Package: rxode2random (via r-universe)

July 17, 2024

Title Random Number Generation Functions for 'rxode2'

Version 2.1.1.9000

Description Provides the random number generation (in parallel) needed for 'rxode2' (Wang, Hallow and James (2016) <doi:10.1002/psp4.12052>) and 'nlmixr2' (Fidler et al (2019) <doi:10.1002/psp4.12445>). This split will reduce computational burden of recompiling 'rxode2'.

License GPL (>= 3)

URL `https://nlmixr2.github.io/rxode2random/`, `https://github.com/nlmixr2/rxode2random/`

BugReports `https://github.com/nlmixr2/rxode2random/issues/`

Depends R (>= 4.0.0)

Imports stats, Rcpp, checkmate, lotri, rxode2parse (>= 2.0.19)

Suggests testthat (>= 3.0.0)

LinkingTo sitmo, rxode2parse (>= 2.0.19), Rcpp, RcppArmadillo, BH

Biarch true

Config/testthat/edition 3

Encoding UTF-8

Language en-US

NeedsCompilation yes

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Repository https://nlmixr2.r-universe.dev

RemoteUrl https://github.com/nlmixr2/rxode2random

RemoteRef HEAD

RemoteSha 0d07eebe7253c89d5335d3909ee3e7f67fee5b04

Contents

`.cbindOme` *cbind Ome*

Description

cbind Ome

Usage

```
.cbindOme(et, mat, n)
```

Arguments

`et`	The theta data frame
`mat`	The full matrix simulation from omegas
`n`	number of subject simulated

Value

data frame with et combined with simulated omega matrix values

Author(s)

Matthew Fidler

`.vecDf` *Convert numeric vector to repeated data.frame*

Description

Convert numeric vector to repeated data.frame

Usage

```
.vecDf(vec, n)
```

Arguments

`vec`	Named input vector
`n`	Number of columns

Value

Data frame with repeated vec

Author(s)

Matthew Fidler

`cvPost` *Sample a covariance Matrix from the Posterior Inverse Wishart distribution.*

Description

Note this Inverse wishart rescaled to match the original scale of the covariance matrix.

Usage

```
cvPost(
  nu,
  omega,
  n = 1L,
  omegaIsChol = FALSE,
  returnChol = FALSE,
  type = c("invWishart", "lkj", "separation"),
  diagXformType = c("log", "identity", "variance", "nlmixrSqrt", "nlmixrLog",
    "nlmixrIdentity")
)
```

Arguments

`nu`	Degrees of Freedom (Number of Observations) for covariance matrix simulation.
`omega`	Either the estimate of covariance matrix or the estimated standard deviations in matrix form each row forming the standard deviation simulated values
`n`	Number of Matrices to sample. By default this is 1. This is only useful when `omega` is a matrix. Otherwise it is determined by the number of rows in the input `omega` matrix of standard deviations
`omegaIsChol`	is an indicator of if the omega matrix is in the Cholesky decomposition. This is only used when `type="invWishart"`
`returnChol`	Return the Cholesky decomposition of the covariance matrix sample. This is only used when `type="invWishart"`
`type`	The type of covariance posterior that is being simulated. This can be: • `invWishart` The posterior is an inverse wishart; This allows for correlations between parameters to be modeled. All the uncertainty in the parameter is captured in the degrees of freedom parameter. • `lkj` The posterior separates the standard deviation estimates (modeled outside and provided in the `omega` argument) and the correlation estimates. The correlation estimate is simulated with the `rLKJ1()`. This simulation uses the relationship `eta=(nu-1)/2`. This is relationship based on the proof of the relationship between the restricted LKJ-distribution and inverse wishart distribution (XXXXXX). Once the correlation posterior is calculated, the estimated standard deviations are then combined with the simulated correlation matrix to create the covariance matrix. • `separation` Like the `lkj` option, this separates out the estimation of the correlation and standard deviation. Instead of using the `LKJ` distribution to simulate the correlation, it simulates the inverse wishart of the identity matrix and converts the result to a correlation matrix. This correlation matrix is then used with the standard deviation to calculate the simulated covariance matrix.
`diagXformType`	Diagonal transformation type. These could be: • `log` The standard deviations are log transformed, so the actual standard deviations are exp(omega)

