Package: qfratio (via r-universe)

August 30, 2024

Type Package

Title Moments and Distributions of Ratios of Quadratic Forms Using Recursion

Version 1.1.1.9000

Date 2024-08-30

Description Evaluates moments of ratios (and products) of quadratic forms in normal variables, specifically using recursive algorithms developed by Bao and Kan (2013) <doi:10.1016/j.jmva.2013.03.002> and Hillier et al. (2014) <doi:10.1017/S0266466613000364>. Also provides distribution, quantile, and probability density functions of simple ratios of quadratic forms in normal variables with several algorithms. Originally developed as a supplement to Watanabe (2023) <doi:10.1007/s00285-023-01930-8> for evaluating average evolvability measures in evolutionary quantitative genetics, but can be used for a broader class of statistics. Generating functions for these moments are also closely related to the top-order zonal and invariant polynomials of matrix arguments.

License GPL (>= 3)

URL `https://github.com/watanabe-j/qfratio`

BugReports `https://github.com/watanabe-j/qfratio/issues`

Imports Rcpp, MASS, stats

LinkingTo Rcpp, RcppEigen

Suggests mvtnorm, CompQuadForm, graphics, testthat (>= 3.0.0), knitr, rmarkdown

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Config/testthat/edition 3

VignetteBuilder knitr, rmarkdown

Repository https://watanabe-j.r-universe.dev

RemoteUrl https://github.com/watanabe-j/qfratio

RemoteRef HEAD

RemoteSha b005bd05400f3d694ade357adc042dc62678f65d

Contents

`a1_pk` *Recursion for a_{p,k}*

Description

`a1_pk()` is an internal function to calculate $a_{p,k}$ ($a_{r,l}$ in Hillier et al. 2014; eq. 24), which is used in the calculation of the moment of such a ratio of quadratic forms in normal variables where the denominator matrix is identity.

Usage

```
a1_pk(L, mu = rep.int(0, n), m = 10L)
```

Arguments

`L`	Eigenvalues of the argument matrix; vector of λ_i
`mu`	Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$
`m`	Scalar to specify the desired order

Details

This function implements the super-short recursion described in Hillier et al. (2014 eqs. 31–32). Note that $w_{r,i}$ there should be understood as $w_{r,l,i}$ with the index l fixed for each $a_{r,l}$.

See Also

`qfrm_ApIq_int()`, in which this function is used (for noncentral cases only)

---

d1_i — *Coefficients in polynomial expansion of generating function—single matrix*

Description

These are internal functions to calculate the coefficients in polynomial expansion of generating functions for quadratic forms in multivariate normal variables.

`d1_i()` is for standard multivariate normal variables, $\mathbf{x} \sim N_n(\mathbf{0}_n, \mathbf{I}_n)$.

`dtil1_i_v()` is for noncentral multivariate normal variables, $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \mathbf{I}_n)$.

`dtil1_i_m()` is a wrapper for `dtil1_i_v()` and takes the argument matrix rather than its eigenvalues.

Usage

```
d1_i(L, m = 100L, thr_margin = 100)

dtil1_i_v(L, mu = rep.int(0, n), m = 100L, thr_margin = 100)

dtil1_i_m(A, mu = rep.int(0, n), m = 100L, thr_margin = 100)
```

Arguments

`L`	Vector of eigenvalues of the argument matrix
`m`	Integer-alike to specify the order of polynomials
`thr_margin`	Optional argument to adjust the threshold for scaling (see “Details”)
`mu`	Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$
`A`	Argument matrix. Assumed to be symmetric in these functions.

Details

`d1_i()` calculates $d_k(\mathbf{A})$, and `dtil1_i_v()` and `dtil1_i_m()` calculate $\tilde{d}_k(\mathbf{A})$ in Hillier et al. (2009, 2014) and Bao and Kan (2013). The former is related to the top-order zonal polynomial $C_{[k]}(\mathbf{A})$ in the following way: $d_k(\mathbf{A}) = \frac{1}{k!}\left(\frac{1}{2}\right)_k C_{[k]}(\mathbf{A})$, where $(x)_k = x(x+1)\dots(x+k-1)$.

These functions calculate the coefficients based on the super-short recursion algorithm described in Hillier et al. (2014: 3.2, eqs. 28–30).

Scaling:

The coefficients described herein (and in `d2_ij` and `d3_ijk`) can become very large for higher-order terms, so there is a practical risk of numerical overflow when applied to large matrices or matrices with many large eigenvalues (note that the latter typically arises from those with many small eigenvalues for the front-end `qfrm()` functions). To avoid numerical overflow, these functions automatically scale coefficients (and temporary objects used to calculate them) by a large number (`1e10` at present) when any value in the temporary objects exceeds a threshold, `.Machine$double.xmax / thr_margin / n`, where `n` is the number of variables. This default value empirically seems to work well in most conditions, but use a large `thr_margin` (e.g., `1e5`) if you encounter numerical overflow. (The C++ functions use an equivalent expression, `std::numeric_limits<Scalar>::max() / thr_margin / Scalar(n)`, with `Scalar` being `double` or `long double`.)

In these R functions, the scaling happens order-wise; i.e., it influences all the coefficients of the same order in multidimensional coefficients (in `d2_ij` and `d3_ijk`) and the coefficients of the subsequent orders.

These scaling factors are recorded in the attribute `"logscale"` of the return value, which is a vector/matrix/array whose size is identical to the return value, so that `value / exp(attr(value, "logscale"))` equals the original quantities to be obtained (if there were no overflow).

The `qfrm` and `qfmrm` functions handle return values of these functions by first multiplying them with hypergeometric coefficients (which are typically $\ll 1$) and then scaling the products back to the original scale using the recorded scaling factors. (To be precise, this typically happens within `hgs` functions.) The C++ functions handle the problem similarly (but by using separate objects rather than attributes).

However, this procedure does not always mitigate the problem in multiple series; when there are very large and very small coefficients in the same order, smaller ones can diminish/underflow to the numerical `0` after repeated scaling. (The `qfrm` and `qfmrm` functions try to detect and warn against this by examining whether any of the highest-order terms is `0`.) The present version of this package has implemented two methods to mitigate this problem, but only through C++ functions. One is to use the `long double` variable type, and the other is to use coefficient-wise scaling (see `qfrm` and `qfmrm`).

Value

Vector of length `m + 1`, corresponding to the 0th, 1st, ..., and mth order terms. Hence, the `[m + 1]`-th element should be extracted when the coefficient for the mth order term is required.

Has the attribute `"logscale"` as described in “Scaling” above.

References

Bao, Y. and Kan, R. (2013) On the moments of ratios of quadratic forms in normal random variables. *Journal of Multivariate Analysis*, **117**, 229–245. doi:10.1016/j.jmva.2013.03.002.

Hillier, G., Kan, R. and Wang, X. (2009) Computationally efficient recursions for top-order invariant polynomials with applications. *Econometric Theory*, **25**, 211–242. doi:10.1017/S0266466608090075.

Hillier, G., Kan, R. and Wang, X. (2014) Generating functions and short recursions, with applications to the moments of quadratic forms in noncentral normal vectors. *Econometric Theory*, **30**, 436–473. doi:10.1017/S0266466613000364.

See Also

`qfpm`, `qfrm`, and `qfmrm` are major front-end functions that utilize these functions

`dtil2_pq` for $\tilde{d}$ used for moments of a product of quadratic forms

`d2_ij` and `d3_ijk` for d, h, $\tilde{h}$, and $\hat{h}$ used for moments of ratios of quadratic forms

---

`d2_ij` *Coefficients in polynomial expansion of generating function—for ratios with two matrices*

Description

These are internal functions to calculate the coefficients in polynomial expansion of joint generating functions for two quadratic forms in potentially noncentral multivariate normal variables, $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \mathbf{I}_n)$. They are primarily used in calculations around moments of a ratio involving two or three quadratic forms.

Usage

```
d2_ij_m(
  A1,
  A2,
  m = 100L,
  p = m,
  q = m,
  thr_margin = 100,
  fill_all = !missing(p) || !missing(q)
)

d2_ij_v(
  L1,
  L2,
  m = 100L,
  p = m,
  q = m,
  thr_margin = 100,
  fill_all = !missing(p) || !missing(q)
)

d2_pj_m(A1, A2, m = 100L, p = 1L, thr_margin = 100)
```

```
d2_1j_m(A1, A2, m = 100L, thr_margin = 100)

d2_pj_v(L1, L2, m = 100L, p = 1L, thr_margin = 100)

d2_1j_v(L1, L2, m = 100L, thr_margin = 100)

h2_ij_m(
  A1,
  A2,
  mu = rep.int(0, n),
  m = 100L,
  p = m,
  q = m,
  thr_margin = 100,
  fill_all = !missing(p) || !missing(q)
)

h2_ij_v(
  L1,
  L2,
  mu = rep.int(0, n),
  m = 100L,
  p = m,
  q = m,
  thr_margin = 100,
  fill_all = !missing(p) || !missing(q)
)

htil2_pj_m(A1, A2, mu = rep.int(0, n), m = 100L, p = 1L, thr_margin = 100)

htil2_1j_m(A1, A2, mu = rep.int(0, n), m = 100L, thr_margin = 100)

htil2_pj_v(L1, L2, mu = rep.int(0, n), m = 100L, p = 1L, thr_margin = 100)

htil2_1j_v(L1, L2, mu = rep.int(0, n), m = 100L, thr_margin = 100)

hhat2_pj_m(A1, A2, mu = rep.int(0, n), m = 100L, p = 1L, thr_margin = 100)

hhat2_1j_m(A1, A2, mu = rep.int(0, n), m = 100L, thr_margin = 100)

hhat2_pj_v(L1, L2, mu = rep.int(0, n), m = 100L, p = 1L, thr_margin = 100)

hhat2_1j_v(L1, L2, mu = rep.int(0, n), m = 100L, thr_margin = 100)
```

Arguments

`A1, A2`	Argument matrices. Assumed to be symmetric and of the same order.

m	Integer-alike to specify the desired order along A2/L2
p, q	Integer-alikes to specify the desired orders along A1/L1 and A2/L2, respectively.
thr_margin	Optional argument to adjust the threshold for scaling (see "Scaling" in d1_i)
fill_all	Logical to specify whether all the output matrix should be filled. See "Details".
L1, L2	Eigenvalues of the argument matrices
mu	Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$

Details

d2_**_*() functions calculate $d_{i,j}(\mathbf{A}_1, \mathbf{A}_2)$ in Hillier et al. (2009, 2014) and Bao and Kan (2013). These are also related to the top-order invariant polynomials $C_{[k_1],[k_2]}(\mathbf{A}_1, \mathbf{A}_2)$ in the following way: $d_{i,j}(\mathbf{A}_1, \mathbf{A}_2) = \frac{1}{k_1!k_2!}\left(\frac{1}{2}\right)_{k_1+k_2} C_{[k_1],[k_2]}(\mathbf{A}_1, \mathbf{A}_2)$, where $(x)_k = x(x+1)\dots(x+k-1)$ (Chikuse 1987; Hillier et al. 2009).

h2_ij_*() and htil2_pj_*() functions calculate $h_{i,j}(\mathbf{A}_1, \mathbf{A}_2)$ and $\tilde{h}_{i,j}(\mathbf{A}_1; \mathbf{A}_2)$, respectively, in Bao and Kan (2013). Note that the latter is denoted by the symbol $h_{i,j}$ in Hillier et al. (2014). hhat2_pj_*() functions are for $\hat{h}_{i,j}(\mathbf{A}_1; \mathbf{A}_2)$ in Hillier et al. (2014), used to calculate an error bound for truncated sum for moments of a ratio of quadratic forms. The mean vector $\boldsymbol{\mu}$ is a parameter in all these.

There are two different situations in which these coefficients are used in calculation of moments of ratios of quadratic forms: **1**) within an infinite series for one of the subscripts, with the other subscript fixed (when the exponent p of the numerator is integer); **2**) within a double infinite series for both subscripts (when p is non-integer) (see Bao and Kan 2013). In this package, the situation **1** is handled by the *_pj_* (and *_1j_*) functions, and **2** is by the *_ij_* functions.

