

Package: mvnfast (via r-universe)

May 19, 2026

Type Package

Title Fast Multivariate Normal and Student's t Methods

Version 0.2.9

Date 2026-03-20

Maintainer Matteo Fasiolo <matteo.fasiolo@gmail.com>

Description Provides computationally efficient tools related to the multivariate normal and Student's t distributions. The main functionalities are: simulating multivariate random vectors, evaluating multivariate normal or Student's t densities and Mahalanobis distances. These tools are very efficient thanks to the use of C++ code and of the OpenMP API.

License GPL (>= 2.0)

URL <https://github.com/mfasiolo/mvnfast/>

Imports Rcpp

Suggests knitr, rmarkdown, testthat, mvtnorm, microbenchmark, MASS, plyr, RhpcBLASctl

LinkingTo Rcpp, RcppArmadillo, BH

VignetteBuilder knitr

RoxygenNote 7.3.3

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

Date/Publication 2026-03-20 11:28:40 UTC

RemoteUrl <https://github.com/mfasiolo/mvnfast>

RemoteRef HEAD

RemoteSha b914565cfbdfbecc91fa8d86c8601b6d1fbe583

Contents

dmixn	2
dmixt	4
dmvn	5

dmvt	7
maha	8
ms	9
rmixn	11
rmixt	13
rmvn	15
rmvt	17

Index	20
--------------	-----------

dmixn	<i>Fast density computation for mixture of multivariate normal distributions.</i>
-------	---

Description

Fast density computation for mixture of multivariate normal distributions.

Usage

```
dmixn(X, mu, sigma, w, log = FALSE, ncores = 1, isChol = FALSE, A = NULL)
```

Arguments

X	matrix n by d where each row is a d dimensional random vector. Alternatively X can be a d-dimensional vector.
mu	an (m x d) matrix, where m is the number of mixture components.
sigma	as list of m covariance matrices (d x d) on for each mixture component. Alternatively it can be a list of m cholesky decomposition of the covariance. In that case isChol should be set to TRUE.
w	vector of length m, containing the weights of the mixture components.
log	boolean set to true the logarithm of the pdf is required.
ncores	Number of cores used. The parallelization will take place only if OpenMP is supported.
isChol	boolean set to true is sigma is the cholesky decomposition of the covariance matrix.
A	an (optional) numeric matrix of dimension (m x d), which will be used to store the evaluations of each mixture density over each mixture component. It is useful when m and n are large and one wants to call dmixt() several times, without reallocating memory for the whole matrix each time. NB1: A will be modified, not copied! NB2: the element of A must be of class "numeric".

Details

NB: at the moment the parallelization does not work properly on Solaris OS when ncores>1. Hence, dmixt() checks if the OS is Solaris and, if this the case, it imposes ncores==1.

Value

A vector of length n where the i -th entry contains the pdf of the i -th random vector (i.e. the i -th row of X).

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>.

Examples

```
#### 1) Example use
# Set up mixture density
mu <- matrix(c(1, 2, 10, 20), 2, 2, byrow = TRUE)
sigma <- list(diag(c(1, 10)), matrix(c(1, -0.9, -0.9, 1), 2, 2))
w <- c(0.1, 0.9)

# Simulate
X <- rmixn(1e4, mu, sigma, w)

# Evaluate density
ds <- dmixn(X, mu, sigma, w = w)
head(ds)

##### 2) More complicated example
# Define mixture
set.seed(5135)
N <- 10000
d <- 2
w <- rep(1, 2) / 2
mu <- matrix(c(0, 0, 2, 3), 2, 2, byrow = TRUE)
sigma <- list(matrix(c(1, 0, 0, 2), 2, 2), matrix(c(1, -0.9, -0.9, 1), 2, 2))

# Simulate random variables
X <- rmixn(N, mu, sigma, w = w, retInd = TRUE)

# Plot mixture density
np <- 100
xvals <- seq(min(X[, 1]), max(X[, 1]), length.out = np)
yvals <- seq(min(X[, 2]), max(X[, 2]), length.out = np)
theGrid <- expand.grid(xvals, yvals)
theGrid <- as.matrix(theGrid)
dens <- dmixn(theGrid, mu, sigma, w = w)
plot(X, pch = '.', col = attr(X, "index")+1)
contour(x = xvals, y = yvals, z = matrix(dens, np, np),
        levels = c(0.002, 0.01, 0.02, 0.04, 0.08, 0.15 ), add = TRUE, lwd = 2)
```

dmixt	<i>Fast density computation for mixture of multivariate Student's t distributions.</i>
-------	--

Description

Fast density computation for mixture of multivariate Student's t distributions.