- `identity` The standard deviations are not transformed. The standard deviations are not transformed; They should be positive.
- `variance` The variances are specified in the `omega` matrix; They are transformed into standard deviations.
- `nlmixrSqrt` These standard deviations come from an nlmixr omega matrix where diag(chol(inv(omega))) = x^2
- `nlmixrLog` These standard deviations come from a nlmixr omega matrix omega matrix where diag(chol(solve(omega))) = exp(x)
- `nlmixrIdentity` These standard deviations come from a nlmixr omega matrix omega matrix where diag(chol(solve(omega))) = x

The nlmixr transformations only make sense when there is no off-diagonal correlations modeled.

Details

If your covariance matrix is a 1x1 matrix, this uses an scaled inverse chi-squared which is equivalent to the Inverse Wishart distribution in the uni-directional case.

In general, the separation strategy is preferred for diagonal matrices. If the dimension of the matrix is below 10, `lkj` is numerically faster than `separation` method. However, the `lkj` method has densities too close to zero (XXXX) when the dimension is above 10. In that case, though computationally more expensive `separation` method performs better.

For matrices with modeled covariances, the easiest method to use is the inverse Wishart which allows the simulation of correlation matrices (XXXX). This method is more well suited for well behaved matrices, that is the variance components are not too low or too high. When modeling nonlinear mixed effects modeling matrices with too high or low variances are considered sub-optimal in describing a system. With these rules in mind, it is reasonable to use the inverse Wishart.

Value

a matrix (n=1) or a list of matrices (n > 1)

Author(s)

Matthew L.Fidler & Wenping Wang

References

Alvarez I, Niemi J and Simpson M. (2014) *Bayesian Inference for a Covariance Matrix*. Conference on Applied Statistics in Agriculture.

Wang1 Z, Wu Y, and Chu H. (2018) *On Equivalence of the LKJ distribution and the restricted Wishart distribution*. <doi:10.48550/arXiv.1809.047463

Examples

```
## Sample a single covariance.
draw1 <- cvPost(3, matrix(c(1, .3, .3, 1), 2, 2))

## Sample 3 covariances
```

```
set.seed(42)
draw3 <- cvPost(3, matrix(c(1, .3, .3, 1), 2, 2), n = 3)

## Sample 3 covariances, but return the cholesky decomposition
set.seed(42)
draw3c <- cvPost(3, matrix(c(1, .3, .3, 1), 2, 2), n = 3, returnChol = TRUE)

## Sample 3 covariances with lognormal standard deviations via LKJ
## correlation sample
cvPost(3, sapply(1:3, function(...) {
  rnorm(10)
}), type = "lkj")

## or return cholesky decomposition
cvPost(3, sapply(1:3, function(...) {
  rnorm(10)
}),
type = "lkj",
returnChol = TRUE
)

## Sample 3 covariances with lognormal standard deviations via separation
## strategy using inverse Wishart correlation sample
cvPost(3, sapply(1:3, function(...) {
  rnorm(10)
}), type = "separation")

## or returning the cholesky decomposition
cvPost(3, sapply(1:3, function(...) {
  rnorm(10)
}),
type = "separation",
returnChol = TRUE
)
```

dfWishart *This uses simulations to match the rse*

Description

This uses simulations to match the rse

Usage

```
dfWishart(omega, n, rse, upper, totN = 1000, diag = TRUE, seed = 1234)
```

Arguments

omega	represents the matrix for simulation

`n`	This represents the number of subjects/samples this comes from (used to calculate rse). When present it assumes the rse= sqrt(2)/sqrt(n)
`rse`	This is the rse that we try to match, if not specified, it is derived from `n`
`upper`	The upper boundary for root finding in terms of degrees of freedom. If not specified, it is n*200
`totN`	This represents the total number of simulated inverse wishart deviates
`diag`	When `TRUE`, represents the rse to match is the diagonals, otherwise it is the total matrix.
`seed`	to make the simulation reproducible, this represents the seed that is used for simulating the inverse Wishart distribution