In particular, the *_pj_* functions always return a (p + 1) * (m + 1) matrix where all elements are filled with the relevant coefficients (e.g., $d_{i,j}$, $\tilde{h}_{i,j}$), from which, typically, the [p + 1,]-th row is used for subsequent calculations. (Those with *_1q_* are simply fast versions for the commonly used case where $p = 1$.) On the other hand, the *_ij_* functions by default return a (m + 1) * (m + 1) matrix whose upper-left triangular part (including the diagonals) is filled with the coefficients ($d_{i,j}$ or $h_{i,j}$), the rest being 0, and all the coefficients are used in subsequent calculations.

(At present, the *_ij_* functions also have the functionality to fill all coefficients of a potentially non-square output matrix, but this is less efficient than *_pj_* functions so may be omitted in the future development.)

Those ending with _m take matrices as arguments, whereas those with _v take eigenvalues. The latter can be used only when the argument matrices share the same eigenvectors, to which the eigenvalues correspond in the orders given, but is substantially faster.

This package also involves C++ equivalents for most of these functions (which are suffixed by E for Eigen), but these are exclusively for internal use and not exposed to the user.

Value

(p + 1) * (m + 1) matrix for the *_pj_* functions.

(m + 1) * (m + 1) matrix for the *_ij_* functions.

The rows and columns correspond to increasing orders for $\mathbf{A}_1$ and $\mathbf{A}_2$, respectively. And the 1st row/column of each dimension corresponds to the 0th order (hence [p + 1, q + 1] for the (p, q)-th order).

Has the attribute "logscale" as described in the “Scaling” section in d1_i. This is a matrix of the same size as the return itself.

References

Bao, Y. and Kan, R. (2013) On the moments of ratios of quadratic forms in normal random variables. *Journal of Multivariate Analysis*, **117**, 229–245. doi:10.1016/j.jmva.2013.03.002.

Chikuse, Y. (1987) Methods for constructing top order invariant polynomials. *Econometric Theory*, **3**, 195–207. doi:10.1017/S026646660001029X.

Hillier, G., Kan, R. and Wang, X. (2009) Computationally efficient recursions for top-order invariant polynomials with applications. *Econometric Theory*, **25**, 211–242. doi:10.1017/S0266466608090075.

Hillier, G., Kan, R. and Wang, X. (2014) Generating functions and short recursions, with applications to the moments of quadratic forms in noncentral normal vectors. *Econometric Theory*, **30**, 436–473. doi:10.1017/S0266466613000364.

See Also

qfrm and qfmrm are major front-end functions that utilize these functions

dtil2_pq for $\tilde{d}$ used for moments of a product of quadratic forms

d3_ijk for equivalents for three matrices

---

d3_ijk — *Coefficients in polynomial expansion of generating function—for ratios with three matrices*

Description

These are internal functions to calculate the coefficients in polynomial expansion of joint generating functions for three quadratic forms in potentially noncentral multivariate normal variables, $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \mathbf{I}_n)$. They are primarily used in calculations around moments of a ratio involving three quadratic forms.

Usage

```
d3_ijk_m(
  A1,
  A2,
  A3,
  m = 100L,
  p = m,
  q = m,
  r = m,
  thr_margin = 100,
  fill_across = c(!missing(p), !missing(q), !missing(r))
)
```

```
d3_ijk_v(
  L1,
  L2,
  L3,
  m = 100L,
  p = m,
  q = m,
  r = m,
  thr_margin = 100,
  fill_across = c(!missing(p), !missing(q), !missing(r))
)

d3_pjk_m(A1, A2, A3, m = 100L, p = 1L, thr_margin = 100)

d3_pjk_v(L1, L2, L3, m = 100L, p = 1L, thr_margin = 100)

h3_ijk_m(
  A1,
  A2,
  A3,
  mu = rep.int(0, n),
  m = 100L,
  p = m,
  q = m,
  r = m,
  thr_margin = 100,
  fill_across = c(!missing(p), !missing(q), !missing(r))
)

h3_ijk_v(
  L1,
  L2,
  L3,
  mu = rep.int(0, n),
  m = 100L,
  p = m,
  q = m,
  r = m,
  thr_margin = 100,
  fill_across = c(!missing(p), !missing(q), !missing(r))
)

htil3_pjk_m(A1, A2, A3, mu = rep.int(0, n), m = 100L, p = 1L, thr_margin = 100)

htil3_pjk_v(L1, L2, L3, mu = rep.int(0, n), m = 100L, p = 1L, thr_margin = 100)

hhat3_pjk_m(A1, A2, A3, mu = rep.int(0, n), m = 100L, p = 1L, thr_margin = 100)
```

```
hhat3_pjk_v(L1, L2, L3, mu = rep.int(0, n), m = 100L, p = 1L, thr_margin = 100)
```

Arguments

A1, A2, A3	Argument matrices. Assumed to be symmetric and of the same order.
m	Integer-alike to specify the desired order along A2/L2 and A3/L3
p, q, r	Integer-alikes to specify the desired orders along A1/L1, A2/L2, and A3/L3, respectively.
thr_margin	Optional argument to adjust the threshold for scaling (see "Scaling" in d1_i)
fill_across	Logical vector of length 3, to specify whether each dimension of the output matrix should be filled.
L1, L2, L3	Eigenvalues of the argument matrices
mu	Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$

Details

All these functions have equivalents for two-matrix cases (d2_ij), to which the user is referred for documentations. The primary difference of these functions from the latter is the addition of arguments for the third matrix A3/L3.

d3_*jk_*() functions calculate $d_{i,j,k}(\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3)$ in Hillier et al. (2009, 2014) and Bao and Kan (2013). These are also related to the top-order invariant polynomials as described in d2_ij.

h3_ijk_*(), htil3_pjk_*(), and hhat3_pjk_*() functions calculate $h_{i,j,k}(\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3)$, $\tilde{h}_{i;j,k}(\mathbf{A}_1; \mathbf{A}_2, \mathbf{A}_3)$, and $\hat{h}_{i;j,k}(\mathbf{A}_1; \mathbf{A}_2, \mathbf{A}_3)$, respectively, as described in the package vignette. These are equivalent to similar coefficients described in Bao and Kan (2013) and Hillier et al. (2014).

The difference between the *_pjk_* and *_ijk_* functions is as described for *_pj_* and *_ij_* (see "Details" in d2_ij). The only difference is that these functions return a 3D array. In the *_pjk_* functions, all the slices along the first dimension (i.e., [i, ,]) are an upper-left triangular matrix like what the *_ij_* functions return in the 2D case; in other words, the return has the coefficients for the terms that satisfy $j + k \leq m$ for all $i = 0, 1, \ldots, p$. Typically, the [p + 1, ,]-th slice is used for subsequent calculations. In the return of the *_ijk_* functions, only the triangular prism close to the [1, 1, 1] is filled with coefficients, which correspond to the terms satisfying $i + j + k \leq m$.

Value

(p + 1) * (m + 1) * (m + 1) array for the *_pjk_* functions

(m + 1) * (m + 1) * (m + 1) array for the *_ijk_* functions (by default; see "Details").

The 1st, 2nd, and 3rd dimensions correspond to increasing orders for $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$, respectively. And the 1st row/column of each dimension corresponds to the 0th order (hence [p + 1, q + 1, r + 1] for the (p, q, r)-th order).

Has the attribute "logscale" as described in the "Scaling" section in d1_i. This is an array of the same size as the return itself.

References

Bao, Y. and Kan, R. (2013) On the moments of ratios of quadratic forms in normal random variables. *Journal of Multivariate Analysis*, **117**, 229–245. doi:10.1016/j.jmva.2013.03.002.

Hillier, G., Kan, R. and Wang, X. (2014) Generating functions and short recursions, with applications to the moments of quadratic forms in noncentral normal vectors. *Econometric Theory*, **30**, 436–473. doi:10.1017/S0266466613000364.

See Also

`qfmrm` is a major front-end function that utilizes these functions

`dtil2_pq` for $\tilde{d}$ used for moments of a product of quadratic forms

`d2_ij` for equivalents for two matrices

`dqfr` *Probability distribution of ratio of quadratic forms*

Description

`dqfr()`: Density of the (power of) ratio of quadratic forms, $\left(\frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{B}\mathbf{x}}\right)^p$, where $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

`pqfr()`: Distribution function of the same.

`qqfr()`: Quantile function of the same.

`dqfr_A1I1()`: internal for `dqfr()`, exact series expression of Hillier (2001). Only accommodates the simple case where $\mathbf{B} = \mathbf{I}_n$ and $\boldsymbol{\mu} = \mathbf{0}_n$.

`dqfr_broda()`: internal for `dqfr()`, exact numerical inversion algorithm of Broda & Paolella (2009).

`dqfr_butler()`: internal for `dqfr()`, saddlepoint approximation of Butler & Paolella (2007, 2008).

`pqfr_A1B1()`: internal for `pqfr()`, exact series expression of Forchini (2002, 2005).

`pqfr_imhof()`: internal for `pqfr()`, exact numerical inversion algorithm of Imhof (1961).

`pqfr_davies()`: internal for `pqfr()`, exact numerical inversion algorithm of Davies (1973, 1980). This is **experimental** and may be removed in the future.

`pqfr_butler()`: internal for `pqfr()`, saddlepoint approximation of Butler & Paolella (2007, 2008).

The user is supposed to use the exported functions `dqfr()`, `pqfr()`, and `qqfr()`, which are (pseudo-)vectorized with respect to `quantile` or `probability`. The actual calculations are done by one of the internal functions, which only accommodate a length-one `quantile`. The internal functions skip most checks on argument structures and do not accommodate `Sigma` to reduce execution time.

Usage

```
dqfr(
  quantile,
  A,
  B,
  p = 1,
  mu = rep.int(0, n),
  Sigma = diag(n),
  log = FALSE,
  method = c("broda", "hillier", "butler"),
  trim_values = TRUE,
  normalize_spa = FALSE,
  return_abserr_attr = FALSE,
  m = 100L,
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  ...
)

pqfr(
  quantile,
  A,
  B,
  p = 1,
  mu = rep.int(0, n),
  Sigma = diag(n),
  lower.tail = TRUE,
  log.p = FALSE,
  method = c("imhof", "davies", "forchini", "butler"),
  trim_values = TRUE,
  return_abserr_attr = FALSE,
  m = 100L,
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  ...
)

qqfr(
  probability,
  A,
  B,
  p = 1,
  mu = rep.int(0, n),
  Sigma = diag(n),
  lower.tail = TRUE,
  log.p = FALSE,
  trim_values = FALSE,
  return_abserr_attr = FALSE,
```

```
  stop_on_error = FALSE,
  m = 100L,
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  epsabs_q = .Machine$double.eps^(1/2),
  maxiter_q = 5000,
  ...
)

dqfr_A1I1(
  quantile,
  LA,
  m = 100L,
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  tol_conv = .Machine$double.eps^(1/4),
  thr_margin = 100
)

dqfr_broda(
  quantile,
  A,
  B,
  mu = rep.int(0, n),
  autoscale_args = 1,
  stop_on_error = TRUE,
  use_cpp = TRUE,
  tol_zero = .Machine$double.eps * 100,
  epsabs = epsrel,
  epsrel = 1e-06,
  limit = 10000
)

dqfr_butler(
  quantile,
  A,
  B,
  mu = rep.int(0, n),
  order_spa = 2,
  stop_on_error = FALSE,
  use_cpp = TRUE,
  tol_zero = .Machine$double.eps * 100,
  epsabs = .Machine$double.eps^(1/2),
  epsrel = 0,
  maxiter = 5000
)

pqfr_A1B1(
```

```
  quantile,
  A,
  B,
  m = 100L,
  mu = rep.int(0, n),
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  stop_on_error = FALSE,
  use_cpp = TRUE,
  cpp_method = c("double", "long_double", "coef_wise"),
  nthreads = 1,
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
  thr_margin = 100
)

pqfr_imhof(
  quantile,
  A,
  B,
  mu = rep.int(0, n),
  autoscale_args = 1,
  stop_on_error = TRUE,
  use_cpp = TRUE,
  tol_zero = .Machine$double.eps * 100,
  epsabs = epsrel,
  epsrel = 1e-06,
  limit = 10000
)

pqfr_davies(
  quantile,
  A,
  B,
  mu = rep.int(0, n),
  autoscale_args = 1,
  stop_on_error = NULL,
  tol_zero = .Machine$double.eps * 100,
  ...
)

pqfr_butler(
  quantile,
  A,
  B,
  mu = rep.int(0, n),
  order_spa = 2,
  stop_on_error = FALSE,
  use_cpp = TRUE,
```

```
  tol_zero = .Machine$double.eps * 100,
  epsabs = .Machine$double.eps^(1/2),
  epsrel = 0,
  maxiter = 5000
)
```

