

Package ‘randtoolbox’

December 11, 2009

Type Package

Title toolbox for pseudo and quasi random number generation and RNG tests.

Version 1.09

Date 2009-12-01

Author Yohan Chalabi, Christophe Dutang, Petr Savicky and Diethelm Wuertz (except underlying C codes of (i) the SFMT algorithm from M. Matsumoto and M. Saito, (ii) the Knuth-TAOCP RNG from D. Knuth). See LICENCE file for details.

Maintainer Christophe Dutang <christophe.dutang@ensimag.fr>

Description The package provides (1) pseudo random generators - general linear congruential generators (Park Miller) and multiple recursive generators (Knuth TAOCP), generalized feedback shift register (SF-Mersenne Twister algorithm and WELL generators); (2) quasi random generators - the Torus algorithm, the Sobol sequence, the Halton sequence (thus include Van der Corput sequence) and (3) some additional tests such as the gap test, the serial test, the poker test... For true random number generation, use the ‘random’ package, for Latin Hypercube Sampling (a hybrid qmc method), use the ‘lhs’ package, a number of RNGs and tests for RNGs are provided by ‘RDieHarder’, all available on CRAN. There is also a small stand-alone package ‘rngwell19937’ for the WELL19937a RNG.

Depends R (>= 2.6.0), rngWELL

Imports rngWELL

License BSD

LazyLoad yes

Repository CRAN

Date/Publication 2009-12-11 19:01:26

R topics documented:

randtoolbox-package	2
auxiliary	3
coll.test	4
freq.test	6
gap.test	8
get.primes	9
order.test	10
poker.test	12
pseudo.randtoolbox	13
quasi.randtoolbox	19
runifInterface	23
serial.test	25
Index	27

randtoolbox-package

Toolbox for pseudo and quasi random number generation

Description

The randtoolbox-package started in 2007 during an ISFA (France) working group. From then, it grew quickly thanks to the contribution of Diethelm Wuertz and Petr Savicky. It was presented in the Rmetrics Meielisalp workshop in 2009. Slides can be found here: <http://www.rmetrics.org/Meielisalp2009/Presentations/Dutang.pdf>.

This package has currently implementations for state-of-the-art pseudo RNGs for simulations as well as the usual quasi RNGs. There are also some RNG tests. See `?pseudo.randtoolbox` and `?quasi.randtoolbox`.

We recommend first users to take a look at the first part of the vignette. The second part is for people who want a deeper presentation of this topic.

The stable version is available on CRAN <http://cran.r-project.org/web/packages/randtoolbox/index.html>, while the development version is hosted on R-forge <http://r-forge.r-project.org/projects/rmetrics/>.

Details

Package: randtoolbox
 Type: Package
 Version: 1.09
 Date: 2009-10-30
 License: GPL version 2 or later

Author(s)

Christophe Dutang and Petr Savicky

 auxiliary

Auxiliary functions for 'randtoolbox' package.

Description

Stirling numbers of the second kind and permutation of positive integers.

Usage

```
stirling(n)
permut(n)
```

Arguments

n a positive integer.

Details

`stirling` computes stirling numbers of second kind i.e.

$$Stirl_n^k = k * Stirl_{n-1}^k + Stirl_{n-1}^{k-1}$$

with $Stirl_n^1 = Stirl_n^n = 1$. e.g.

- $n = 0$, returns 1
- $n = 1$, returns a vector with 0,1
- $n = 2$, returns a vector with 0,1,1
- $n = 3$, returns a vector with 0,1,3,1
- $n = 4$, returns a vector with 0,1,7,6,1...

Go to wikipedia for more details.

`permut` compute permutation of $1, \dots, n$ and store it in a matrix. e.g.

- $n = 1$, returns a matrix with

$$1$$

- $n = 2$, returns a matrix with

$$\begin{matrix} 1 & 2 \\ 2 & 1 \end{matrix}$$

- $n = 3$ returns a matrix with

```

3  1  2
3  2  1
1  3  2
2  3  1
1  2  3
2  1  3

```

Value

a vector with stirling numbers.

Author(s)

Christophe Dutang.

See Also

[choose](#) for combination numbers.

Examples

```

# should be 1
stirling(0)

# should be 0,1,7,6,1
stirling(4)

```

coll.test

the Collision test

Description

The Collision test for testing random number generators.