Value

output from `uniroot()` to find the right estimate

Author(s)

Matthew L. Fidler

Examples

```
dfWishart(lotri::lotri(a+b~c(1, 0.5, 1)), 100)
```

`phi` *Cumulative distribution of standard normal*

Description

Cumulative distribution of standard normal

Usage

```
phi(q)
```

Arguments

`q`	vector of quantiles

Value

cumulative distribution of standard normal distribution

Author(s)

Matthew Fidler

Examples

```
# phi is equivalent to pnorm(x)
phi(3)

# See
pnorm(3)

# This is provided for NONMEM-like compatibility in rxode2 models
```

`rinvchisq` *Scaled Inverse Chi Squared distribution*

Description

Scaled Inverse Chi Squared distribution

Usage

```
rinvchisq(n = 1L, nu = 1, scale = 1)
```

Arguments

`n`	Number of random samples
`nu`	degrees of freedom of inverse chi square
`scale`	Scale of inverse chi squared distribution (default is 1).

Value

a vector of inverse chi squared deviates.

Examples

```
rinvchisq(3, 4, 1) ## Scale = 1, degrees of freedom = 4
rinvchisq(2, 4, 2) ## Scale = 2, degrees of freedom = 4
```

rxbeta *Simulate beta variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxbeta(shape1, shape2, n = 1L, ncores = 1L)
```

Arguments

`shape1, shape2`	non-negative parameters of the Beta distribution.
`n`	number of observations. If `length(n)` > 1, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

beta random deviates

Examples

```
## Use threefry engine

rxbeta(0.5, 0.5, n = 10) # with rxbeta you have to explicitly state n
rxbeta(5, 1, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxbeta(1, 3)
```

rxbinom *Simulate Binomial variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxbinom(size, prob, n = 1L, ncores = 1L)
```

Arguments

`size`	number of trials (zero or more).
`prob`	probability of success on each trial.
`n`	number of observations. If `length(n)` > 1, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

binomial random deviates

Examples

```
## Use threefry engine

rxbinom(10, 0.9, n = 10) # with rxbinom you have to explicitly state n
rxbinom(3, 0.5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxbinom(4, 0.7)
```

rxcauchy	*Simulate Cauchy variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxcauchy(location = 0, scale = 1, n = 1L, ncores = 1L)
```

Arguments

`location, scale`	location and scale parameters.
`n`	number of observations. If `length(n)` > 1, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

Cauchy random deviates

Examples

```
## Use threefry engine

rxcauchy(0, 1, n = 10) # with rxcauchy you have to explicitly state n
rxcauchy(0.5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxcauchy(3)
```

rxchisq *Simulate chi-squared variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxchisq(df, n = 1L, ncores = 1L)
```

Arguments

`df`	degrees of freedom (non-negative, but can be non-integer).
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

chi squared random deviates

Examples

```
## Use threefry engine

rxchisq(0.5, n = 10) # with rxchisq you have to explicitly state n
rxchisq(5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxchisq(1)
```

rxexp — *Simulate exponential variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxexp(rate, n = 1L, ncores = 1L)
```

Arguments

`rate`	vector of rates.
`n`	number of observations. If `length(n)` > 1, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

exponential random deviates

Examples

```
## Use threefry engine

rxexp(0.5, n = 10) # with rxexp you have to explicitly state n
rxexp(5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxexp(1)
```

rxf *Simulate F variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxf(df1, df2, n = 1L, ncores = 1L)
```

Arguments

df1, df2	degrees of freedom. Inf is allowed.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

f random deviates

Examples

```
## Use threefry engine

rxf(0.5, 0.5, n = 10) # with rxf you have to explicitly state n
rxf(5, 1, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxf(1, 3)
```

rxgamma *Simulate gamma variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the sitmo threefry, this currently generates one random deviate from the uniform distribution to seed the engine threefry and then run the code.