Arguments

quantile
: Numeric vector of quantiles q

A, B
: Argument matrices. Should be square. B should be nonnegative definite. Will be automatically symmetrized in dqfr() and pqfr().

p
: Positive exponent of the ratio, default 1. Unlike in qfrm(), the numerator and denominator cannot have different exponents. When p is non-integer, A must be nonnegative definite. For details, see vignette vignette("qfratio_distr").

mu
: Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$

Sigma
: Covariance matrix $\boldsymbol{\Sigma}$ for $\mathbf{x}$

log, lower.tail, log.p
: Logical; as in regular probability distribution functions. But these are for convenience only, and not meant for accuracy.

method
: Method to specify an internal function (see "Details"). In dqfr(), options are:

  "broda" default; uses dqfr_broda(), numerical inversion of Broda & Paolella (2009)

  "hillier" uses dqfr_A1I1(), series expression of Hillier (2001)

  "butler" uses dqfr_butler(), saddlepoint approximation of Butler & Paolella (2007, 2008)

  In pqfr(), options are:

  "imhof" default; uses pqfr_imhof(), numerical inversion of Imhof (1961)

  "davies" uses pqfr_davies(), numerical inversion of Davies (1973, 1980)

  "forchini" uses pqfr_A1B1(), series expression of Forchini (2002, 2005)

  "butler" uses pqfr_butler(), saddlepoint approximation of Butler & Paolella (2007, 2008)

trim_values
: If TRUE (default), numerical values outside the mathematically permissible support are trimmed in (see "Details")

normalize_spa
: If TRUE and method == "butler", result is normalized so that the density integrates to unity (see "Details")

return_abserr_attr
: If TRUE, absolute error of numerical evaluation is returned as an attribute "abserr" (see "Value")

m
: Order of polynomials at which the series expression is truncated. M in Hillier et al. (2009, 2014).

tol_zero
: Tolerance against which numerical zero is determined. Used to determine, e.g., whether mu is a zero vector, A or B equals the identity matrix, etc.

tol_sing
: Tolerance against which matrix singularity and rank are determined. The eigenvalues smaller than this are considered zero.

`...` Additional arguments passed to internal functions. In `qqfr()`, these are passed to `pqfr()`.

`probability` Numeric vector of probabilities

`stop_on_error` If `TRUE`, execution is stopped upon an error (including non-convergence) in evaluation of hypergeometric function, numerical integration, or root finding. If `FALSE`, further execution is attempted regardless.

`LA` Eigenvalues of $\mathbf{A}$

`check_convergence`
Specifies how numerical convergence is checked for series expression (see `qfrm`)

`use_cpp` Logical to specify whether the calculation is done with C++ functions via `Rcpp`. `TRUE` by default.

`tol_conv` Tolerance against which numerical convergence of series is checked. Used with `check_convergence`.

`thr_margin` Optional argument to adjust the threshold for scaling (see "Scaling" in `d1_i`). Passed to internal functions (`d1_i`, `d2_ij`, `d3_ijk`) or their C++ equivalents.

`autoscale_args` Numeric; if > `0` (default), arguments are scaled to avoid failure in numerical integration (see `vignette("qfratio_distr")`). If <= `0`, the scaling is skipped.

`epsabs`, `epsrel`, `limit`, `maxiter`, `epsabs_q`, `maxiter_q`
Optional arguments used in numerical integration or root-finding algorithm (see vignette: `vignette("qfratio_distr")`). In `qqfr()`, `epsabs_q` and `maxiter_q` are used in root-finding for quantiles whereas `epsabs` and `maxiter` are passed to `pqfr()` internally.

`order_spa` Numeric to determine order of saddlepoint approximation. More accurate second-order approximation is used for any `order` > `1` (default); otherwise, (very slightly) faster first-order approximation is used.

`cpp_method` Method used in C++ calculations to avoid numerical overflow/underflow (see "Details" in `qfrm`)

`nthreads` Number of threads used in OpenMP-enabled C++ functions (see "Multithreading" in `qfrm`)

Details

`qqfr()` is based on numerical root-finding with `pqfr()` using `uniroot()`, so its result can be affected by the numerical errors in both the algorithm used in `pqfr()` and root-finding.

`dqfr_A1I1()` and `pqfr_A1B1()` evaluate the probability density and (cumulative) distribution function, respectively, as a partial sum of infinite series involving top-order zonal or invariant polynomials (Hillier 2001; Forchini 2002, 2005). As in other functions of this package, these are evaluated with the recursive algorithm `d1_i`.

`pqfr_imhof()` and `pqfr_davies()` evaluate the distribution function by numerical inversion of the characteristic function based on Imhof (1961) or Davies (1973, 1980), respectively. The latter calls `davies()`, and the former with `use_cpp = FALSE` calls `imhof()`, from the package **CompQuadForm**. Additional arguments for `davies()` can be passed via `...`, except for `sigma`, which is not applicable.

dqfr_broda() evaluates the probability density by numerical inversion of the characteristic function using Geary's formula based on Broda & Paolella (2009). Parameters for numerical integration can be controlled via the arguments epsabs, epsrel, and limit (see vignette: vignette("qfratio_distr")).

dqfr_butler() and pqfr_butler() evaluate saddlepoint approximations of the density and distribution function, respectively, based on Butler & Paolella (2007, 2008). These are fast but not exact. They conduct numerical root-finding for the saddlepoint by the Brent method, parameters for which can be controlled by the arguments epsabs, epsrel, and maxiter (see vignette: vignette("qfratio_distr")). The saddlepoint approximation density does not integrate to unity, but can be normalized by dqfr(..., method = "butler", normalize_spa = TRUE). Note that this is usually slower than dqfr(..., method = "broda") for a small number of quantiles.

The density is undefined, and the distribution function has points of nonanalyticity, at the eigenvalues of $\mathbf{B}^{-1}\mathbf{A}$ (assuming nonsingular $\mathbf{B}$). Around these points, the series expressions tends to fail. Avoid using the series expression methods for these cases.

Algorithms based on numerical integration can yield spurious results that are outside the mathematically permissible support; e.g., $[0, 1]$ for pqfr(). By default, dqfr() and pqfr() trim those values into the permissible range with a warning; e.g., negative p-values are replaced by ~2.2e-14 (default tol_zero). Turn trim_values = FALSE to skip these trimming and warning, although pqfr_imhof() and pqfr_davies() can still throw a warning from **CompQuadForm** functions. Note that, on the other hand, all these functions try to return exact 0 or 1 when q is outside the possible range of the statistic.

Value

dqfr() and pqfr() give the density and distribution (or p-values) functions, respectively, corresponding to quantile, whereas qqfr() gives the quantile function corresponding to probability.

When return_abserr_attr = TRUE, an absolute error bound of numerical evaluation is returned as an attribute; this feature is currently available with dqfr(..., method = "broda"), pqfr(..., method = "imhof"), and qqfr(..., method = "imhof") (all default) only. This error bound is automatically transformed when trimming happens with trim_values (above) or when log/log.p = TRUE. See vignette for details (vignette("qfratio_distr")).

The internal functions return a list containing $d or $p (for density and lower p-value, respectively), and only this is passed to the external function by default. Other components may be inspected for debugging purposes:

dqfr_A1I1() **and** pqfr_A1B1() have $terms, a vector of 0th to mth order terms.

pqfr_imhof() **and** dqfr_broda() have $abserr, absolute error of numerical integration; the one returned from CompQuadForm::imhof() is divided by pi, as the integration result itself is (internally). This is passed to the external functions when return_abserr_attr = TRUE (above).

pqfr_davies() has the same components as CompQuadForm::davies() apart from Qq which is replaced by p = 1 - Qq.

References

Broda, S. and Paolella, M. S. (2009) Evaluating the density of ratios of noncentral quadratic forms in normal variables. *Computational Statistics and Data Analysis*, **53**, 1264–1270. doi:10.1016/j.csda.2008.10.035

Butler, R. W. and Paolella, M. S. (2007) Uniform saddlepoint approximations for ratios of quadratic forms. Technical Reports, Department of Statistical Science, Southern Methodist University, no. **351**. [Available on *arXiv* as a preprint.] doi:10.48550/arXiv.0803.2132

Butler, R. W. and Paolella, M. S. (2008) Uniform saddlepoint approximations for ratios of quadratic forms. *Bernoulli*, **14**, 140–154. doi:10.3150/07BEJ6169

Davis, R. B. (1973) Numerical inversion of a characteristic function. *Biometrika*, **60**, 415–417. doi:10.1093/biomet/60.2.415.

Davis, R. B. (1980) Algorithm AS 155: The distribution of a linear combination of χ^2 random variables. *Journal of the Royal Statistical Society, Series C—Applied Statistics*, **29**, 323–333. doi:10.2307/2346911.

Forchini, G. (2002) The exact cumulative distribution function of a ratio of quadratic forms in normal variables, with application to the AR(1) model. *Econometric Theory*, **18**, 823–852. doi:10.1017/S0266466602184015.

Forchini, G. (2005) The distribution of a ratio of quadratic forms in noncentral normal variables. *Communications in Statistics—Theory and Methods*, **34**, 999–1008. doi:10.1081/STA200056855.

Hillier, G. (2001) The density of quadratic form in a vector uniformly distributed on the n-sphere. *Econometric Theory*, **17**, 1–28. doi:10.1017/S026646660117101X.

Imhof, J. P. (1961) Computing the distribution of quadratic forms in normal variables. *Biometrika*, **48**, 419–426. doi:10.1093/biomet/48.34.419.

See Also

`rqfr`, a Monte Carlo random number generator

`vignette("qfratio_distr")` for theoretical details

Examples

```
## Some symmetric matrices and parameters
nv <- 4
A <- diag(nv:1)
B <- diag(sqrt(1:nv))
mu <- 0.2 * nv:1
Sigma <- matrix(0.5, nv, nv)
diag(Sigma) <- 1

## density and p-value for (x^T A x) / (x^T x) where x ~ N(0, I)
dqfr(1.5, A)
pqfr(1.5, A)

## 95 percentile for the same
qqfr(0.95, A)
qqfr(0.05, A, lower.tail = FALSE) # same

## P{ (x^T A x) / (x^T B x) <= 1.5} where x ~ N(mu, Sigma)
pqfr(1.5, A, B, mu = mu, Sigma = Sigma)

## These are (pseudo-)vectorized
qs <- 0:nv + 0.5
```

```
dqfr(qs, A, B, mu = mu)
(pres <- pqfr(qs, A, B, mu = mu))

## Quantiles for above p-values
## Results equal qs, except that those for prob = 0 and 1
## are replaced by mininum and maximum of the ratio
qqfr(pres, A, B, mu = mu) # = qs

## Various methods for density
dqfr(qs, A, method = "broda")   # default
dqfr(qs, A, method = "hillier") # series; B, mu, Sigma not permitted
## Saddlepoint approximations (fast but inexact):
dqfr(qs, A, method = "butler")  # 2nd order by default
dqfr(qs, A, method = "butler", normalize_spa = TRUE) # normalized
dqfr(qs, A, method = "butler", normalize_spa = TRUE,
     order_spa = 1) # 1st order, normalized

## Various methods for distribution function
pqfr(qs, A, method = "imhof")    # default
pqfr(qs, A, method = "davies")   # very similar
pqfr(qs, A, method = "forchini") # series expression
pqfr(qs, A, method = "butler")   # saddlepoint approximation (2nd order)
pqfr(qs, A, method = "butler", order_spa = 1) # 1st order

## To see error bounds
dqfr(qs, A, return_abserr_attr = TRUE)
pqfr(qs, A, return_abserr_attr = TRUE)
qqfr(pres, A, return_abserr_attr = TRUE)
```

---

dtil2_pq *Coefficients in polynomial expansion of generating function—for products*

Description

These are internal functions to calculate the coefficients in polynomial expansion of joint generating functions for two or three quadratic forms in potentially noncentral multivariate normal variables, $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \mathbf{I}_n)$. They are primarily used to calculate moments of a product of two or three quadratic forms.