Usage

```
coll.test(rand, lenSample = 2^14, nbCell = 2^20, nbSample = 1000, echo = TRUE, ...)
```

Arguments

rand	a function generating random numbers. its first argument must be the 'number of observation' argument as in <code>runif</code> .
lenSample	numeric for the length of generated samples.
nbCell	numeric for the number of cells in which to put random numbers.

`nbSample` numeric for the overall sample number.
`echo` logical to plot detailed results, default TRUE
`...` further arguments to pass to function `rand`

Details

We consider outputs of multiple calls to a random number generator `rand`. Let us denote by n the length of samples (i.e. `lenSample` argument), k the number of cells (i.e. `nbCell` argument) and m the number of samples (i.e. `nbSample` argument).

A collision is defined as when a random number falls in a cell where there are already random numbers. Let us note C the number of collisions

The distribution of collision number C is given by

$$P(C = c) = \prod_{i=0}^{n-c-1} \frac{k-i}{k} \frac{1}{k^c} {}_2S_n^{n-c},$$

where ${}_2S_n^k$ denotes the Stirling number of the second kind and $c = 0, \dots, n-1$.

But we cannot use this formula for large n since the Stirling number need $O(n \log(n))$ time to be computed. We use a Gaussian approximation if $\frac{n}{k} > \frac{1}{32}$ and $n \geq 2^8$, a Poisson approximation if $\frac{n}{k} < \frac{1}{32}$ and the exact formula otherwise.

Finally we compute m samples of random numbers, on which we calculate the number of collisions. Then we are able to compute a chi-squared statistic.

Value

a list with the following components :

`statistic` the value of the chi-squared statistic.

`p.value` the p-value of the test.

`observed` the observed counts.

`expected` the expected counts under the null hypothesis.

`residuals` the Pearson residuals, $(\text{observed} - \text{expected}) / \sqrt{\text{expected}}$.

Author(s)

Christophe Dutang.

References

Planchet F., Jacquemin J. (2003), *L'utilisation de methodes de simulation en assurance*. Bulletin Francais d'Actuariat, vol. 6, 11, 3-69. (available online)

L'Ecuyer P. (2001), *Software for uniform random number generation distinguishing the good and the bad*. Proceedings of the 2001 Winter Simulation Conference. (available online)

L'Ecuyer P. (2007), *Test U01: a C library for empirical testing of random number generators*. ACM Trans. on Mathematical Software 33(4), 22.

See Also

other tests of this package [freq.test](#), [serial.test](#), [poker.test](#), [order.test](#) and [gap.test](#)
[ks.test](#) for the Kolmogorov Smirnov test and [acf](#) for the autocorrelation function.

Examples

```
# (1) poisson approximation
#
coll.test(runif)

# (2) exact distribution
#
coll.test(SFMT, 2^7, 2^10, 10000)
```

freq.test

the Frequency test

Description

The Frequency test for testing random number generators.

Usage

```
freq.test(u, seq = 0:15, echo = TRUE)
```

Arguments

`u` sample of random numbers in]0,1[.
`echo` logical to plot detailed results, default TRUE
`seq` a vector of contiguous integers, default 0:15.

Details

We consider a vector `u`, realisation of i.i.d. uniform random variables U_1, \dots, U_n .

The frequency test works on a serie `seq` of ordered contiguous integers (s_1, \dots, s_d) , where $s_j \in \mathbb{Z}$. From the sample `u`, we compute observed integers as

$$d_i = \lfloor u_i * (s_d + 1) + s_1 \rfloor,$$

(i.e. d_i are uniformly distributed in $\{s_1, \dots, s_d\}$). The expected number of integers equals to j is $m = \frac{1}{s_d - s_1 + 1} \times n$. Finally, the chi-squared statistic is

$$S = \sum_{j=1}^d \frac{(\text{card}(d_i = s_j) - m)^2}{m}.$$

Value

a list with the following components :

`statistic` the value of the chi-squared statistic.

`p.value` the p-value of the test.

`observed` the observed counts.

`expected` the expected counts under the null hypothesis.

`residuals` the Pearson residuals, $(\text{observed} - \text{expected}) / \sqrt{\text{expected}}$.

Author(s)

Christophe Dutang.