Usage

```
rxgamma(shape, rate = 1, n = 1L, ncores = 1L)
```

Arguments

shape	The shape of the gamma random variable
rate	an alternative way to specify the scale.
n	number of observations. If length(n) > 1, the length is taken to be the number required.
ncores	Number of cores for the simulation rxnorm simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the rxode2 environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with rxSetSeed()

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

gamma random deviates

Examples

```
## Use threefry engine

rxgamma(0.5, n = 10) # with rxgamma you have to explicitly state n
rxgamma(5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxgamma(1)
```

`rxgeom` *Simulate geometric variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxgeom(prob, n = 1L, ncores = 1L)
```

Arguments

`prob`	probability of success in each trial. `0` < `prob` <= `1`.
`n`	number of observations. If `length(n)` > `1`, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

geometric random deviates

Examples

```
## Use threefry engine

rxgeom(0.5, n = 10) # with rxgeom you have to explicitly state n
rxgeom(0.25, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxgeom(0.75)
```

rxGetSeed — *Get the rxode2 seed*

Description

Get the rxode2 seed

Usage

```
rxGetSeed()
```

Value

rxode2 seed state or -1 when the seed isn't set

See Also

rxSetSeed, rxWithSeed, rxWithPreserveSeed

Examples

```
# without setting seed

rxGetSeed()
# Now set the seed
rxSetSeed(42)

rxGetSeed()

rxnorm()

rxGetSeed()

# don't use the rxode2 seed again

rxSetSeed(-1)

rxGetSeed()

rxnorm()

rxGetSeed()
```

`rxnbinom`	*Simulate Binomial variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described `https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/`. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxnbinom(size, prob, n = 1L, ncores = 1L)

rxnbinomMu(size, mu, n = 1L, ncores = 1L)
```

Arguments

`size`	target for number of successful trials, or dispersion parameter (the shape parameter of the gamma mixing distribution). Must be strictly positive, need not be integer.
`prob`	probability of success in each trial. `0` < `prob` <= `1`.
`n`	number of observations. If `length(n)` > `1`, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;
`mu`	alternative parametrization via mean: see 'Details'.

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

negative binomial random deviates. Note that `rxbinom2` uses the `mu` parameterization an the `rxbinom` uses the `prob` parameterization (`mu=size/(prob+size)`)

Examples

```
## Use threefry engine

rxnbinom(10, 0.9, n = 10) # with rxbinom you have to explicitly state n
rxnbinom(3, 0.5, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxnbinom(4, 0.7)

# use mu parameter
rxnbinomMu(40, 40, n=10)
```

`rxnorm` *Simulate random normal variable from threefry/vandercorput generator*

Description

Simulate random normal variable from threefry/vandercorput generator

Usage

```
rxnorm(mean = 0, sd = 1, n = 1L, ncores = 1L)
```

Arguments

`mean`	vector of means.
`sd`	vector of standard deviations.
`n`	number of observations
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Value

normal random number deviates

Examples

```
## Use threefry engine

rxnorm(n = 10) # with rxnorm you have to explicitly state n
rxnorm(n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxnorm(2, 3) ## The first 2 arguments are the mean and standard deviation
```

`rxode2randomMd5` *Get the MD5 hash of the current rxode2random revision*

Description

Get the MD5 hash of the current rxode2random revision

Usage

```
rxode2randomMd5()
```

Value

md5 hash of rxode2random revision

Author(s)

Matthew L. Fidler

Examples

```
rxode2randomMd5()
```

`rxord` *Simulate ordinal value*

Description

Simulate ordinal value

Usage

```
rxord(...)
```

Arguments

`...`	the probabilities to be simulated. These should sum up to a number below one.