Usage

```
dtil2_pq_m(A1, A2, mu = rep.int(0, n), p = 1L, q = 1L)

dtil2_1q_m(A1, A2, mu = rep.int(0, n), q = 1L)

dtil2_pq_v(L1, L2, mu = rep.int(0, n), p = 1L, q = 1L)
```

```
dtil2_1q_v(L1, L2, mu = rep.int(0, n), q = 1L)

dtil3_pqr_m(A1, A2, A3, mu = rep.int(0, n), p = 1L, q = 1L, r = 1L)

dtil3_pqr_v(L1, L2, L3, mu = rep.int(0, n), p = 1L, q = 1L, r = 1L)
```

Arguments

A1, A2, A3	Argument matrices. Assumed to be symmetric and of the same order.
mu	Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$
p, q, r	Integer-alikes to specify the order along the three argument matrices
L1, L2, L3	Eigenvalues of the argument matrices

Details

dtil2_pq_m() and dtil2_pq_v() calculate $\tilde{d}_{p,q}(\mathbf{A}_1, \mathbf{A}_2)$ in Hillier et al. (2014). dtil2_1q_m() and dtil2_1q_v() are fast versions for the commonly used case where $p = 1$. Similarly, dtil3_pqr_m() and dtil3_pqr_v() are for $\tilde{d}_{p,q,r}(\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3)$.

Those ending with _m take matrices as arguments, whereas those with _v take eigenvalues. The latter can be used only when the argument matrices share the same eigenvectors, to which the eigenvalues correspond in the orders given, but is substantially faster.

These functions calculate the coefficients based on the super-short recursion algorithm described in Hillier et al. (2014: sec. 4.2).

Value

A (p + 1) * (q + 1) matrix for the 2D functions, or a (p + 1) * (q + 1) * (r + 1) array for the 3D functions.

The 1st, 2nd, and 3rd dimensions correspond to increasing orders for $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$, respectively. And the 1st row/column of each dimension corresponds to the 0th order (hence [p + 1, q + 1] for the (p, q)-th moment).

References

Hillier, G., Kan, R. and Wang, X. (2014) Generating functions and short recursions, with applications to the moments of quadratic forms in noncentral normal vectors. *Econometric Theory*, **30**, 436–473. doi:10.1017/S0266466613000364.

See Also

qfpm is a front-end functions that utilizes these functions

d1_i for a single-matrix equivalent of $\tilde{d}$

hgs *Calculate hypergeometric series*

Description

These internal functions calculate (summands of) hypergeometric series.

hgs_1d() calculates the hypergeometric series $c\frac{(a_1)_i}{(b)_i}d_i$

hgs_2d() calculates the hypergeometric series $c\frac{(a_1)_i(a_2)_j}{(b)_{i+j}}d_{i,j}$

hgs_3d() calculates the hypergeometric series $c\frac{(a_1)_i(a_2)_j(a_3)_k}{(b)_{i+j+k}}d_{i,j,k}$

Usage

```
hgs_1d(dks, a1, b, lconst = 0)

hgs_2d(dks, a1, a2, b, lconst = 0)

hgs_3d(dks, a1, a2, a3, b, lconst = 0)
```

Arguments

dks	(m + 1) vector for d_i, (m + 1) * (m + 1) square matrix for $d_{i,j}$, or (m + 1) * (m + 1) * (m + 1) array for $d_{i,j,k}$ $(i, j, k = 0, 1, \dots m)$
a1, a2, a3	Numerator parameters
b	Denominator parameter
lconst	Scalar $\log c$

Details

The denominator parameter b is assumed positive, whereas the numerator parameters can be positive or negative. The signs of the latter will be reflected in the result.

Value

Numeric with the same dimension with dks

hyperg_1F1_vec_b *Internal C++ wrappers for GSL*

Description

These are internal C++ functions which wrap hypergeometric functions from GSL with vectorization. These are for particular use cases in this package, and direct access by the user is **not** assumed.

Usage

```
hyperg_1F1_vec_b(a, bvec, x)

hyperg_2F1_mat_a_vec_c(Amat, b, cvec, x)
```

Arguments

`a, b`	Parameters of hypergeometric functions; passed as `double`
`bvec, cvec`	Parameters of hypergeometric functions; passed as `Rcpp::NumericVector`
`x`	Argument of hypergeometric functions; passed as `double`
`Amat`	Parameter of hypergeometric functions; passed as `Rcpp::NumericMatrix`. Dimension must be square of the length of `cvec`.

Value

Return a list via `Rcpp::List` of the following:

`$val` Evaluation result, numeric

`$err` Absolute error, numeric

`$status` Error code, integer

In `hyperg_1F1_vec_b`, these are vectors from `Rcpp::NumericVector` and `Rcpp::IntegerVector`, whereas in `hyperg_2F1_mat_a_vec_c`, they are matrices from `Rcpp::NumericMatrix` and `Rcpp::IntegerMatrix`.

Functions

- `hyperg_1F1_vec_b()`: wrapper of `gsl_hyperg_1F1_e()`, looping along `bvec`
- `hyperg_2F1_mat_a_vec_c()`: wrapper of `gsl_hyperg_2F1_e()`, looping along `Amat` and recycling `cvec`

`iseq` *Are these vectors equal?*

Description

This internal function is used to determine whether two vectors/matrices have the same elements (or, a vector/matrix is all equal to 0) using `all.equal()`. Attributes and dimensions are ignored as they are passed as vectors using `c()`.

Usage

```
iseq(x, y = rep.int(0, length(x)), tol = .Machine$double.eps * 100)
```

Arguments

`x`	Main `target` vector/matrix in `all.equal()`
`y`	`current` in `all.equal()`. Default zero vector.
`tol`	Numeric to specify `tolerance` in `all.equal()`

See Also

`all.equal`

`is_diagonal` *Is this matrix diagonal?*

Description

This internal function is used to determine whether a square matrix is diagonal (within a specified tolerance). Returns `TRUE` when the absolute values of all off-diagonal elements are below `tol`, using `all.equal()`.

Usage

```
is_diagonal(A, tol = .Machine$double.eps * 100, symmetric = FALSE)
```

Arguments

`A`	Square matrix. No check is done.
`tol`	Numeric to specify `tolerance` in `all.equal()`
`symmetric`	If `FALSE` (default), sum of absolute values of the corresponding lower and upper triangular elements are examined with a doubled `tol`. If `TRUE`, only the lower triangular elements are examined assuming symmetry.

See Also

`all.equal`

KiK *Matrix square root and generalized inverse*

Description

This internal function calculates the decomposition $\mathbf{S} = \mathbf{K}\mathbf{K}^T$ for an $n \times n$ covariance matrix $\mathbf{S}$, so that $\mathbf{K}$ is an $n \times m$ matrix with m being the rank of $\mathbf{S}$. Returns this $\mathbf{K}$ and its generalized inverse, $\mathbf{K}^-$, in a list.

Usage

```
KiK(S, tol = .Machine$double.eps * 100)
```

Arguments

S	Covariance matrix. Symmetry and positive (semi-)definiteness are checked.
tol	Tolerance to determine the rank of $\mathbf{S}$. Eigenvalues smaller than this value are considered zero.

Details

At present, this utilizes `svd()`, although there may be better alternatives.

Value

List with `K` and `iK`, with the latter being $\mathbf{K}^-$

new_qfrm *Construct qfrm object*

Description

These are internal "constructor" functions used to make `qfrm` and `qfpm` objects, which are used as a return value from the `qfrm`, `qfmrm`, and `qfpm` functions.

Usage

```
new_qfrm(
  statistic,
  error_bound = NULL,
  terms = statistic,
  seq_error = NULL,
  exact = FALSE,
  twosided = FALSE,
  alphaout = FALSE,
```

```
  singular_arg = FALSE,
  diminished = FALSE,
  ...,
  class = character()
)

new_qfpm(statistic, exact = TRUE, ..., class = character())
```

Arguments

`statistic`	Terminal value (partial sum) for the moment. When missing, obtained as `sum(terms)`.
`error_bound`	Terminal error bound. When missing, obtained as `seq_error[length(seq_error)]`.
`terms`	Terms in series expression for the moment along varying polynomial degrees
`seq_error`	Vector of error bounds corresponding to `cumsum(terms)`
`exact, twosided, alphaout, singular_arg`	Logicals used to append attributes to the resultant error bound (see "Value")
`diminished`	Logical used to append attribute to the resultant statistic and terms (see "Value")
`...`	Additional arguments for accommodating subclasses
`class`	Character vector to (pre-)append classes to the return value

Value

`new_qfrm()` and `new_qfpm()` return a list of class `qfrm` and `c(qfpm, qfrm)`, respectively. These classes are defined for the `print` and `plot` methods.

The return object is a list of 4 elements which are intended to be:

`$statistic` evaluation result (`sum(terms)`)

`$terms` vector of 0th to mth order terms

`$error_bound` error bound of `statistic`

`$seq_error` vector of error bounds corresponding to partial sums (`cumsum(terms)`)

When the result is exact, `$terms` can be of length 1 and equal to `$statistic`. This is always the case for the `qfpm` class.

When the relevant flags are provided in the constructor, `$error_bound` and `$seq_error` have the following attributes which control behaviors of the `print` and `plot` methods:

`"exact"` indicates whether the moment is exact

`"twosided"` indicates whether the error bounds are two-sided

`"alphaout"` indicates whether any of the scaling factors (`alphaA`, `alphaB`, `alphaD`) is outside $(0, 1]$, when error bound does not strictly hold

`"singular"` indicates whether the relevant argument matrix is (numerically) singular, in which case the error bound is invalid

Similarly, when `diminished = TRUE`, `$statistic` and `$terms` have the attribute `"diminished"` being `TRUE`, which indicates that numerical underflow/diminishing happened during scaling (see "Scaling" in d1_i).

See Also

qfrm, qfmrm, qfpm: functions that return objects of these classes

methods.qfrm: the print and plot methods

---

print.qfrm *Methods for qfrm and qfpm objects*

Description

Straightforward print and plot methods are defined for qfrm and qfpm objects which result from the qfrm, qfmrm, and qfpm functions.

Usage

```
## S3 method for class 'qfrm'
print(
  x,
  digits = getOption("digits"),
  show_range = !is.null(x$error_bound),
  ...
)

## S3 method for class 'qfpm'
print(x, digits = getOption("digits"), ...)

## S3 method for class 'qfrm'
plot(
  x,
  add_error = length(x$seq_error) > 0,
  add_legend = add_error,
  ylim = x$statistic * ylim_f,
  ylim_f = c(0.9, 1.1),
  xlab = "Order of evaluation",
  ylab = "Moment of ratio",
  col_m = "royalblue4",
  col_e = "tomato",
  lwd_m = 1,
  lwd_e = 1,
  lty_m = 1,
  lty_e = 2,
  pos_leg = "topright",
  ...
)
```

Arguments

x	qfrm or qfpm object
digits	Number of significant digits to be printed.
show_range	Logical to specify whether the possible range for the moment is printed (when available). Default TRUE when available.
...	In the plot methods, passed to plot.default. In the print methods, ignored (retained for the compatibility with the generic method).
add_error	Logical to specify whether the sequence of error bounds is plotted (when available). Default TRUE when available.
add_legend	Logical to specify whether a legend is added. Turned on by default when add_error = TRUE.
ylim, ylim_f	ylim is passed to plot.default. By default, this is automatically set to ylim_f times the terminal value of the series expression (x$statistic). ylim_f is by default c(0.9, 1.1).
xlab, ylab	Passed to plot.default
col_m, col_e, lwd_m, lwd_e, lty_m, lty_e	col, lwd, and lty to plot the sequences of the moment (***_m) and its error bound (***_e)
pos_leg	Position of the legend, e.g., "topright", "bottomright", passed as the first argument for legend

Details

The print methods simply display the moment x$statistic (typically a partial sum), its error bound x$error_bound (when available), and the possible range of the moment (x$statistic to x$statistic + x$error_bound in case of one-sided error bound; x$statistic - x$error_bound to x$statistic + x$error_bound in case of two-sided).

