 1)
```

Arguments

lower	lower bound of the distribution (default is 0)
upper	upper bound of the distribution (default is 1)
x	quantile
p	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Uniform distribution with lower = 0, upper = 1
dist <- uniform_distribution(0, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
```

```
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
uniform_pdf(0.5)
uniform_lpdf(0.5)
uniform_cdf(0.5)
uniform_lcdf(0.5)
uniform_quantile(0.5)
```

`univariate_statistics` *Univariate Statistics*

Description

Functions to compute robust univariate statistics from a dataset.

Central Tendency:

- `mean_boost`: Computes the arithmetic mean using Higham's numerically stable algorithm.
- `median_boost`: Computes the median (robust to outliers).
- `mode`: Computes the mode(s) of the dataset.

Dispersion (Spread):

- `variance`: Computes the population variance using Higham's algorithm.
- `sample_variance`: Computes the sample variance (unbiased estimator).
- `mean_and_sample_variance`: Efficiently computes both mean and sample variance in one pass.
- `median_absolute_deviation`: Computes the Median Absolute Deviation (MAD), a robust measure of variability.
- `interquartile_range`: Computes the Interquartile Range (IQR = Q3 - Q1), robust to outliers.

Shape:

- `skewness`: Measures the asymmetry of the distribution (Pebay's algorithm).
- `kurtosis`: Measures the "tailedness" of the distribution (Pebay's algorithm).

- `excess_kurtosis`: Kurtosis minus 3 (Normal distribution has 0 excess kurtosis).
- `first_four_moments`: Computes Mean, Variance, Skewness, and Kurtosis in a single pass.

Inequality:

- `gini_coefficient`: Computes the Gini coefficient (population). range $[0, 1 - 1/n]$.
- `sample_gini_coefficient`: Computes the sample Gini coefficient. range $[0, 1]$.

Usage

```
mean_boost(x)

## Default S3 method:
variance(x, ...)

sample_variance(x)

mean_and_sample_variance(x)

## Default S3 method:
skewness(x, ...)

## Default S3 method:
kurtosis(x, ...)

excess_kurtosis(x)

first_four_moments(x)

median_boost(x)

median_absolute_deviation(x)

interquartile_range(x)

gini_coefficient(x)

sample_gini_coefficient(x)

## Default S3 method:
mode(x, ...)
```

Arguments

`x`	A numeric vector containing the dataset.
`...`	Additional arguments (for S3 compatibility, e.g., with defaults).

Details

These functions are designed to be numerically stable and efficient. Most implementations follow algorithms described by Higham (Accuracy and Stability of Numerical Algorithms) or Pebay (Sandia Labs) for one-pass parallel computation.

Value

A numeric value (or vector for moments/mode) with the computed statistic.

References

Higham, N. J. (2002). Accuracy and stability of numerical algorithms. SIAM. Pebay, P. P. (2008). Formulas for Robust, One-Pass Parallel Computation of Covariances and Arbitrary-Order Statistical Moments. Sandia Report.

See Also

Boost Documentation

Examples

```
data <- c(1, 2, 3, 4, 100) # Dataset with an outlier

# --- Central Tendency ---
mean_boost(data)
median_boost(data) # Less affected by 100
mode(c(1, 2, 2, 3))

# --- Dispersion ---
variance(data)
sample_variance(data)
median_absolute_deviation(data) # Robust
interquartile_range(data)       # Robust

# --- Shape ---
skewness(data)
excess_kurtosis(data)
first_four_moments(data)

# --- Inequality ---
gini_coefficient(c(1, 0, 0, 0)) # High inequality
# Gini Coefficient
gini_coefficient(c(1, 2, 3, 4, 5))
# Sample Gini Coefficient
sample_gini_coefficient(c(1, 2, 3, 4, 5))
# Mode
mode(c(1, 2, 2, 3, 4))
```

vector_functionals *Vector Norms and Distances*

Description

Functions to compute various vector norms, distances, and functional properties. These functionals form the basis of many numerical analysis and signal processing algorithms.

Norms (Magnitude):

- `l1_norm`: Computes the L1 norm (Manhattan norm), sum of absolute values.
- `l2_norm`: Computes the L2 norm (Euclidean norm), square root of sum of squares.
- `sup_norm`: Computes the Supremum (L-infinity) norm, the maximum absolute value.
- `lp_norm`: Computes the Lp norm for an arbitrary integer `p`.