Usage

```
dmixt(X, mu, sigma, df, w, log = FALSE, ncores = 1, isChol = FALSE, A = NULL)
```

Arguments

X	matrix n by d where each row is a d dimensional random vector. Alternatively X can be a d-dimensional vector.
mu	an (m x d) matrix, where m is the number of mixture components.
sigma	as list of m covariance matrices (d x d) on for each mixture component. Alternatively it can be a list of m cholesky decomposition of the covariance. In that case isChol should be set to TRUE.
df	a positive scalar representing the degrees of freedom. All the densities in the mixture have the same df.
w	vector of length m, containing the weights of the mixture components.
log	boolean set to true the logarithm of the pdf is required.
ncores	Number of cores used. The parallelization will take place only if OpenMP is supported.
isChol	boolean set to true is sigma is the cholesky decomposition of the covariance matrix.
A	an (optional) numeric matrix of dimension (m x d), which will be used to store the evaluations of each mixture density over each mixture component. It is useful when m and n are large and one wants to call dmixt() several times, without reallocating memory for the whole matrix each time. NB1: A will be modified, not copied! NB2: the element of A must be of class "numeric".

Details

There are many candidates for the multivariate generalization of Student's t-distribution, here we use the parametrization described here https://en.wikipedia.org/wiki/Multivariate_t-distribution. NB: at the moment the parallelization does not work properly on Solaris OS when ncores>1. Hence, dmixt() checks if the OS is Solaris and, if this the case, it imposes ncores==1.

Value

A vector of length n where the i-th entry contains the pdf of the i-th random vector (i.e. the i-th row of X).

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>.

Examples

```
#### 1) Example use
# Set up mixture density
df <- 6
mu <- matrix(c(1, 2, 10, 20), 2, 2, byrow = TRUE)
sigma <- list(diag(c(1, 10)), matrix(c(1, -0.9, -0.9, 1), 2, 2))
w <- c(0.1, 0.9)

# Simulate
X <- rmixt(1e4, mu, sigma, df, w)

# Evaluate density
ds <- dmixt(X, mu, sigma, w = w, df = df)
head(ds)

##### 2) More complicated example
# Define mixture
set.seed(5135)
N <- 10000
d <- 2
df = 10
w <- rep(1, 2) / 2
mu <- matrix(c(0, 0, 2, 3), 2, 2, byrow = TRUE)
sigma <- list(matrix(c(1, 0, 0, 2), 2, 2), matrix(c(1, -0.9, -0.9, 1), 2, 2))

# Simulate random variables
X <- rmixt(N, mu, sigma, w = w, df = df, retInd = TRUE)

# Plot mixture density
np <- 100
xvals <- seq(min(X[, 1]), max(X[, 1]), length.out = np)
yvals <- seq(min(X[, 2]), max(X[, 2]), length.out = np)
theGrid <- expand.grid(xvals, yvals)
theGrid <- as.matrix(theGrid)
dens <- dmixt(theGrid, mu, sigma, w = w, df = df)
plot(X, pch = '.', col = attr(X, "index")+1)
contour(x = xvals, y = yvals, z = matrix(dens, np, np),
        levels = c(0.002, 0.01, 0.02, 0.04, 0.08, 0.15 ), add = TRUE, lwd = 2)
```

Description

Fast computation of the multivariate normal density.

Usage

```
dmvn(X, mu, sigma, log = FALSE, ncores = 1, isChol = FALSE)
```

Arguments

X	matrix n by d where each row is a d dimensional random vector. Alternatively X can be a d-dimensional vector.
mu	vector of length d, representing the mean of the distribution.
sigma	covariance matrix (d x d). Alternatively it can be the cholesky decomposition of the covariance. In that case isChol should be set to TRUE.
log	boolean set to true the logarithm of the pdf is required.
ncores	Number of cores used. The parallelization will take place only if OpenMP is supported.
isChol	boolean set to true is sigma is the cholesky decomposition of the covariance matrix.

Value

A vector of length n where the i-th entry contains the pdf of the i-th random vector.