References

Planchet F., Jacquemin J. (2003), *L'utilisation de methodes de simulation en assurance*. Bulletin Francais d'Actuariat, vol. 6, 11, 3-69. (available online)

L'Ecuyer P. (2001), *Software for uniform random number generation distinguishing the good and the bad*. Proceedings of the 2001 Winter Simulation Conference. (available online)

L'Ecuyer P. (2007), *Test U01: a C library for empirical testing of random number generators*. ACM Trans. on Mathematical Software 33(4), 22.

See Also

other tests of this package [gap.test](#), [serial.test](#), [poker.test](#), [order.test](#) and [coll.test](#)
[ks.test](#) for the Kolmogorov Smirnov test and [acf](#) for the autocorrelation function.

Examples

```
# (1)
#
freq.test(runif(1000))
print( freq.test( runif(1000000), echo=FALSE) )

# (2)
#
freq.test(runif(1000), 1:4)

freq.test(runif(1000), 10:40)
```

gap.test

the Gap test

Description

The Gap test for testing random number generators.

Usage

```
gap.test(u, lower = 0, upper = 1/2, echo = TRUE)
```

Arguments

u	sample of random numbers in]0,1[.
lower	numeric for the lower bound, default 0.
upper	numeric for the upper bound, default 1/2.
echo	logical to plot detailed results, default TRUE

Details

We consider a vector u , realisation of i.i.d. uniform random variables U_1, \dots, U_n .

The gap test works on the 'gap' variables defined as

$$G_i = \begin{cases} 1 & \text{if } lower \leq U_i \leq upper \\ 0 & \text{otherwise} \end{cases}$$

Let p the probability that G_i equals to one. Then we compute the length of zero gaps and denote by n_j the number of zero gaps of length j . The chi-squared statistic is given by

$$S = \sum_{j=1}^m \frac{(n_j - np_j)^2}{np_j},$$

where p_j stands for the probability the length of zero gaps equals to j ($(1-p)^2 p^j$) and m the max number of lengths (at least $\left\lfloor \frac{\log(10^{-1}) - 2 \log(1-p) - \log(n)}{\log(p)} \right\rfloor$).

Value

a list with the following components :

`statistic` the value of the chi-squared statistic.

`p.value` the p-value of the test.

`observed` the observed counts.

`expected` the expected counts under the null hypothesis.

`residuals` the Pearson residuals, $(\text{observed} - \text{expected}) / \sqrt{\text{expected}}$.

Author(s)

Christophe Dutang.

References

Planchet F., Jacquemin J. (2003), *L'utilisation de methodes de simulation en assurance*. Bulletin Francais d'Actuariat, vol. 6, 11, 3-69. (available online)

L'Ecuyer P. (2001), *Software for uniform random number generation distinguishing the good and the bad*. Proceedings of the 2001 Winter Simulation Conference. (available online)

L'Ecuyer P. (2007), *Test U01: a C library for empirical testing of random number generators*. ACM Trans. on Mathematical Software 33(4), 22.

See Also

other tests of this package [freq.test](#), [serial.test](#), [poker.test](#), [order.test](#) and [coll.test](#)

[ks.test](#) for the Kolmogorov Smirnov test and [acf](#) for the autocorrelation function.

Examples

```
# (1)
#
gap.test(runif(1000))
print( gap.test( runif(1000000), echo=FALSE ) )

# (2)
#
gap.test(runif(1000), 1/3, 2/3)
```

get.primes

Get primes for quasi random number generation

Description

Provides a vector of a specified number of smallest primes from the internal table of the package.

Usage

```
get.primes(n)
```

Arguments

n The required number of primes. Should be at most 100 000.

Details

The package contains an internal table of the smallest 100 000 primes, which may be used in `torus` algorithm.

Value

Vector of $\min(n, 100000)$ smallest primes.

See Also

[torus](#)

Examples

```
p <- get.primes(20)
torus(5, dim=10, prime=p[11:20])
```

`order.test`

the Order test

Description

The Order test for testing random number generators.

Usage

```
order.test(u, d = 3, echo = TRUE)
```

Arguments

<code>u</code>	sample of random numbers in $]0,1[$.
<code>echo</code>	logical to plot detailed results, default TRUE
<code>d</code>	a numeric for the dimension, see details. When necessary we assume that <code>d</code> is a multiple of the length of <code>u</code> .