Distances (Difference):

- `l1_distance`: Computes the L1 distance between two vectors.
- `l2_distance`: Computes the L2 (Euclidean) distance between two vectors.
- `sup_distance`: Computes the Supremum (L-infinity) distance/Chebyshev distance.
- `lp_distance`: Computes the Lp distance for an arbitrary integer `p`.

Sparsity & Structure:

- `l0_pseudo_norm`: Counts the number of non-zero elements (Hamming weight).
- `hamming_distance`: Counts the number of mismatching elements between two vectors.
- `total_variation`: Computes the total variation (sum of absolute differences between adjacent elements).

Usage

```
l0_pseudo_norm(x)

hamming_distance(x, y)

l1_norm(x)

l1_distance(x, y)

l2_norm(x)

l2_distance(x, y)

sup_norm(x)

sup_distance(x, y)
```

```
lp_norm(x, p)

lp_distance(x, y, p)

total_variation(x)
```

Arguments

x	A numeric vector.
y	A numeric vector of the same length as x (for distance functions).
p	A positive integer indicating the order of the norm or distance (for Lp functions).

Details

- **L0 Pseudo-Norm:** Not a true norm (doesn't satisfy homogeneity), but useful for sparsity (e.g., Compressed Sensing).
- **L1 Norm:** Often used in sparse signal recovery (LASSO).
- **Total Variation:** Useful in signal processing for denoising while preserving edges (Total Variation Denoising).

The implementations are designed to be efficient and work with various numeric types.

Value

A single numeric value with the computed norm or distance.

References

Higham, N. J. (2002). Accuracy and stability of numerical algorithms. SIAM. Mallat, S. (2008). A wavelet tour of signal processing: the sparse way. Academic press.

See Also

Boost Documentation

Examples

```
v1 <- c(1, -2, 3)
v2 <- c(4, -5, 6)

# --- Norms ---
l1_norm(v1)      # |1| + |-2| + |3| = 6
l2_norm(v1)      # sqrt(1^2 + (-2)^2 + 3^2) = sqrt(14)
sup_norm(v1)     # max(|1|, |-2|, |3|) = 3
lp_norm(v1, 3)   # Cube root of sum of cubes

# --- Distances ---
l1_distance(v1, v2)
l2_distance(v1, v2)
hamming_distance(c(1, 0, 1), c(0, 1, 1)) # 2 differences (pos 1 and 2)
```

```
# --- Structure ---
l0_pseudo_norm(c(0, 5, 0, 2)) # Returns 2 (two non-zeros)
total_variation(c(1, 5, 2))   # |5-1| + |2-5| = 4 + 3 = 7
```

`weibull_distribution` *Weibull Distribution Functions*

Description

Functions to compute the probability density function, cumulative distribution function, and quantile function for the Weibull distribution.

The Weibull distribution is a continuous distribution often used in the field of failure analysis; in particular it can mimic distributions where the failure rate varies over time.

It has the probability density function (PDF):

$$f(x; \alpha, \beta) = \frac{\alpha}{\beta} \left(\frac{x}{\beta} \right)^{\alpha - 1} e^{-(x/\beta)^{\alpha}}$$

for shape parameter $\alpha > 0$, scale parameter $\beta > 0$, and $x > 0$.

Accuracy and Implementation Notes: The Weibull distribution is implemented in terms of the standard library `log` and `exp` functions plus `expm1` and `log1p` and as such should have very low error rates.

Usage

```
weibull_distribution(shape, scale = 1)

weibull_pdf(x, shape, scale = 1)

weibull_lpdf(x, shape, scale = 1)

weibull_cdf(x, shape, scale = 1)

weibull_lcdf(x, shape, scale = 1)

weibull_quantile(p, shape, scale = 1)
```

Arguments

`shape`	shape parameter
`scale`	scale parameter (default is 1)
`x`	quantile
`p`	probability (0 <= p <= 1)

Value

A single numeric value with the computed probability density, log-probability density, cumulative distribution, log-cumulative distribution, or quantile depending on the function called.