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>

Examples

```
N <- 100
d <- 5
mu <- 1:d
X <- t(t(matrix(rnorm(N*d), N, d)) + mu)
tmp <- matrix(rnorm(d^2), d, d)
mcov <- tcrossprod(tmp, tmp) + diag(0.5, d)
myChol <- chol(mcov)

head(dmvn(X, mu, mcov), 10)
head(dmvn(X, mu, myChol, isChol = TRUE), 10)

## Not run:
# Performance comparison: microbenchmark does not work on all
# platforms, hence we need to check whether it is installed
if( "microbenchmark" %in% rownames(installed.packages()) ){
  library(mvtnorm)
  library(microbenchmark)

  a <- cbind(
    dmvn(X, mu, mcov),
    dmvn(X, mu, myChol, isChol = TRUE),
    dmvnorm(X, mu, mcov))
```

```

# Check if we get the same output as dmvnorm()
a[ , 1] / a[ , 3]
a[ , 2] / a[ , 3]

microbenchmark(dmvn(X, mu, myChol, isChol = TRUE),
               dmvn(X, mu, mcov),
               dmvnorm(X, mu, mcov))

detach("package:mvtnorm", unload=TRUE)
}

## End(Not run)

```

dmvt

Fast computation of the multivariate Student's t density.

Description

Fast computation of the multivariate Student's t density.

Usage

```
dmvt(X, mu, sigma, df, log = FALSE, ncores = 1, isChol = FALSE)
```

Arguments

X	matrix n by d where each row is a d dimensional random vector. Alternatively X can be a d-dimensional vector.
mu	vector of length d, representing the mean of the distribution.
sigma	scale matrix (d x d). Alternatively it can be the cholesky decomposition of the scale matrix. In that case isChol should be set to TRUE. Notice that ff the degrees of freedom (the argument df) is larger than 2, the $\text{Cov}(X)=\text{sigma}\cdot\text{df}/(\text{df}-2)$.
df	a positive scalar representing the degrees of freedom.
log	boolean set to true the logarithm of the pdf is required.
ncores	Number of cores used. The parallelization will take place only if OpenMP is supported.
isChol	boolean set to true is sigma is the cholesky decomposition of the covariance matrix.

Details

There are many candidates for the multivariate generalization of Student's t-distribution, here we use the parametrization described here https://en.wikipedia.org/wiki/Multivariate_t-distribution. NB: at the moment the parallelization does not work properly on Solaris OS when ncores>1. Hence, dmvt() checks if the OS is Solaris and, if this the case, it imposes ncores==1.

Value

A vector of length n where the i -th entry contains the pdf of the i -th random vector.

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>

Examples

```
N <- 100
d <- 5
mu <- 1:d
df <- 4
X <- t(t(matrix(rnorm(N*d), N, d)) + mu)
tmp <- matrix(rnorm(d^2), d, d)
mcov <- tcrossprod(tmp, tmp) + diag(0.5, d)
myChol <- chol(mcov)

head(dmvt(X, mu, mcov, df = df), 10)
head(dmvt(X, mu, myChol, df = df, isChol = TRUE), 10)
```

maha

Fast computation of squared mahalanobis distance between all rows of X and the vector mu with respect to sigma.

Description

Fast computation of squared mahalanobis distance between all rows of X and the vector μ with respect to σ .

Usage

```
maha(X, mu, sigma, ncores = 1, isChol = FALSE)
```

Arguments

<code>X</code>	matrix n by d where each row is a d dimensional random vector. Alternatively X can be a d -dimensional vector.
<code>mu</code>	vector of length d , representing the central position.
<code>sigma</code>	covariance matrix ($d \times d$). Alternatively it can be the cholesky decomposition of the covariance. In that case <code>isChol</code> should be set to <code>TRUE</code> .
<code>ncores</code>	Number of cores used. The parallelization will take place only if OpenMP is supported.
<code>isChol</code>	boolean set to true if <code>sigma</code> is the cholesky decomposition of the covariance matrix.

Value

a vector of length n where the i -th entry contains the square mahalanobis distance i -th random vector.