Details

The values entered into the 'rxord' simulation will simulate the probability of falling each group. If it falls outside of the specified probabilities, it will simulate the group (number of probabilities specified + 1)

Value

A number from 1 to the (number of probabilities specified + 1)

Author(s)

Matthew L. Fidler

Examples

```
# This will give values 1, and 2
rxord(0.5)
rxord(0.5)
rxord(0.5)
rxord(0.5)

# This will give values 1, 2 and 3
rxord(0.3, 0.3)
rxord(0.3, 0.3)
rxord(0.3, 0.3)
```

`rxpois` *Simulate random Poisson variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described `https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/`. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxpois(lambda, n = 1L, ncores = 1L)
```

Arguments

`lambda`	vector of (non-negative) means.
`n`	number of random values to return.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

poission random number deviates

Examples

```
## Use threefry engine

rxpois(lambda = 3, n = 10) # with rxpois you have to explicitly state n
rxpois(lambda = 3, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxpois(4) ## The first arguments are the lambda parameter
```

`rxPp` *Simulate a from a Poisson process*

Description

Simulate a from a Poisson process

Usage

```
rxPp(
  n,
  lambda,
  gamma = 1,
  prob = NULL,
  t0 = 0,
  tmax = Inf,
  randomOrder = FALSE
)
```

Arguments

`n`	Number of time points to simulate in the Poisson process
`lambda`	Rate of Poisson process
`gamma`	Asymmetry rate of Poisson process. When gamma=1.0, this simulates a homogenous Poisson process. When gamma<1.0, the Poisson process has more events early, when gamma > 1.0, the Poisson process has more events late in the process. When gamma is non-zero, the tmax should not be infinite but indicate the end of the Poisson process to be simulated. In most pharamcometric cases, this will be the end of the study. Internally this uses a rate of: l(t) = lambda*gamma*(t/tmax)^(gamma-1)
`prob`	When specified, this is a probability function with one argument, time, that gives the probability that a Poisson time t is accepted as a rejection time.

`t0`	the starting time of the Poisson process
`tmax`	the maximum time of the Poisson process
`randomOrder`	when `TRUE` randomize the order of the Poisson events. By default (`FALSE`) it returns the Poisson process is in order of how the events occurred.

Value

This returns a vector of the Poisson process times; If the dropout is >= tmax, then all the rest of the times are = tmax to indicate the dropout is equal to or after tmax.

Author(s)

Matthew Fidler

Examples

```
## Sample homogenous Poisson process of rate 1/10
rxPp(10, 1 / 10)

## Sample inhomogenous Poisson rate of 1/10

rxPp(10, 1 / 10, gamma = 2, tmax = 100)

## Typically the Poisson process times are in a sequential order,
## using randomOrder gives the Poisson process in random order

rxPp(10, 1 / 10, gamma = 2, tmax = 10, randomOrder = TRUE)

## This uses an arbitrary function to sample a non-homogenous Poisson process

rxPp(10, 1 / 10, prob = function(x) {
  1 / x
})
```

`rxRmvn` *Simulate from a (truncated) multivariate normal*

Description

This is simulated with the fast, thread-safe threefry simulator and can use multiple cores to generate the random deviates.

Usage