The plot method is designed for quick inspection of the profile of the partial sum of the series along varying orders cumsum(x$terms). When the object has a sequence for error bounds x$seq_error, this is also shown with a broken line (by default). When the object has an exact moment (i.e., resulting from qfrm_ApIq_int() or the qfpm functions), a message is thrown to tell inspection of the plot will not be required in this case.

Value

The print method invisibly returns the input.

The plot method is used for the side effect (and invisibly returns NULL).

See Also

new_qfrm: descriptions of the classes and their "constructors"

Examples

```
nv <- 4
A <- diag(nv:1)
B <- diag(1:nv)
mu <- rep.int(1, nv)

res1 <- qfrm(A, B, p = 3, mu = mu)
print(res1)
print(res1, digits = 5)
print(res1, digits = 10)

## Default plot: ylim too narrow to see the error bound at this m
plot(res1)

## With extended ylim
plot(res1, ylim_f = c(0.8, 1.2), pos_leg = "topleft")

## In this case, it is easy to increase m
(res2 <- qfrm(A, B, p = 3, mu = mu, m = 200))
plot(res2)
```

---

p_A1B1_Ed *Internal C++ functions*

Description

These are internal C++ functions called from corresponding R functions when use_cpp = TRUE. Direct access by the user is **not** assumed. All parameters are assumed to be appropriately structured.

Usage

```
p_A1B1_Ed(
  quantile,
  A,
  B,
  mu,
  m,
  stop_on_error,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

p_A1B1_El(
  quantile,
  A,
  B,
```

```
    mu,
    m,
    stop_on_error,
    thr_margin = 100L,
    nthreads = 0L,
    tol_zero = 2.2e-14
  )

  p_A1B1_Ec(
    quantile,
    A,
    B,
    mu,
    m,
    stop_on_error,
    thr_margin = 100,
    nthreads = 0L,
    tol_zero = 2.2e-14
  )

  d_A1I1_Ed(quantile, LA, m, thr_margin = 100)

  p_imhof_Ed(
    quantile,
    A,
    B,
    mu,
    autoscale_args,
    stop_on_error,
    tol_zero,
    epsabs,
    epsrel,
    limit
  )

  d_broda_Ed(
    quantile,
    A,
    B,
    mu,
    autoscale_args,
    stop_on_error,
    tol_zero,
    epsabs,
    epsrel,
    limit
  )
```

```
d_butler_Ed(
  quantile,
  A,
  B,
  mu,
  order_spa,
  stop_on_error,
  tol_zero,
  epsabs,
  epsrel,
  maxiter
)

p_butler_Ed(
  quantile,
  A,
  B,
  mu,
  order_spa,
  stop_on_error,
  tol_zero,
  epsabs,
  epsrel,
  maxiter
)

Ap_int_E(A, mu, p_ = 1, thr_margin = 100, tol_zero = 2.2e-14)

ABpq_int_E(A, LB, mu, p_ = 1, q_ = 1, thr_margin = 100, tol_zero = 2.2e-14)

ABDpqr_int_E(
  A,
  LB,
  D,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  thr_margin = 100,
  tol_zero = 2.2e-14
)

ApIq_int_cE(A, p_ = 1, q_ = 1, thr_margin = 100)

ApIq_int_nE(A, mu, p_ = 1, q_ = 1, thr_margin = 100)

ApIq_npi_cE(
  LA,
```

```
    bA,
    p_ = 1,
    q_ = 1,
    m = 100L,
    error_bound = TRUE,
    thr_margin = 100
  )

  ApIq_npi_nEd(
    LA,
    bA,
    mu,
    p_ = 1,
    q_ = 1,
    m = 100L,
    thr_margin = 100,
    nthreads = 1L
  )

  ApBq_int_E(
    A,
    LB,
    bB,
    mu,
    p_ = 1,
    q_ = 1,
    m = 100L,
    error_bound = TRUE,
    thr_margin = 100,
    tol_zero = 2.2e-14
  )

  ApBq_npi_Ed(
    A,
    LB,
    bA,
    bB,
    mu,
    p_ = 1,
    q_ = 1,
    m = 100L,
    thr_margin = 100,
    nthreads = 0L,
    tol_zero = 2.2e-14
  )

  ApBIqr_int_cEd(
    A,
```

```
  LB,
  bB,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  error_bound = TRUE,
  thr_margin = 100,
  tol_zero = 2.2e-14
)

ApBIqr_int_nEd(
  A,
  LB,
  bB,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  error_bound = TRUE,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBIqr_npi_Ed(
  A,
  LB,
  bA,
  bB,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

IpBDqr_gen_Ed(
  LB,
  D,
  bB,
  bD,
  mu,
  p_ = 1,
```

```
  q_ = 1,
  r_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBDqr_int_Ed(
  A,
  LB,
  D,
  bB,
  bD,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBDqr_npi_Ed(
  A,
  LB,
  D,
  bA,
  bB,
  bD,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApIq_npi_nEc(
  LA,
  bA,
  mu,
  p_ = 1,
  q_ = 1,
  m = 100L,
```

```
  thr_margin = 100,
  nthreads = 1L
)

ApBq_npi_Ec(
  A,
  LB,
  bA,
  bB,
  mu,
  p_ = 1,
  q_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBIqr_int_nEc(
  A,
  LB,
  bB,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  error_bound = TRUE,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBIqr_npi_Ec(
  A,
  LB,
  bA,
  bB,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)
```

```
IpBDqr_gen_Ec(
  LB,
  D,
  bB,
  bD,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBDqr_int_Ec(
  A,
  LB,
  D,
  bB,
  bD,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBDqr_npi_Ec(
  A,
  LB,
  D,
  bA,
  bB,
  bD,
  mu,
  p_ = 1,
  q_ = 1,
  r_ = 1,
  m = 100L,
  thr_margin = 100,
  nthreads = 0L,
  tol_zero = 2.2e-14
)
```

```
ApIq_npi_nEl(
  LA,
  bA,
  mu,
  p_ = 1L,
  q_ = 1L,
  m = 100L,
  thr_margin = 100L,
  nthreads = 1L
)

ApBq_npi_El(
  A,
  LB,
  bA,
  bB,
  mu,
  p_ = 1L,
  q_ = 1L,
  m = 100L,
  thr_margin = 100L,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBIqr_int_nEl(
  A,
  LB,
  bB,
  mu,
  p_ = 1L,
  q_ = 1L,
  r_ = 1L,
  m = 100L,
  error_bound = TRUE,
  thr_margin = 100L,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBIqr_npi_El(
  A,
  LB,
  bA,
  bB,
  mu,
  p_ = 1L,
  q_ = 1L,
```

```
  r_ = 1L,
  m = 100L,
  thr_margin = 100L,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

IpBDqr_gen_El(
  LB,
  D,
  bB,
  bD,
  mu,
  p_ = 1L,
  q_ = 1L,
  r_ = 1L,
  m = 100L,
  thr_margin = 100L,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBDqr_int_El(
  A,
  LB,
  D,
  bB,
  bD,
  mu,
  p_ = 1L,
  q_ = 1L,
  r_ = 1L,
  m = 100L,
  thr_margin = 100L,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

ApBDqr_npi_El(
  A,
  LB,
  D,
  bA,
  bB,
  bD,
  mu,
  p_ = 1L,
  q_ = 1L,
```

```
  r_ = 1L,
  m = 100L,
  thr_margin = 100L,
  nthreads = 0L,
  tol_zero = 2.2e-14
)

rqfpE(nit, A, B, D, p_, q_, r_, mu, Sigma)
```

Arguments

quantile	Scalar of quantile q, passed as double
A, B, D	Argument matrices passed as Eigen::Matrix. Symmetry is assumed.
mu	Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$ passed as Eigen::Array. For d_broda_Ed(), assumed to be rotated by the eigenvectors of $\mathbf{A} - q\mathbf{B}$
m	Integer to specify the order of polynomials at which the series expression is truncated. Passed as Eigen::Index (aka std::ptrdiff_t or long long int)
stop_on_error	bool to specify whether execution is stopped upon error in numerical integration or root finding
thr_margin	Optional argument to adjust the threshold for scaling. See "Scaling" in d1_i.
nthreads	int to specify the number of threads in OpenMP-enabled functions. See "Multithreading" in qfrm.
tol_zero	Tolerance against which numerical zero is determined
LA, LB	Eigenvalues of the argument matrices passed as Eigen::Array
autoscale_args	Factor to which the largest absolute eigenvalue of $\mathbf{A} - q\mathbf{B}$ is scaled, passed as double
epsrel, epsabs, limit, maxiter	Optional arguments passed to gsl_integration_qagi() or gsl_root_test_delta()
order_spa	int to specify order of saddlepoint approximation
p_, q_, r_	Exponents for $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$. Passed as double or long double.
bA, bB, bD	Scaling coefficients for $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$. Passed as double or long double.
error_bound	bool to specify whether the error bound is returned
nit	int to specify the number of iteration or sample size
Sigma	Covariance matrix $\boldsymbol{\Sigma}$ for $\mathbf{x}$. Passed as Eigen::Matrix.

Details

ApIq_int_nmE() calls the C function gsl_sf_hyperg_1F1() from GSL via **RcppGSL**.

Value

All return a list via Rcpp::List of the following (as appropriate):

$ans Exact moment, from double or long double

$ansseq Series for the moment, from Eigen::Array

$errseq Series of errors, from Eigen::Array

$twosided Logical, from bool

$dimnished Logical, from bool

Functions

- p_A1B1_Ed(): pqfm_A1B1(), double
- p_A1B1_El(): pqfm_A1B1(), long double
- p_A1B1_Ec(): pqfm_A1B1(), coefficient-wise scaling
- d_A1I1_Ed(): dqfm_A1I1()
- p_imhof_Ed(): pqfm_imhof()
- d_broda_Ed(): dqfm_broda()
- d_butler_Ed(): dqfm_butler()
- p_butler_Ed(): pqfm_butler()
- Ap_int_E(): qfm_Ap_int()
- ABpq_int_E(): qfpm_ABpq_int()
- ABDpqr_int_E(): qfpm_ABDpqr_int()
- ApIq_int_cE(): qfrm_ApIq_int(), central
- ApIq_int_nE(): qfrm_ApIq_int(), noncentral
- ApIq_npi_cE(): qfrm_ApIq_npi(), central
- ApIq_npi_nEd(): qfrm_ApIq_npi(), noncentral, double
- ApBq_int_E(): qfrm_ApBq_int()
- ApBq_npi_Ed(): qfrm_ApBq_npi(), double
- ApBIqr_int_cEd(): qfmrm_ApBIqr_int(), central
- ApBIqr_int_nEd(): qfmrm_ApBIqr_int(), noncentral, double
- ApBIqr_npi_Ed(): qfmrm_ApBIqr_npi(), double
- IpBDqr_gen_Ed(): qfmrm_IpBDqr_gen(), double
- ApBDqr_int_Ed(): qfmrm_ApBDqr_int(), double
- ApBDqr_npi_Ed(): qfmrm_ApBDqr_npi(), double
- ApIq_npi_nEc(): qfrm_ApIq_npi(), noncentral, coefficient-wise scaling
- ApBq_npi_Ec(): qfrm_ApBq_npi(), coefficient-wise scaling
- ApBIqr_int_nEc(): qfmrm_ApBIqr_int(), noncentral, coefficient-wise scaling
- ApBIqr_npi_Ec(): qfmrm_ApBIqr_npi(), coefficient-wise scaling
- IpBDqr_gen_Ec(): qfmrm_IpBDqr_gen(), double
- ApBDqr_int_Ec(): qfmrm_ApBDqr_int(), coefficient-wise scaling
- ApBDqr_npi_Ec(): qfmrm_ApBDqr_npi(), coefficient-wise scaling
- ApIq_npi_nEl(): qfrm_ApIq_npi(), noncentral, long double

- `ApBq_npi_El()`: `qfrm_ApBq_npi()`, long double
- `ApBIqr_int_nEl()`: `qfmrm_ApBIqr_int()`, noncentral, long double
- `ApBIqr_npi_El()`: `qfmrm_ApBIqr_npi()`, long double
- `IpBDqr_gen_El()`: `qfmrm_IpBDqr_gen()`, long double
- `ApBDqr_int_El()`: `qfmrm_ApBDqr_int()`, long double
- `ApBDqr_npi_El()`: `qfmrm_ApBDqr_npi()`, long double
- `rqfpE()`: `rqfp()`

`qfmrm` *Moment of multiple ratio of quadratic forms in normal variables*

Description

`qfmrm()` is a front-end function to obtain the (compound) moment of a multiple ratio of quadratic forms in normal variables in the following special form: $\mathrm{E}\left(\frac{(\mathbf{x}^T\mathbf{A}\mathbf{x})^p}{(\mathbf{x}^T\mathbf{B}\mathbf{x})^q(\mathbf{x}^T\mathbf{D}\mathbf{x})^r}\right)$, where $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Like `qfrm()`, this function calls one of the following "internal" functions for actual calculation, as appropriate.