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>

Examples

```

N <- 100
d <- 5
mu <- 1:d
X <- t(t(matrix(rnorm(N*d), N, d)) + mu)
tmp <- matrix(rnorm(d^2), d, d)
mcov <- tcrossprod(tmp, tmp)
myChol <- chol(mcov)

rbind(head(maha(X, mu, mcov), 10),
      head(maha(X, mu, myChol, isChol = TRUE), 10),
      head(mahalanobis(X, mu, mcov), 10))

## Not run:
# Performance comparison: microbenchmark does not work on all
# platforms, hence we need to check whether it is installed
if( "microbenchmark" %in% rownames(installed.packages()) ){
  library(microbenchmark)

  a <- cbind(
    maha(X, mu, mcov),
    maha(X, mu, myChol, isChol = TRUE),
    mahalanobis(X, mu, mcov))

  # Same output as mahalanobis
  a[ , 1] / a[ , 3]
  a[ , 2] / a[ , 3]

  microbenchmark(maha(X, mu, mcov),
                 maha(X, mu, myChol, isChol = TRUE),
                 mahalanobis(X, mu, mcov))
}

## End(Not run)

```

Description

Given a sample from a d -dimensional distribution, an initialization point and a bandwidth the algorithm finds the nearest mode of the corresponding Gaussian kernel density.

Usage

```
ms(X, init, H, tol = 1e-06, ncores = 1, store = FALSE)
```

Arguments

<code>X</code>	n by d matrix containing the data.
<code>init</code>	d-dimensional vector containing the initial point for the optimization.
<code>H</code>	Positive definite bandwidth matrix representing the covariance of each component of the Gaussian kernel density.
<code>tol</code>	Tolerance used to assess the convergence of the algorithm, which is stopped if the absolute values of increments along all the dimensions are smaller then tol at any iteration. Default value is 1e-6.
<code>ncores</code>	Number of cores used. The parallelization will take place only if OpenMP is supported.
<code>store</code>	If FALSE only the latest iteration is returned, if TRUE the function will return a matrix where the i-th row is the position of the algorithms at the i-th iteration.

Value

A list where `estim` is a d-dimensional vector containing the last position of the algorithm, while `traj` is a matrix with d-columns representing the trajectory of the algorithm along each dimension. If `store == FALSE` the whole trajectory is not stored and `traj = NULL`.

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>.

Examples

```
set.seed(434)

# Simulating multivariate normal data
N <- 1000
mu <- c(1, 2)
sigma <- matrix(c(1, 0.5, 0.5, 1), 2, 2)
X <- rmvn(N, mu = mu, sigma = sigma)

# Plotting the true density function
steps <- 100
range1 <- seq(min(X[, 1]), max(X[, 1]), length.out = steps)
range2 <- seq(min(X[, 2]), max(X[, 2]), length.out = steps)
grid <- expand.grid(range1, range2)
vals <- dmvn(as.matrix(grid), mu, sigma)

contour(z = matrix(vals, steps, steps), x = range1, y = range2, xlab = "X1", ylab = "X2")
points(X[, 1], X[, 2], pch = '.')

# Estimating the mode from "nrep" starting points
nrep <- 10
```

```

index <- sample(1:N, nrep)
for(ii in 1:nrep) {
  start <- X[index[ii], ]
  out <- ms(X, init = start, H = 0.1 * sigma, store = TRUE)
  lines(out$traj[ , 1], out$traj[ , 2], col = 2, lwd = 2)
  points(out$final[1], out$final[2], col = 4, pch = 3, lwd = 3) # Estimated mode (blue)
  points(start[1], start[2], col = 2, pch = 3, lwd = 3)      # ii-th starting value
}

```

rmixn

Fast simulation of r.v.s from a mixture of multivariate normal densities

Description

Fast simulation of r.v.s from a mixture of multivariate normal densities

Usage

```

rmixn(
  n,
  mu,
  sigma,
  w,
  ncores = 1,
  isChol = FALSE,
  retInd = FALSE,
  A = NULL,
  kpnames = FALSE
)

```

Arguments

n	number of random vectors to be simulated.
mu	an (m x d) matrix, where m is the number of mixture components.
sigma	as list of m covariance matrices (d x d) on for each mixture component. Alternatively it can be a list of m cholesky decomposition of the covariance. In that case isChol should be set to TRUE.
w	vector of length m, containing the weights of the mixture components.
ncores	Number of cores used. The parallelization will take place only if OpenMP is supported.
isChol	boolean set to true is sigma is the cholesky decomposition of the covariance matrix.
retInd	when set to TRUE an attribute called "index" will be added to the output matrix of random variables. This is a vector specifying to which mixture components each random vector belongs. FALSE by default.