```
rxRmvn(
  n,
  mu = NULL,
  sigma,
  lower = -Inf,
  upper = Inf,
  ncores = 1,
  isChol = FALSE,
  keepNames = TRUE,
  a = 0.4,
  tol = 2.05,
  nlTol = 1e-10,
  nlMaxiter = 100L
)
```

Arguments

`n`	Number of random row vectors to be simulated OR the matrix to use for simulation (faster).
`mu`	mean vector
`sigma`	Covariance matrix for multivariate normal or a list of covariance matrices. If a list of covariance matrix, each matrix will simulate `n` matrices and combine them to a full matrix
`lower`	is a vector of the lower bound for the truncated multivariate norm
`upper`	is a vector of the upper bound for the truncated multivariate norm
`ncores`	Number of cores used in the simulation
`isChol`	A boolean indicating if `sigma` is a cholesky decomposition of the covariance matrix.
`keepNames`	Keep the names from either the mean or covariance matrix.

`a`	threshold for switching between methods; They can be tuned for maximum speed; There are three cases that are considered:
	case 1: a < l < u
	case 2: l < u < -a
	case 3: otherwise
	where l=lower and u = upper

`tol`	When case 3 is used from the above possibilities, the tol value controls the acceptance rejection and inverse-transformation; When abs(u-l)>tol, uses accept-reject from randn
`nlTol`	Tolerance for newton line-search
`nlMaxiter`	Maximum iterations for newton line-search

Value

If `n==integer` (default) the output is an (n x d) matrix where the i-th row is the i-th simulated vector.

If `is.matrix(n)` then the random vector are store in `n`, which is provided by the user, and the function returns `NULL` invisibly.

Author(s)

Matthew Fidler, Zdravko Botev and some from Matteo Fasiolo

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.

The thread safe multivariate normal was inspired from the `mvnfast` package by Matteo Fasiolo https://CRAN.R-project.org/package=mvnfast

The concept of the truncated multivariate normal was taken from Zdravko Botev Botev (2017) doi:10.1111/rssb.12162 and Botev and L'Ecuyer (2015) doi:10.1109/WSC.2015.7408180 and converted to thread safe simulation;

Examples

```
## From mvnfast
## Unlike mvnfast, uses threefry simulation

d <- 5
mu <- 1:d

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


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

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

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

## You can also simulate from a truncated normal:

rxRmvn(10, 1:d, mcov, lower = 1:d - 1, upper = 1:d + 1)


# You can also simulate from different matrices (if they match
# dimensions) by using a list of matrices.
```

```
matL <- lapply(1:4, function(...) {
  tmp <- matrix(rnorm(d^2), d, d)
  tcrossprod(tmp, tmp)
})


rxRmvn(4, setNames(1:d, paste0("a", 1:d)), matL)
```

---

rxSetSeed *Set the parallel seed for rxode2 random number generation*

Description

This sets the seed for the rxode2 parallel random number generation. If set, then whenever a seed is set for the threefry or vandercorput simulation engine, it will use this seed, increment for the number of seeds and continue with the sequence the next time the random number generator is called.

Usage

```
rxSetSeed(seed)
```

Arguments

`seed`	An integer that represents the rxode2 parallel and internal random number generator seed. When positive, use this seed for random number generation and increment and reseed any parallel or new engines that are being called. When negative, turn off the rxode2 seed and generate a seed from the R's uniform random number generator. Best practice is to set this seed.

Details

In contrast, when this is not called, the time that the vandercorput or threefry simulation engines are seeded it comes from a uniform random number generated from the standard R random seed. This may cause a duplicate seed based on the R seed state. This means that there could be correlations between simulations that do not exist This will avoid the birthday problem picking exactly the same seed using the seed state of the R random number generator. The more times the seed is called, the more likely this becomes.

Value

Nothing, called for its side effects

Author(s)

Matthew Fidler

References

JD Cook. (2016). Random number generator seed mistakes. https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/

See Also

rxGetSeed, rxWithSeed, rxWithPreserveSeed

Examples

```
rxSetSeed(42)

# seed with generator 42
rxnorm()

# Use R's random number generator
rnorm(1)

rxSetSeed(42)

# reproduces the same number
rxnorm()

# But R's random number is not the same

rnorm(1)

# If we reset this to use the R's seed
# (internally rxode2 uses a uniform random number to span seeds)
# This can lead to duplicate sequences and seeds

rxSetSeed(-1)

# Now set seed works for both.

# This is not recommended, but illustrates the different types of
# seeds that can be generated.

set.seed(42)

rxnorm()

rnorm(1)

set.seed(42)

rxnorm()

rnorm(1)
```

rxt — *Simulate student t variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxt(df, n = 1L, ncores = 1L)
```

Arguments

`df`	degrees of freedom (> 0, maybe non-integer). `df = Inf` is allowed.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

t-distribution random numbers

Examples

```
## Use threefry engine

rxt(df = 3, n = 10) # with rxt you have to explicitly state n
rxt(df = 3, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxt(4) ## The first argument is the df parameter
```