`qfmrm_ApBIqr_int()`: For $\mathbf{D} = \mathbf{I}_n$ and positive-integral p

`qfmrm_ApBIqr_npi()`: For $\mathbf{D} = \mathbf{I}_n$ and non-integral p

`qfmrm_IpBDqr_gen()`: For $\mathbf{A} = \mathbf{I}_n$

`qfmrm_ApBDqr_int()`: For general $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$, and positive-integral p

`qfmrm_ApBDqr_npi()`: For general $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$, and non-integral p

Usage

```
qfmrm(
  A,
  B,
  D,
  p = 1,
  q = p/2,
  r = q,
  m = 100L,
  mu = rep.int(0, n),
  Sigma = diag(n),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  ...
)

qfmrm_ApBIqr_int(
  A,
  B,
```

```
  p = 1,
  q = 1,
  r = 1,
  m = 100L,
  mu = rep.int(0, n),
  error_bound = TRUE,
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  cpp_method = c("double", "long_double", "coef_wise"),
  nthreads = 0,
  alphaB = 1,
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  thr_margin = 100
)

qfmrm_ApBIqr_npi(
  A,
  B,
  p = 1,
  q = 1,
  r = 1,
  m = 100L,
  mu = rep.int(0, n),
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  cpp_method = c("double", "long_double", "coef_wise"),
  nthreads = 0,
  alphaA = 1,
  alphaB = 1,
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  thr_margin = 100
)

qfmrm_IpBDqr_gen(
  B,
  D,
  p = 1,
  q = 1,
  r = 1,
  mu = rep.int(0, n),
  m = 100L,
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  cpp_method = c("double", "long_double", "coef_wise"),
```

```
  nthreads = 0,
  alphaB = 1,
  alphaD = 1,
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  thr_margin = 100
)

qfmrm_ApBDqr_int(
  A,
  B,
  D,
  p = 1,
  q = 1,
  r = 1,
  m = 100L,
  mu = rep.int(0, n),
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  cpp_method = c("double", "long_double", "coef_wise"),
  nthreads = 0,
  alphaB = 1,
  alphaD = 1,
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  thr_margin = 100
)

qfmrm_ApBDqr_npi(
  A,
  B,
  D,
  p = 1,
  q = 1,
  r = 1,
  m = 100L,
  mu = rep.int(0, n),
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  cpp_method = c("double", "long_double", "coef_wise"),
  nthreads = 0,
  alphaA = 1,
  alphaB = 1,
  alphaD = 1,
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
```

```
  tol_sing = tol_zero,
  thr_margin = 100
)
```

Arguments

A, B, D — Argument matrices. Should be square. Will be automatically symmetrized.

p, q, r — Exponents for $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$, respectively. By default, q equals p/2 and r equals q. If unsure, specify all explicitly.

m — Order of polynomials at which the series expression is truncated. M in Hillier et al. (2009, 2014).

mu — Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$

Sigma — Covariance matrix $\mathbf{\Sigma}$ for $\mathbf{x}$. Accommodated only by the front-end qfmrm(). See "Details" in qfrm.

tol_zero — Tolerance against which numerical zero is determined. Used to determine, e.g., whether mu is a zero vector, A or B equals the identity matrix, etc.

tol_sing — Tolerance against which matrix singularity and rank are determined. The eigenvalues smaller than this are considered zero.

... — Additional arguments in the front-end qfmrm() will be passed to the appropriate "internal" function.

error_bound — Logical to specify whether an error bound is returned (if available).

check_convergence — Specifies how numerical convergence is checked (see "Details"). Options:

- "relative" default; magnitude of the last term of the series relative to the sum is compared with tol_conv
- "strict_relative" **or** TRUE same, but stricter than default by setting tol_conv = .Machine$double.eps (unless a smaller value is specified by the user)
- "absolute" absolute magnitude of the last term is compared with tol_conv
- "none" **or** FALSE skips convergence check

use_cpp — Logical to specify whether the calculation is done with C++ functions via Rcpp. TRUE by default.

cpp_method — Method used in C++ calculations to avoid numerical overflow/underflow (see "Details"). Options:

- "double" default; fastest but prone to underflow in some conditions
- "long_double" same algorithm but using the long double variable type; robust but slow and memory-inefficient
- "coef_wise" coefficient-wise scaling algorithm; most robust but variably slow

nthreads — Number of threads used in OpenMP-enabled C++ functions. See "Multithreading" in qfrm.

tol_conv — Tolerance against which numerical convergence of series is checked. Used with check_convergence.

thr_margin — Optional argument to adjust the threshold for scaling (see "Scaling" in d1_i). Passed to internal functions (d1_i, d2_ij, d3_ijk) or their C++ equivalents.

alphaA, alphaB, alphaD — Factors for the scaling constants for $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$, respectively. See "Details" in qfrm.

Details

The usage of these functions is similar to qfrm, to which the user is referred for documentation. It is assumed that $\mathbf{B} \neq \mathbf{D}$ (otherwise, the problem reduces to a simple ratio).

When B is identity or missing, this and its exponent q will be swapped with D and r, respectively, before qfmrm_ApBIqr_***() is called.

The existence conditions for the moments of this multiple ratio can be reduced to those for a simple ratio, provided that one of the null spaces of $\mathbf{B}$ and $\mathbf{D}$ is a subspace of the other (including the case they are null). The conditions of Bao and Kan (2013: proposition 1) can then be applied by replacing q and m there by $q + r$ and $\min\left(\mathrm{rank}(\mathbf{B}), \mathrm{rank}(\mathbf{D})\right)$, respectively (see also Smith 1989: p. 258 for nonsingular $\mathbf{B}$, $\mathbf{D}$). An error is thrown if these conditions are not met in this case. Otherwise (i.e., $\mathbf{B}$ and $\mathbf{D}$ are both singular and neither of their null spaces is a subspace of the other), it seems difficult to define general moment existence conditions. A sufficient condition can be obtained by applying the same proposition with a new denominator matrix whose null space is union of those of $\mathbf{B}$ and $\mathbf{D}$ (Watanabe, 2023). A warning is thrown if that condition is not met in this case.

Most of these functions, excepting qfmrm_ApBIqr_int() with zero mu, involve evaluation of multiple series, which can suffer from numerical overflow and underflow (see “Scaling” in d1_i and “Details” in qfrm). To avoid this, cpp_method = "long_double" or "coef_wise" options can be used (see “Details” in qfrm).

Value

A qfrm object, as in qfrm() functions.

References

Bao, Y. and Kan, R. (2013) On the moments of ratios of quadratic forms in normal random variables. *Journal of Multivariate Analysis*, **117**, 229–245. doi:10.1016/j.jmva.2013.03.002.

Smith, M. D. (1989). On the expectation of a ratio of quadratic forms in normal variables. *Journal of Multivariate Analysis*, **31**, 244–257. doi:10.1016/0047259X(89)900651.

Watanabe, J. (2023) Exact expressions and numerical evaluation of average evolvability measures for characterizing and comparing $\mathbf{G}$ matrices. *Journal of Mathematical Biology*, **86**, 95. doi:10.1007/s00285023019308.

See Also

qfrm for simple ratio

Examples

```
## Some symmetric matrices and parameters
nv <- 4
A <- diag(nv:1)
B <- diag(sqrt(1:nv))
D <- diag((1:nv)^2 / nv)
mu <- nv:1 / nv
Sigma <- matrix(0.5, nv, nv)
diag(Sigma) <- 1
```

```
## Expectation of (x^T A x)^2 / (x^T B x) (x^T x) where x ~ N(0, I)
(res1 <- qfmrm(A, B, p = 2, q = 1, r = 1))
plot(res1)

# The above internally calls the following:
qfmrm_ApBIqr_int(A, B, p = 2, q = 1, r = 1) ## The same

# Similar result with different expression
# This is a suboptimal option and throws a warning
qfmrm_ApBIqr_npi(A, B, p = 2, q = 1, r = 1)

## Expectation of (x^T A x) / (x^T B x)^(1/2) (x^T D x)^(1/2) where x ~ N(0, I)
(res2 <- qfmrm(A, B, D, p = 1, q = 1/2, r = 1/2))
plot(res2)

# The above internally calls the following:
qfmrm_ApBDqr_int(A, B, D, p = 1, q = 1/2, r = 1/2) ## The same

## Average response correlation between A and B
(res3 <- qfmrm(crossprod(A, B), crossprod(A), crossprod(B),
               p = 1, q = 1/2, r = 1/2))
plot(res3)

## Same, but with x ~ N(mu, Sigma)
(res4 <- qfmrm(crossprod(A, B), crossprod(A), crossprod(B),
               p = 1, q = 1/2, r = 1/2, mu = mu, Sigma = Sigma))
plot(res4)

## Average autonomy of D
(res5 <- qfmrm(B = D, D = solve(D), p = 2, q = 1, r = 1))
plot(res5)
```

---

qfpm *Moment of (product of) quadratic forms in normal variables*

Description

Functions to obtain (compound) moments of a product of quadratic forms in normal variables, i.e., $\mathrm{E}\left((\mathbf{x}^T\mathbf{A}\mathbf{x})^p(\mathbf{x}^T\mathbf{B}\mathbf{x})^q(\mathbf{x}^T\mathbf{D}\mathbf{x})^r\right)$, where $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

`qfm_Ap_int()` is for $q = r = 0$ (simple moment)

`qfpm_ABpq_int()` is for $r = 0$

`qfpm_ABDpqr_int()` is for the product of all three powers

Usage

```
qfm_Ap_int(
  A,
```

```
  p = 1,
  mu = rep.int(0, n),
  Sigma = diag(n),
  use_cpp = TRUE,
  cpp_method = "double",
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero
)

qfpm_ABpq_int(
  A,
  B,
  p = 1,
  q = 1,
  mu = rep.int(0, n),
  Sigma = diag(n),
  use_cpp = TRUE,
  cpp_method = "double",
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero
)

qfpm_ABDpqr_int(
  A,
  B,
  D,
  p = 1,
  q = 1,
  r = 1,
  mu = rep.int(0, n),
  Sigma = diag(n),
  use_cpp = TRUE,
  cpp_method = "double",
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero
)
```

Arguments

A, B, D	Argument matrices. Should be square. Will be automatically symmetrized.
p, q, r	Exponents for $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$, respectively. By default, these are set to the same value. If unsure, specify all explicitly.
mu	Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$
Sigma	Covariance matrix $\boldsymbol{\Sigma}$ for $\mathbf{x}$
use_cpp	Logical to specify whether the calculation is done with C++ functions via Rcpp. TRUE by default.
cpp_method	Variable type used in C++ calculations. In these functions this is ignored.

tol_zero Tolerance against which numerical zero is determined. Used to determine, e.g., whether mu is a zero vector, A or B equals the identity matrix, etc.

tol_sing Tolerance against which matrix singularity and rank are determined. The eigenvalues smaller than this are considered zero.