A	an (optional) numeric matrix of dimension (n x d), which will be used to store the output random variables. It is useful when n and d are large and one wants to call <code>rmvn()</code> several times, without reallocating memory for the whole matrix each time. NB: the element of A must be of class "numeric".
kpnames	if TRUE the dimensions' names are preserved. That is, the i-th column of the output has the same name as the i-th entry of mu or the i-th column of sigma. <code>kpnames==FALSE</code> by default.

Details

Notice that this function does not use one of the Random Number Generators (RNGs) provided by R, but one of the parallel cryptographic RNGs described in (Salmon et al., 2011). It is important to point out that this RNG can safely be used in parallel, without risk of collisions between parallel sequence of random numbers. The initialization of the RNG depends on R's seed, hence the `set.seed()` function can be used to obtain reproducible results. Notice though that changing `ncores` causes most of the generated numbers to be different even if R's seed is the same (see example below). NB: at the moment the RNG does not work properly on Solaris OS when `ncores>1`. Hence, `rmixn()` checks if the OS is Solaris and, if this the case, it imposes `ncores==1`.

Value

If `A==NULL` (default) the output is an (n x d) matrix where the i-th row is the i-th simulated vector. If `A!=NULL` then the random vector are store in A, which is provided by the user, and the function returns NULL. Notice that if `retInd==TRUE` an attribute called "index" will be added to A. This is a vector specifying to which mixture components each random vector belongs.

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>, C++ RNG engine by Thijs van den Berg <<http://sitmo.com/>>.

References

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw (2011). Parallel Random Numbers: As Easy as 1, 2, 3. D. E. Shaw Research, New York, NY 10036, USA.

Examples

```
# Create mixture of two components
mu <- matrix(c(1, 2, 10, 20), 2, 2, byrow = TRUE)
sigma <- list(diag(c(1, 10)), matrix(c(1, -0.9, -0.9, 1), 2, 2))
w <- c(0.1, 0.9)

# Simulate
X <- rmixn(1e4, mu, sigma, w, retInd = TRUE)
plot(X, pch = '.', col = attr(X, "index"))

# Simulate with fixed seed
set.seed(414)
rmixn(4, mu, sigma, w)
```

```

set.seed(414)
rmixn(4, mu, sigma, w)

set.seed(414)
rmixn(4, mu, sigma, w, ncores = 2) # r.v. generated on the second core are different

##### Here we create the matrix that will hold the simulated random variables upfront.
A <- matrix(NA, 4, 2)
class(A) <- "numeric" # This is important. We need the elements of A to be of class "numeric".

set.seed(414)
rmixn(4, mu, sigma, w, ncores = 2, A = A) # This returns NULL ...
A                                           # ... but the result is here

```

rmixt	<i>Fast simulation of r.v.s from a mixture of multivariate Student's t densities</i>
-------	--

Description

Fast simulation of r.v.s from a mixture of multivariate Student's t densities

Usage

```

rmixt(
  n,
  mu,
  sigma,
  df,
  w,
  ncores = 1,
  isChol = FALSE,
  retInd = FALSE,
  A = NULL,
  kpnames = FALSE
)

```

Arguments

n	number of random vectors to be simulated.
mu	an (m x d) matrix, where m is the number of mixture components.
sigma	as list of m covariance matrices (d x d) on for each mixture component. Alternatively it can be a list of m cholesky decomposition of the covariance. In that case isChol should be set to TRUE.
df	a positive scalar representing the degrees of freedom. All the densities in the mixture have the same df.

w	vector of length m, containing the weights of the mixture components.
ncores	Number of cores used. The parallelization will take place only if OpenMP is supported.
isChol	boolean set to true is sigma is the cholesky decomposition of the covariance matrix.
retInd	when set to TRUE an attribute called "index" will be added to the output matrix of random variables. This is a vector specifying to which mixture components each random vector belongs. FALSE by default.
A	an (optional) numeric matrix of dimension (n x d), which will be used to store the output random variables. It is useful when n and d are large and one wants to call <code>rmvn()</code> several times, without reallocating memory for the whole matrix each time. NB: the element of A must be of class "numeric".
kpnames	if TRUE the dimensions' names are preserved. That is, the i-th column of the output has the same name as the i-th entry of mu or the i-th column of sigma. <code>kpnames==FALSE</code> by default.

Details

There are many candidates for the multivariate generalization of Student's t-distribution, here we use the parametrization described here https://en.wikipedia.org/wiki/Multivariate_t-distribution.