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Weibull distribution with shape = 1, scale = 1
dist <- weibull_distribution(1, 1)
# Apply generic functions
cdf(dist, 0.5)
logcdf(dist, 0.5)
pdf(dist, 0.5)
logpdf(dist, 0.5)
hazard(dist, 0.5)
chf(dist, 0.5)
mean(dist)
median(dist)
mode(dist)
range(dist)
quantile(dist, 0.2)
standard_deviation(dist)
support(dist)
variance(dist)
skewness(dist)
kurtosis(dist)
kurtosis_excess(dist)

# Convenience functions
weibull_pdf(1, shape = 1, scale = 1)
weibull_lpdf(1, shape = 1, scale = 1)
weibull_cdf(1, shape = 1, scale = 1)
weibull_lcdf(1, shape = 1, scale = 1)
weibull_quantile(0.5, shape = 1, scale = 1)
```

---

z_tests *Z-Tests*

Description

Statistical hypothesis tests for population means using the Normal (Z) distribution.

Z-tests are typically used when:

1. The population variance is known.
2. The sample size is large (N > 30), allowing the sample variance to approximate the population variance (via Central Limit Theorem).

One-Sample Tests:

- `one_sample_z_test`: Performs a Z-test on a data vector `u` against an `assumed_mean`.
- `one_sample_z_test_params`: Performs a Z-test given summary statistics (mean, variance, N).

Two-Sample Tests:

- `two_sample_z_test`: Performs a Z-test comparing the means of two data vectors `u` and `v`.

Usage

```
one_sample_z_test_params(
  sample_mean,
  sample_variance,
  num_samples,
  assumed_mean
)

one_sample_z_test(u, assumed_mean)

two_sample_z_test(u, v)
```

Arguments

`sample_mean`	Numeric. The mean of the sample.
`sample_variance`	Numeric. The variance of the sample.
`num_samples`	Integer. The size of the sample.
`assumed_mean`	Numeric. The null hypothesis mean value to test against.
`u`	A numeric vector containing the first sample.
`v`	A numeric vector containing the second sample (for two-sample test).

Details

- **Statistic:** The Z-statistic is calculated as

$$Z = \frac{\bar{x} - \mu}{\sigma/\sqrt{n}}$$

.

- **Assumptions:** The underlying distribution is Normal, or the sample size is large enough for the CLT to apply.

Value

A numeric vector containing:

1. **Statistic:** The computed Z-statistic.
2. **P-Value:** The two-sided p-value associated with the Z-statistic.

See Also

Boost Documentation

Examples

```
# --- One-Sample Z-Test ---
data1 <- c(5, 6, 7, 5, 6, 8)
# Test if population mean is 4
one_sample_z_test(data1, assumed_mean = 4)

# Using Summary Statistics
# Mean = 2, Variance = 1, N = 30, Null Mean = 0
one_sample_z_test_params(sample_mean = 2,
                         sample_variance = 1,
                         num_samples = 30,
                         assumed_mean = 0)

# --- Two-Sample Z-Test ---
data2 <- c(4, 5, 6, 4, 5, 7)
# Test if data1 and data2 have different means
two_sample_z_test(data1, data2)
```

`zeta` *Riemann Zeta Function*

Description

Computes the Riemann zeta function zeta(z), one of the most important functions in analytic number theory.

Mathematical Definition:

The Riemann zeta function is defined by the series:

$$\zeta(z) = \sum_{n=1}^{\infty} \frac{1}{n^z}$$

for Re(z) > 1, and by analytic continuation elsewhere.

Special Values:

- zeta(2) = pi^2/6 (Basel problem)
- zeta(4) = pi^4/90
- zeta(0) = -1/2
- zeta(-1) = -1/12
- Closed forms exist for all even positive integers and all negative integers
- For odd positive integers > 1, values are computed numerically

Implementation:

The function uses different computational strategies depending on the argument:

- For 0 < z < 1: Rational approximation form
- For 1 < z < 4: Rational approximation around nearby integers
- For z > 4: Simple rational approximation series
- Reflection formula for negative arguments
- Pre-computed cached values for positive odd integers
- Specialised rational approximations for standard floating-point precisions

Applications:

The Riemann zeta function appears in number theory (distribution of primes), physics (quantum field theory, statistical mechanics), and probability theory. The famous Riemann Hypothesis concerns the non-trivial zeros of this function.

Usage

```
zeta(z)
```

Arguments

z	Real number input

Value

The value of the Riemann zeta function zeta(z).

See Also

Boost Documentation for more details on the mathematical background.

Examples

```
# Riemann Zeta Function
zeta(2)  # Should return pi^2 / 6 ~= 1.6449340668
zeta(3)  # Apery's constant ~= 1.2020569032
zeta(4)  # pi^4 / 90 ~= 1.0823232337
```

Index