Details

These functions implement the super-short recursion algorithms described in Hillier et al. (2014: sec. 3.1–3.2 and 4). At present, only positive integers are accepted as exponents (negative exponents yield ratios, of course). All these yield exact results.

Value

A qfpm object which has the same elements as those returned by the qfrm functions. Use $statistic to access the value of the moment.

See Also

qfrm and qfmrm for moments of ratios

Examples

```
## Some symmetric matrices and parameters
nv <- 4
A <- diag(nv:1)
B <- diag(sqrt(1:nv))
D <- diag((1:nv)^2 / nv)
mu <- nv:1 / nv
Sigma <- matrix(0.5, nv, nv)
diag(Sigma) <- 1

## Expectation of (x^T A x)^2 where x ~ N(0, I)
qfm_Ap_int(A, 2)

## This is the same but obviously less efficient
qfpm_ABpq_int(A, p = 2, q = 0)

## Expectation of (x^T A x) (x^T B x) (x^T D x) where x ~ N(0, I)
qfpm_ABDpqr_int(A, B, D, 1, 1, 1)

## Expectation of (x^T A x) (x^T B x) (x^T D x) where x ~ N(mu, Sigma)
qfpm_ABDpqr_int(A, B, D, 1, 1, 1, mu = mu, Sigma = Sigma)

## Expectations of (x^T x)^2 where x ~ N(0, I) and x ~ N(mu, I)
## i.e., roundabout way to obtain moments of
## central and noncentral chi-square variables
qfm_Ap_int(diag(nv), 2)
qfm_Ap_int(diag(nv), 2, mu = mu)
```

qfrm *Moment of ratio of quadratic forms in normal variables*

Description

qfrm() is a front-end function to obtain the (compound) moment of a ratio of quadratic forms in normal variables, i.e., $\mathrm{E}\left(\frac{(\mathbf{x}^T\mathbf{A}\mathbf{x})^p}{(\mathbf{x}^T\mathbf{B}\mathbf{x})^q}\right)$, where $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Internally, qfrm() calls one of the following functions which does the actual calculation, depending on $\mathbf{A}$, $\mathbf{B}$, and p. Usually the best one is automatically selected.

qfrm_ApIq_int(): For $\mathbf{B} = \mathbf{I}_n$ and positive-integral p.

qfrm_ApIq_npi(): For $\mathbf{B} = \mathbf{I}_n$ and non-positive-integral p (fraction or negative).

qfrm_ApBq_int(): For general $\mathbf{B}$ and positive-integral p.

qfrm_ApBq_npi(): For general $\mathbf{B}$ and non-integral p.

Usage

```
qfrm(
  A,
  B,
  p = 1,
  q = p,
  m = 100L,
  mu = rep.int(0, n),
  Sigma = diag(n),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  ...
)

qfrm_ApIq_int(
  A,
  p = 1,
  q = p,
  m = 100L,
  mu = rep.int(0, n),
  use_cpp = TRUE,
  exact_method = TRUE,
  cpp_method = "double",
  nthreads = 1,
  tol_zero = .Machine$double.eps * 100,
  thr_margin = 100
)

qfrm_ApIq_npi(
  A,
```

```
  p = 1,
  q = p,
  m = 100L,
  mu = rep.int(0, n),
  error_bound = TRUE,
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  cpp_method = c("double", "long_double", "coef_wise"),
  nthreads = 1,
  alphaA = 1,
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  thr_margin = 100
)

qfrm_ApBq_int(
  A,
  B,
  p = 1,
  q = p,
  m = 100L,
  mu = rep.int(0, n),
  error_bound = TRUE,
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  cpp_method = "double",
  nthreads = 1,
  alphaB = 1,
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  thr_margin = 100
)

qfrm_ApBq_npi(
  A,
  B,
  p = 1,
  q = p,
  m = 100L,
  mu = rep.int(0, n),
  check_convergence = c("relative", "strict_relative", "absolute", "none"),
  use_cpp = TRUE,
  cpp_method = c("double", "long_double", "coef_wise"),
  nthreads = 0,
  alphaA = 1,
  alphaB = 1,
```

```
  tol_conv = .Machine$double.eps^(1/4),
  tol_zero = .Machine$double.eps * 100,
  tol_sing = tol_zero,
  thr_margin = 100
)
```

Arguments

`A, B` Argument matrices. Should be square. Will be automatically symmetrized.

`p, q` Exponents corresponding to $\mathbf{A}$ and $\mathbf{B}$, respectively. When only one is provided, the other is set to the same value. Should be length-one numeric (see "Details" for further conditions).

`m` Order of polynomials at which the series expression is truncated. M in Hillier et al. (2009, 2014).

`mu` Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$

`Sigma` Covariance matrix $\mathbf{\Sigma}$ for $\mathbf{x}$. Accommodated only by the front-end `qfrm()`. See "Details".

`tol_zero` Tolerance against which numerical zero is determined. Used to determine, e.g., whether `mu` is a zero vector, `A` or `B` equals the identity matrix, etc.

`tol_sing` Tolerance against which matrix singularity and rank are determined. The eigenvalues smaller than this are considered zero.

`...` Additional arguments in the front-end `qfrm()` will be passed to the appropriate "internal" function.

`use_cpp` Logical to specify whether the calculation is done with C++ functions via `Rcpp`. `TRUE` by default.

`exact_method` Logical to specify whether the exact method is used in `qfrm_ApIq_int()` (see "Details").

`cpp_method` Method used in C++ calculations to avoid numerical overflow/underflow (see "Details"). Options:

- `"double"` default; fastest but prone to underflow in some conditions
- `"long_double"` same algorithm but using the `long double` variable type; robust but slow and memory-inefficient
- `"coef_wise"` coefficient-wise scaling algorithm; most robust but variably slow

`nthreads` Number of threads used in OpenMP-enabled C++ functions. `0` or any negative value is special and means one-half of the number of processors detected. See "Multithreading" in "Details".

`thr_margin` Optional argument to adjust the threshold for scaling (see "Scaling" in `d1_i`). Passed to internal functions (`d1_i`, `d2_ij`, `d3_ijk`) or their C++ equivalents.

`error_bound` Logical to specify whether an error bound is returned (if available).

`check_convergence` Specifies how numerical convergence is checked (see "Details"). Options:

- `"relative"` default; magnitude of the last term of the series relative to the sum is compared with `tol_conv` "strict_relative" **or** TRUE same, but stricter than default by setting tol_conv = .Machine$double.eps (unless a smaller value is specified by the user)

"absolute" absolute magnitude of the last term is compared with tol_conv

"none" **or** FALSE skips convergence check

alphaA, alphaB Factors for the scaling constants for $\mathbf{A}$ and $\mathbf{B}$, respectively. See "Details".

tol_conv Tolerance against which numerical convergence of series is checked. Used with check_convergence.

Details

These functions use infinite series expressions based on the joint moment-generating function (with the top-order zonal/invariant polynomials) (see Smith 1989, Hillier et al. 2009, 2014; Bao and Kan 2013), and the results are typically partial (truncated) sums from these infinite series, which necessarily involve truncation errors. (An exception is when $\mathbf{B} = \mathbf{I}_n$ and p is a positive integer, the case handled by qfrm_ApIq_int().)

The returned value is a list consisting of the truncated sequence up to the order specified by m, its sum, and error bounds corresponding to these (see "Values"). The print method only displays the terminal partial sum and its error bound (when available). Use plot() for visual inspection, or the ordinary list element access as required.

In most cases, p and q must be nonnegative (in addition, p must be an integer in qfrm_ApIq_int() and qfrm_ApBq_int() when used directly), and an error is thrown otherwise. The only exception is qfrm_ApIq_npi() which accepts negative exponents to accommodate $\frac{(\mathbf{x}^T\mathbf{x})^q}{(\mathbf{x}^T\mathbf{A}\mathbf{x})^p}$. Even in the latter case, the exponents must have the same sign. (Technically, not all of these conditions are necessary for the mathematical results to hold, but they are enforced for simplicity).

When error_bound = TRUE (default), qfrm_ApBq_int() evaluates a truncation error bound following Hillier et al. (2009: theorem 6) or Hillier et al. (2014: theorem 7) (for zero and nonzero means, respectively). qfrm_ApIq_npi() implements similar error bounds. No error bound is known for qfrm_ApBq_npi() to the author's knowledge.

For situations when the error bound is unavailable, a *very rough* check of numerical convergence is also conducted; a warning is thrown if the magnitude of the last term does not look small enough. By default, its relative magnitude to the sum is compared with the tolerance controlled by tol_conv, whose default is .Machine$double.eps^(1/4) (= ~1.2e-04) (see check_convergence).

When Sigma is provided, the quadratic forms are transformed into a canonical form; that is, using the decomposition $\mathbf{\Sigma} = \mathbf{K}\mathbf{K}^T$, where the number of columns m of $\mathbf{K}$ equals the rank of $\mathbf{\Sigma}$, $\mathbf{A}_{\text{new}} = \mathbf{K}^T\mathbf{A}\mathbf{K}$, $\mathbf{B}_{\text{new}} = \mathbf{K}^T\mathbf{B}\mathbf{K}$, and $\mathbf{x}_{\text{new}} = \mathbf{K}^{-}\mathbf{x} \sim N_m(\mathbf{K}^{-}\boldsymbol{\mu}, \mathbf{I}_m)$. qfrm() handles this by transforming A, B, and mu and calling itself recursively with these new arguments. Note that the "internal" functions do not accommodate Sigma (the error for unused arguments will happen). For singular $\mathbf{\Sigma}$, one of the following conditions must be met for the above transformation to be valid: **1**) $\boldsymbol{\mu}$ is in the range of $\mathbf{\Sigma}$; **2**) $\mathbf{A}$ and $\mathbf{B}$ are in the range of $\mathbf{\Sigma}$; or **3**) $\mathbf{A}\boldsymbol{\mu} = \mathbf{B}\boldsymbol{\mu} = \mathbf{0}_n$. An error is thrown if none is met with a singular Sigma.

The existence of the moment is assessed by the eigenstructures of $\mathbf{A}$ and $\mathbf{B}$, p, and q, according to Bao and Kan (2013: proposition 1). An error will result if the conditions are not met.

Straightforward implementation of the original recursive algorithms can suffer from numerical overflow when the problem is large. Internal functions (d1_i, d2_ij, d3_ijk) are designed to avoid overflow by order-wise scaling. However, when evaluation of multiple series is required (`qfrm_ApIq_npi()` with nonzero `mu` and `qfrm_ApBq_npi()`), the scaling occasionally yields underflow/diminishing of some terms to numerical `0`, causing inaccuracy. A warning is thrown in this case. (See also "Scaling" in `d1_i`.) To avoid this problem, the C++ versions of these functions have two workarounds, as controlled by `cpp_method`. **1**) The `"long_double"` option uses the `long double` variable type instead of the regular `double`. This is generally slow and most memory-inefficient. **2**) The `"coef_wise"` option uses a coefficient-wise scaling algorithm with the `double` variable type. This is generally robust against underflow issues. Computational time varies a lot with conditions; generally only modestly slower than the `"double"` option, but can be the slowest in some extreme conditions.

For the sake of completeness (only), the scaling parameters β (see the package vignette) can be modified via the arguments `alphaA` and `alphaB`. These are the factors for the inverses of the largest eigenvalues of $\mathbf{A}$ and $\mathbf{B}$, respectively, and must be between 0 and 2. The default is 1, which should suffice for most purposes. Values larger than 1 often yield faster convergence, but are *not* recommended as the error bound will not strictly hold (see Hillier et al. 2009, 2014).

Multithreading:

All these functions use C++ versions to speed up computation by default. Furthermore, some of the C++ functions, in particular those using more than one matrix arguments, are parallelized with OpenMP (when available). Use the argument `nthreads` to control the number of OpenMP threads. By default (`nthreads = 0`), one-half of the processors detected with `omp_get_num_procs()` are used. This is except when all the argument matrices share the same eigenvectors and hence the calculation only involves element-wise operations of eigenvalues. In that case, the calculation is typically fast without parallelization, so `nthreads` is automatically set to `1` unless explicitly specified otherwise; the user can still specify a larger value or `0` for (typically marginal) speed gains in large problems.