Notice that this function does not use one of the Random Number Generators (RNGs) provided by R, but one of the parallel cryptographic RNGs described in (Salmon et al., 2011). It is important to point out that this RNG can safely be used in parallel, without risk of collisions between parallel sequence of random numbers. The initialization of the RNG depends on R's seed, hence the `set.seed()` function can be used to obtain reproducible results. Notice though that changing `ncores` causes most of the generated numbers to be different even if R's seed is the same (see example below). NB: at the moment the parallelization does not work properly on Solaris OS when `ncores>1`. Hence, `rmixt()` checks if the OS is Solaris and, if this the case, it imposes `ncores==1`

Value

If `A==NULL` (default) the output is an (n x d) matrix where the i-th row is the i-th simulated vector. If `A!=NULL` then the random vector are store in A, which is provided by the user, and the function returns NULL. Notice that if `retInd==TRUE` an attribute called "index" will be added to A. This is a vector specifying to which mixture components each random vector belongs.

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>, C++ RNG engine by Thijs van den Berg <<http://sitmo.com/>>.

References

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw (2011). Parallel Random Numbers: As Easy as 1, 2, 3. D. E. Shaw Research, New York, NY 10036, USA.

Examples

```

# Create mixture of two components
df <- 6
mu <- matrix(c(1, 2, 10, 20), 2, 2, byrow = TRUE)
sigma <- list(diag(c(1, 10)), matrix(c(1, -0.9, -0.9, 1), 2, 2))
w <- c(0.1, 0.9)

# Simulate
X <- rmixt(1e4, mu, sigma, df, w, retInd = TRUE)
plot(X, pch = '.', col = attr(X, "index"))

# Simulate with fixed seed
set.seed(414)
rmixt(4, mu, sigma, df, w)

set.seed(414)
rmixt(4, mu, sigma, df, w)

set.seed(414)
rmixt(4, mu, sigma, df, w, ncores = 2) # r.v. generated on the second core are different

##### Here we create the matrix that will hold the simulated random variables upfront.
A <- matrix(NA, 4, 2)
class(A) <- "numeric" # This is important. We need the elements of A to be of class "numeric".

set.seed(414)
rmixt(4, mu, sigma, df, w, ncores = 2, A = A) # This returns NULL ...
A # ... but the result is here

```

 rmvn

Fast simulation of multivariate normal random variables

Description

Fast simulation of multivariate normal random variables

Usage

```
rmvn(n, mu, sigma, ncores = 1, isChol = FALSE, A = NULL, kpname = FALSE)
```

Arguments

n	number of random vectors to be simulated.
mu	vector of length d, representing the mean.
sigma	covariance matrix (d x d). Alternatively it can be the cholesky decomposition of the covariance. In that case isChol should be set to TRUE.

<code>ncores</code>	Number of cores used. The parallelization will take place only if OpenMP is supported.
<code>isChol</code>	boolean set to true is <code>sigma</code> is the cholesky decomposition of the covariance matrix.
<code>A</code>	an (optional) numeric matrix of dimension $(n \times d)$, which will be used to store the output random variables. It is useful when n and d are large and one wants to call <code>rmvn()</code> several times, without reallocating memory for the whole matrix each time. NB: the element of <code>A</code> must be of class "numeric".
<code>kpnames</code>	if TRUE the dimensions' names are preserved. That is, the i -th column of the output has the same name as the i -th entry of <code>mu</code> or the i -th column of <code>sigma</code> . <code>kpnames==FALSE</code> by default.

Details

Notice that this function does not use one of the Random Number Generators (RNGs) provided by R, but one of the parallel cryptographic RNGs described in (Salmon et al., 2011). It is important to point out that this RNG can safely be used in parallel, without risk of collisions between parallel sequence of random numbers. The initialization of the RNG depends on R's seed, hence the `.seed()` function can be used to obtain reproducible results. Notice though that changing `ncores` causes most of the generated numbers to be different even if R's seed is the same (see example below). NB: at the moment the RNG does not work properly on Solaris OS when `ncores>1`. Hence, `rmvn()` checks if the OS is Solaris and, if this the case, it imposes `ncores==1`.

Value

If `A==NULL` (default) the output is an $(n \times d)$ matrix where the i -th row is the i -th simulated vector. If `A!=NULL` then the random vector are store in `A`, which is provided by the user, and the function returns `NULL`.

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>, C++ RNG engine by Thijs van den Berg <<http://sitmo.com/>>.

References

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw (2011). Parallel Random Numbers: As Easy as 1, 2, 3. D. E. Shaw Research, New York, NY 10036, USA.