Details

We consider a vector `u`, realisation of i.i.d. uniform random variables U_1, \dots, U_n .

The Order test works on a sequence of `d`-uplets (x, y, z when `d=3`) of uniform i.i.d. random variables. The triplet is build from the vector `u`. The number of permutation among the components of a triplet is $3! = 6$, i.e. $x < y < z$, $x < z < y$, $y < x < z$, $y < z < x$, $z < x < y$ and $z < y < x$. The Marsaglia test computes the empirical of the different permutations as well as the theoretical one $n/6$ where n is the number of triplets. Finally the chi-squared statistic is

$$S = \sum_{j=0}^6 \frac{(n_j - n/6)^2}{n/6}.$$

Value

a list with the following components :

`statistic` the value of the chi-squared statistic.

`p.value` the p-value of the test.

`observed` the observed counts.

`expected` the expected counts under the null hypothesis.

`residuals` the Pearson residuals, $(\text{observed} - \text{expected}) / \sqrt{\text{expected}}$.

Author(s)

Christophe Dutang.

References

Planchet F., Jacquemin J. (2003), *L'utilisation de methodes de simulation en assurance*. Bulletin Francais d'Actuariat, vol. 6, 11, 3-69. (available online)

L'Ecuyer P. (2001), *Software for uniform random number generation distinguishing the good and the bad*. Proceedings of the 2001 Winter Simulation Conference. (available online)

L'Ecuyer P. (2007), *Test U01: a C library for empirical testing of random number generators*. ACM Trans. on Mathematical Software 33(4), 22.

See Also

other tests of this package [freq.test](#), [serial.test](#), [poker.test](#), [gap.test](#) and [coll.test](#)
[ks.test](#) for the Kolmogorov Smirnov test and [acf](#) for the autocorrelation function.

Examples

```
# (1) mersenne twister vs torus
#
order.test(runif(6000))
order.test(torus(6000))

# (2)
#
order.test(runif(4000), 4)
order.test(torus(4000), 4)

# (3)
#
order.test(runif(5000), 5)
order.test(torus(5000), 5)
```

poker.test

the Poker test

Description

The Poker test for testing random number generators.

Usage

```
poker.test(u , nbcards = 5, echo = TRUE)
```

Arguments

`u` sample of random numbers in]0,1[.

`echo` logical to plot detailed results, default TRUE

`nbcards` a numeric for the number of cards, we assume that the length of `u` is a multiple of `nbcards`.

Details

We consider a vector `u`, realisation of i.i.d. uniform random variables U_1, \dots, U_n .

Let us note k the card number (i.e. `nbcards`). The poker test computes a serie of 'hands' in $\{0, \dots, k-1\}$ from the sample $h_i = \lfloor u_i d \rfloor$ (`u` must have a length dividable by k). Let n_j be the number of 'hands' with (exactly) j different cards. The probability is

$$p_j = \frac{k!}{k^k (k-j)!} * S_k^j * \left(\frac{j}{k}\right)^{k-j},$$

where S_k^j denotes the Stirling numbers of the second kind. Finally the chi-squared statistic is

$$S = \sum_{j=0}^{k-1} \frac{(n_j - np_j/k)^2}{np_j/k}.$$

Value

a list with the following components :

`statistic` the value of the chi-squared statistic.

`p.value` the p-value of the test.

`observed` the observed counts.

`expected` the expected counts under the null hypothesis.

`residuals` the Pearson residuals, (observed - expected) / sqrt(expected).

Author(s)

Christophe Dutang.

References

Planchet F., Jacquemin J. (2003), *L'utilisation de methodes de simulation en assurance*. Bulletin Francais d'Actuariat, vol. 6, 11, 3-69. (available online)

L'Ecuyer P. (2001), *Software for uniform random number generation distinguishing the good and the bad*. Proceedings of the 2001 Winter Simulation Conference. (available online)

L'Ecuyer P. (2007), *Test U01: a C library for empirical testing of random number generators*. ACM Trans. on Mathematical Software 33(4), 22.

See Also

other tests of this package [freq.test](#), [serial.test](#), [gap.test](#), [order.test](#) and [coll.test](#) [ks.test](#) for the Kolmogorov Smirnov test and [acf](#) for the autocorrelation function.