`rxunif` *Simulate uniform variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxunif(min = 0, max = 1, n = 1L, ncores = 1L)
```

Arguments

`min, max`	lower and upper limits of the distribution. Must be finite.
`n`	number of observations. If `length(n)` > 1, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

uniform random numbers

Examples

```
## Use threefry engine

rxunif(min = 0, max = 4, n = 10) # with rxunif you have to explicitly state n
rxunif(min = 0, max = 4, n = 10, ncores = 2) # You can parallelize the simulation using openMP

rxunif()
```

rxweibull *Simulate Weibull variable from threefry generator*

Description

Care should be taken with this method not to encounter the birthday problem, described https://www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. Since the `sitmo` `threefry`, this currently generates one random deviate from the uniform distribution to seed the engine `threefry` and then run the code.

Usage

```
rxweibull(shape, scale = 1, n = 1L, ncores = 1L)
```

Arguments

`shape, scale`	shape and scale parameters, the latter defaulting to 1.
`n`	number of observations. If `length(n)` > 1, the length is taken to be the number required.
`ncores`	Number of cores for the simulation `rxnorm` simulates using the threefry sitmo generator;

Details

Therefore, a simple call to the random number generated followed by a second call to random number generated may have identical seeds. As the number of random number generator calls are increased the probability that the birthday problem will increase.

The key to avoid this problem is to either run all simulations in the `rxode2` environment once (therefore one seed or series of seeds for the whole simulation), pre-generate all random variables used for the simulation, or seed the rxode2 engine with `rxSetSeed()`

Internally each ID is seeded with a unique number so that the results do not depend on the number of cores used.

Value

Weibull random deviates

Examples

```
## Use threefry engine

# with rxweibull you have to explicitly state n
rxweibull(shape = 1, scale = 4, n = 10)

# You can parallelize the simulation using openMP
rxweibull(shape = 1, scale = 4, n = 10, ncores = 2)

rxweibull(3)
```

rxWithSeed *Preserved seed and possibly set the seed*

Description

Preserved seed and possibly set the seed

Usage

```
rxWithSeed(
  seed,
  code,
  rxseed = rxGetSeed(),
  kind = "default",
  normal.kind = "default",
  sample.kind = "default"
)

rxWithPreserveSeed(code)
```

Arguments

`seed`	R seed to use for the session
`code`	Is the code to evaluate
`rxseed`	is the rxode2 seed that is being preserved
`kind`	character or `NULL`. If `kind` is a character string, set R's RNG to the kind desired. Use `"default"` to return to the R default. See 'Details' for the interpretation of `NULL`.
`normal.kind`	character string or `NULL`. If it is a character string, set the method of Normal generation. Use `"default"` to return to the R default. `NULL` makes no change.
`sample.kind`	character string or `NULL`. If it is a character string, set the method of discrete uniform generation (used in `sample`, for instance). Use `"default"` to return to the R default. `NULL` makes no change.

Value

returns whatever the code is returning

See Also

rxGetSeed, rxSetSeed

Examples

```
rxGetSeed()
rxWithSeed(1, {
   print(rxGetSeed())
   rxnorm()
   print(rxGetSeed())
   rxnorm()
}, rxseed=3)
```

`swapMatListWithCube` *Swaps the matrix list with a cube*

Description

Swaps the matrix list with a cube

Usage

```
swapMatListWithCube(matrixListOrCube)
```

Arguments

`matrixListOrCube`	Either a list of 2-dimensional matrices or a cube of matrices

Value

A list or a cube (opposite format as input)

Author(s)

Matthew L. Fidler

Examples

```
# Create matrix list
matLst <- cvPost(10, lotri::lotri(a+b~c(1, 0.25, 1)), 3)
print(matLst)

# Convert to cube
matCube <- swapMatListWithCube(matLst)
print(matCube)

# Convert back to list
matLst2 <- swapMatListWithCube(matCube)
print(matLst2)
```

Index