Exact method for `qfrm_ApIq_int()`:

An exact expression of the moment is available when p is integer and $\mathbf{B} = \mathbf{I}_n$ (handled by `qfrm_ApIq_int()`), whose expression involves a confluent hypergeometric function when $\boldsymbol{\mu}$ is nonzero (Hillier et al. 2014: theorem 4). There is an option (`exact_method = FALSE`) to use the ordinary infinite series expression (Hillier et al. 2009), which is less accurate and slow.

Value

A `qfrm` object consisting of the following:

`$statistic` evaluation result (`sum(terms)`)

`$terms` vector of 0th to mth order terms

`$error_bound` error bound of `statistic`

`$seq_error` vector of error bounds corresponding to partial sums (`cumsum(terms)`)

References

Bao, Y. and Kan, R. (2013) On the moments of ratios of quadratic forms in normal random variables. *Journal of Multivariate Analysis*, **117**, 229–245. doi:10.1016/j.jmva.2013.03.002.

Hillier, G., Kan, R. and Wang, X. (2009) Computationally efficient recursions for top-order invariant polynomials with applications. *Econometric Theory*, **25**, 211–242. doi:10.1017/S0266466608090075.

Hillier, G., Kan, R. and Wang, X. (2014) Generating functions and short recursions, with applications to the moments of quadratic forms in noncentral normal vectors. *Econometric Theory*, **30**, 436–473. doi:10.1017/S0266466613000364.

Smith, M. D. (1989) On the expectation of a ratio of quadratic forms in normal variables. *Journal of Multivariate Analysis*, **31**, 244–257. doi:10.1016/0047259X(89)900651.

Smith, M. D. (1993) Expectations of ratios of quadratic forms in normal variables: evaluating some top-order invariant polynomials. *Australian Journal of Statistics*, **35**, 271–282. doi:10.1111/j.1467-842X.1993.tb01335.x.

See Also

`qfmrm` for multiple ratio

Examples

```
## Some symmetric matrices and parameters
nv <- 4
A <- diag(nv:1)
B <- diag(sqrt(1:nv))
mu <- nv:1 / nv
Sigma <- matrix(0.5, nv, nv)
diag(Sigma) <- 1

## Expectation of (x^T A x)^2 / (x^T x)^2 where x ~ N(0, I)
## An exact expression is available
(res1 <- qfrm(A, p = 2))

# The above internally calls the following:
qfrm_ApIq_int(A, p = 2) ## The same

# Similar result with different expression
# This is a suboptimal option and throws a warning
qfrm_ApIq_npi(A, p = 2)

## Expectation of (x^T A x)^1/2 / (x^T x)^1/2 where x ~ N(0, I)
## Note how quickly the series converges in this case
(res2 <- qfrm(A, p = 1/2))
plot(res2)

# The above calls:
qfrm_ApIq_npi(A, p = 0.5)

# This is not allowed (throws an error):
try(qfrm_ApIq_int(A, p = 0.5))

## (x^T A x)^2 / (x^T B x)^3 where x ~ N(0, I)
(res3 <- qfrm(A, B, 2, 3))
plot(res3)

## (x^T A x)^2 / (x^T B x)^2 where x ~ N(mu, I)
## Note the two-sided error bound
```

```
(res4 <- qfrm(A, B, 2, 2, mu = mu))
plot(res4)

## (x^T A x)^2 / (x^T B x)^2 where x ~ N(mu, Sigma)
(res5 <- qfrm(A, B, p = 2, q = 2, mu = mu, Sigma = Sigma))
plot(res5)

# Sigma is not allowed in the "internal" functions:
try(qfrm_ApBq_int(A, B, p = 2, q = 2, Sigma = Sigma))

# In res5 above, the error bound didn't converge
# Use larger m to evaluate higher-order terms
plot(print(qfrm(A, B, p = 2, q = 2, mu = mu, Sigma = Sigma, m = 300)))
```

---

range_qfr *Get range of ratio of quadratic forms*

Description

`range_qfr()`: internal function to obtain the possible range of a ratio of quadratic forms, $\frac{\mathbf{x}^T \mathbf{A} \mathbf{x}}{\mathbf{x}^T \mathbf{B} \mathbf{x}}$.

`gen_eig()` is an internal function to obtain generalized eigenvalues, i.e., roots of $\det \mathbf{A} - \lambda \mathbf{B} = 0$, which are the eigenvalues of $\mathbf{B}^{-1}\mathbf{A}$ if $\mathbf{B}$ is nonsingular.

Usage

```
range_qfr(
  A,
  B,
  eigB = eigen(B, symmetric = TRUE),
  tol = .Machine$double.eps * 100,
  t = 0.001
)

gen_eig(
  A,
  B,
  eigB = eigen(B, symmetric = TRUE),
  Ad = with(eigB, crossprod(crossprod(A, vectors), vectors)),
  tol = .Machine$double.eps * 100,
  t = 0.001
)
```

Arguments

`A, B`	Symmetric matrices. No check is done.
`eigB`	Result of `eigen(B)` can be passed when already computed

`tol`	Tolerance to determine numerical zero
`t`	Tolerance used to determine whether estimates are numerically stable; t in Jennings et al. (1978).
`Ad`	`A` rotated with eigenvectors of `B` can be passed when already computed

Details

`gen_eig()` solves the generalized eigenvalue problem with Jennings et al.'s (1978) algorithm. The sign of infinite eigenvalue (when present) cannot be determined from this algorithm, so is deduced as follows: (1) $\mathbf{A}$ and $\mathbf{B}$ are rotated by the eigenvectors of $\mathbf{B}$; (2) the submatrix of rotated $\mathbf{A}$ corresponding to the null space of $\mathbf{B}$ is examined; (3) if this is nonnegative (nonpositive) definite, the result must have positive (negative, resp.) infinity; if this is indefinite, the result must have both positive and negative infinities; if this is (numerically) zero, the result must have `NaN`. The last case is expeted to happen very rarely, as in this case Jennings algorithm would fail. This is where the null space of $\mathbf{B}$ is a subspace of that of $\mathbf{A}$, so that the range of ratio of quadratic forms can be well-behaved. `range_qfr()` tries to detect this case and handle the range accordingly, but if that is infeasible it returns `c(-Inf, Inf)`.

References

Jennings, A., Halliday, J. and Cole, M. J. (1978) Solution of linear generalized eigenvalue problems containing singular matrices. *Journal of the Institute of Mathematics and Its Applications*, **22**, 401–410. doi:10.1093/imamat/22.4.401.

`rqfr` *Monte Carlo sampling of ratio/product of quadratic forms*

Description

`rqfr()`, `rqfmr()`, and `rqfp()` calculate a random sample of a simple ratio, multiple ratio (of special form), and product, respectively, of quadratic forms in normal variables of specified mean and covariance (standard multivariate normal by default). These functions are primarily for empirical verification of the analytic results provided in this package.

Usage

```
rqfr(nit = 1000L, A, B, p = 1, q = p, mu, Sigma, use_cpp = TRUE)

rqfmr(nit = 1000L, A, B, D, p = 1, q = p/2, r = q, mu, Sigma, use_cpp = TRUE)

rqfp(nit = 1000L, A, B, D, p = 1, q = 1, r = 1, mu, Sigma, use_cpp = TRUE)
```

Arguments

`nit`	Number of iteration or sample size. Should be an integer-alike of length 1.
`A, B, D`	Argument matrices (see "Details"). Assumed to be square matrices of the same order. When missing, set to the identity matrix. At least one of these must be specified.
`p, q, r`	Exponents for A, B, D, respectively (see "Details"). Assumed to be numeric of length 1 each. See "Details" for default values.
`mu`	Mean vector $\boldsymbol{\mu}$ for $\mathbf{x}$. Default zero vector.
`Sigma`	Covariance matrix $\boldsymbol{\Sigma}$ for $\mathbf{x}$. Default identity matrix. `mu` and `Sigma` are assumed to be of the same order as the argument matrices.
`use_cpp`	Logical to specify whether an C++ version is called or not. `TRUE` by default.

Details

These functions generate a random sample of $\frac{(\mathbf{x}^T\mathbf{A}\mathbf{x})^p}{(\mathbf{x}^T\mathbf{B}\mathbf{x})^q}$ (`rqfr()`), $\frac{(\mathbf{x}^T\mathbf{A}\mathbf{x})^p}{(\mathbf{x}^T\mathbf{B}\mathbf{x})^q(\mathbf{x}^T\mathbf{D}\mathbf{x})^r}$ (`rqfmr()`), and $(\mathbf{x}^T\mathbf{A}\mathbf{x})^p(\mathbf{x}^T\mathbf{B}\mathbf{x})^q(\mathbf{x}^T\mathbf{D}\mathbf{x})^r$ (`rqfp()`), where $\mathbf{x} \sim N_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. (Internally, `rqfr()` and `rqfmr()` just call `rqfp()` with negative exponents.)

When only one of `p` and `q` is provided in `rqfr()`, the other (missing) one is set to the same value.

In `rqfmr()`, `q` and `r` are set to `p/2` when both missing, and set to the same value when only one is missing. When `p` is missing, this is set to be `q + r`. If unsure, specify all these explicitly.

In `rqfp()`, `p`, `q` and `r` are `1` by default, provided that the corresponding argument matrices are given. If both an argument matrix and its exponent (e.g., `D` and `r`) are missing, the exponent is set to `0` so that the factor be unity.

Value

Numeric vector of length `nit`.

See Also

`qfrm` and `qfpm` for analytic moments

`dqfr` for analytic distribution-related functions for simple ratios

Examples

```
p <- 4
A <- diag(1:p)
B <- diag(p:1)
D <- diag(sqrt(1:p))

## By default B = I, p = q = 1;
## i.e., (x^T A x) / (x^T x), x ~ N(0, I)
rqfr(5, A)

## (x^T A x) / ((x^T B x)(x^T D x))^(1/2), x ~ N(0, I)
rqfmr(5, A, B, D, 1, 1/2, 1/2)
```

```
## (x^T A x), x ~ N(0, I)
rqfp(5, A)

## (x^T A x) (x^T B x), x ~ N(0, I)
rqfp(5, A, B)

## (x^T A x) (x^T B x) (x^T D x), x ~ N(0, I)
rqfp(5, A, B, D)

## Example with non-standard normal
mu <- 1:p / p
Sigma <- matrix(0.5, p, p)
diag(Sigma) <- 1
rqfr(5, A, mu = 1:p / p, Sigma = Sigma)

## Compare Monte Carlo sample and analytic expression
set.seed(3)
mcres <- rqfr(1000, A, p = 2)
mean(mcres)
(anres <- qfrm(A, p = 2))
stats::t.test(mcres, mu = anres$statistic)
```

---

sum_counterdiag *Summing up counter-diagonal elements*

Description

`sum_counterdiag()` sums up counter-diagonal elements of a square matrix from the upper-left; i.e., `c(X[1, 1], X[1, 2] + X[2, 1], X[1, 3] + X[2, 2] + X[3, 1], ...)` (or a sequence of $\sum_{i=1}^{k} x_{i,k-i+1}$ for $k = 1, 2, ...$). `sum_counterdiag3D()` does a comparable in a 3D cubic array. No check is done on the structure of `X`.

Usage

```
sum_counterdiag(X)

sum_counterdiag3D(X)
```

Arguments

`X`	Square matrix or cubic array

S_fromUL — *Make covariance matrix from eigenstructure*

Description

This is an internal utility function to make covariance matrix from eigenvectors and eigenvalues. Symmetry is assumed for the original matrix.

Usage

```
S_fromUL(evec, evalues)
```

Arguments

`evec`	Matrix whose columns are eigenvectors
`evalues`	Vector of eigenvalues

tr — *Matrix trace function*

Description

This is an internal function. No check is done on the structure of `X`.

Usage

```
tr(X)
```

Arguments

`X`	Square matrix whose trace is to be calculated

Index