Examples

```
d <- 5
mu <- 1:d

# Creating covariance matrix
tmp <- matrix(rnorm(d^2), d, d)
mcov <- tcrossprod(tmp, tmp)

set.seed(414)
rmvn(4, 1:d, mcov)
```

```

set.seed(414)
rmvn(4, 1:d, mcov)

set.seed(414)
rmvn(4, 1:d, mcov, ncores = 2) # r.v. generated on the second core are different

##### Here we create the matrix that will hold the simulated random variables upfront.
A <- matrix(NA, 4, d)
class(A) <- "numeric" # This is important. We need the elements of A to be of class "numeric".

set.seed(414)
rmvn(4, 1:d, mcov, ncores = 2, A = A) # This returns NULL ...
A                                     # ... but the result is here

```

 rmvt

Fast simulation of multivariate Student's t random variables

Description

Fast simulation of multivariate Student's t random variables

Usage

```
rmvt(n, mu, sigma, df, ncores = 1, isChol = FALSE, A = NULL, kpnames = FALSE)
```

Arguments

n	number of random vectors to be simulated.
mu	vector of length d, representing the mean of the distribution.
sigma	scale matrix (d x d). Alternatively it can be the cholesky decomposition of the scale matrix. In that case isChol should be set to TRUE. Notice that if the degrees of freedom (the argument df) is larger than 2, the $\text{Cov}(X) = \text{sigma} * \text{df} / (\text{df} - 2)$.
df	a positive scalar representing the degrees of freedom.
ncores	Number of cores used. The parallelization will take place only if OpenMP is supported.
isChol	boolean set to true if sigma is the cholesky decomposition of the covariance matrix.
A	an (optional) numeric matrix of dimension (n x d), which will be used to store the output random variables. It is useful when n and d are large and one wants to call <code>rmvn()</code> several times, without reallocating memory for the whole matrix each time. NB: the element of A must be of class "numeric".
kpnames	if TRUE the dimensions' names are preserved. That is, the i-th column of the output has the same name as the i-th entry of mu or the i-th column of sigma. <code>kpnames==FALSE</code> by default.

Details

There are in fact many candidates for the multivariate generalization of Student's t-distribution, here we use the parametrization described here https://en.wikipedia.org/wiki/Multivariate_t-distribution.

Notice that `rmvt()` does not use one of the Random Number Generators (RNGs) provided by R, but one of the parallel cryptographic RNGs described in (Salmon et al., 2011). It is important to point out that this RNG can safely be used in parallel, without risk of collisions between parallel sequence of random numbers. The initialization of the RNG depends on R's seed, hence the `set.seed()` function can be used to obtain reproducible results. Notice though that changing `ncores` causes most of the generated numbers to be different even if R's seed is the same (see example below). NB: at the moment the RNG does not work properly on Solaris OS when `ncores`>1. Hence, `rmvt()` checks if the OS is Solaris and, if this the case, it imposes `ncores`==1.

Value

If `A`==NULL (default) the output is an (n x d) matrix where the i-th row is the i-th simulated vector. If `A`!=NULL then the random vector are store in `A`, which is provided by the user, and the function returns NULL.

Author(s)

Matteo Fasiolo <matteo.fasiolo@gmail.com>, C++ RNG engine by Thijs van den Berg <<http://sitmo.com/>>.

References

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw (2011). Parallel Random Numbers: As Easy as 1, 2, 3. D. E. Shaw Research, New York, NY 10036, USA.

Examples

```
d <- 5
mu <- 1:d
df <- 4

# Creating covariance matrix
tmp <- matrix(rnorm(d^2), d, d)
mcov <- tcrossprod(tmp, tmp) + diag(0.5, d)

set.seed(414)
rmvt(4, 1:d, mcov, df = df)

set.seed(414)
rmvt(4, 1:d, mcov, df = df)

set.seed(414)
rmvt(4, 1:d, mcov, df = df, ncores = 2) # These will not match the r.v. generated on a single core.

##### Here we create the matrix that will hold the simulated random variables upfront.
A <- matrix(NA, 4, d)
class(A) <- "numeric" # This is important. We need the elements of A to be of class "numeric".
```


Index

dmixn, 2
dmixt, 4
dmvn, 5
dmvt, 7

maha, 8
ms, 9

rmixn, 11
rmixt, 13
rmvn, 15
rmvt, 17