Examples

```
# (1) hands of 5 'cards'
#
poker.test(runif(50000))

# (2) hands of 4 'cards'
#
poker.test(runif(40000), 4)

# (3) hands of 42 'cards'
#
poker.test(runif(420000), 42)
```

pseudo.randtoolbox *Toolbox for pseudo and quasi random number generation*

Description

General linear congruential generators such as Park Miller sequence, generalized feedback shift register such as SF-Mersenne Twister algorithm and WELL generator; and a quasi random generator (pseudo random generators) and the Torus algorithm (quasi random generation).

Usage

```
congruRand(n, dim = 1, mod = 2^31-1, mult = 16807, incr = 0, echo)
SFMT(n, dim = 1, mexp = 19937, usepset = TRUE, withtorus = FALSE, usetime = FALSE)
WELL(n, dim = 1, order = 512, temper = FALSE, version = "a")
knuthTAOCP(n, dim = 1)
setSeed(seed)
```

Arguments

n	number of observations. If length(n) > 1, the length is taken to be the required number.
dim	dimension of observations (must be <=100 000, default 1).
seed	a single value, interpreted as a positive integer for the seed. e.g. append your day, your month and your year of birth.
mod	an integer defining the modulus of the linear congruential generator.
mult	an integer defining the multiplier of the linear congruential generator.
incr	an integer defining the increment of the linear congruential generator.
echo	a logical to plot the seed while computing the sequence.
mexp	an integer for the mersenne exponent of SFMT algorithm. see details
withtorus	a numeric in]0,1] defining the proportion of the torus sequence appended to the SFMT sequence; or a logical equals to FALSE (default).
usepset	a logical to use a set of 12 parameters set for SFMT. default TRUE.
usetime	a logical to use the machine time to start the Torus sequence, default TRUE. if FALSE, the Torus sequence start from the first term.
order	a positive integer for the order of the characteristic polynomial. see details
temper	a logical if you want to do a tempering stage. see details
version	a character either 'a' or 'b'. see details

Details

The currently available generator are given below.

Linear congruential generators: The k th term of a linear congruential generator is defined as

$$u_k = \frac{(a * u_{k-1} + c) \bmod m}{m}$$

where a denotes the multiplier, c the increment and m the modulus, with the constraint $0 \leq a < m$ and $0 \leq c < m$. The default setting is the Park Miller sequence with $a = 16807$, $m = 2^{31} - 1$ and $c = 0$.

SF Mersenne-Twister algorithm: SFMT function implements the SIMD-oriented Fast Mersenne Twister algorithm (cf. <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html>). The SFMT generator has a period of length $2^m - 1$ where m is a Mersenne exponent. In the function SFMT, m is given through `mexp` argument. By default it is 19937 like the "old" MT algorithm. The possible values for the Mersenne exponent are 607, 1279, 2281, 4253, 11213, 19937, 44497, 86243, 132049, 216091.

There are numerous parameters for the SFMT algorithm (see the article for details). By default, we use a different set of parameters (among 32 sets) at *each call* of SFMT (`usepset=TRUE`). The user can use a fixed set of parameters with `usepset=FALSE`. Let us note there is for the moment just *one* set of parameters for 44497, 86243, 132049, 216091 mersenne exponent. Sets of parameters can be found in appendix of the vignette.

The use of different parameter sets is motivated by the following citation of Matsumoto and Saito on this topic :

"Using one same pseudorandom number generator for generating multiple independent streams by changing the initial values may cause a problem (with negligibly small probability). To avoid the problem, using different parameters for each generation is preferred. See Matsumoto M. and Nishimura T. (1998) for detailed information."

All the C code for SFMT algorithm used in this package is the code of M. Matsumoto and M. Saito (cf. <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>), except we add some C code to *interface* with R. Streaming SIMD Extensions 2 (SSE2) operations are not yet supported.

WELL generator: The WELL (which stands for Well Equidistributed Long-period Linear) is in a sentence a generator with better equidistribution than Mersenne Twister algorithm but this gain of quality has to be paid by a slight higher cost of time. See Panneton et al. (2006) for details.

The `order` argument of WELL generator is the order of the characteristic polynomial, which is denoted by k in Paneton F., L'Ecuyer P. and Matsumoto M. (2006). Possible values for `order` are 512, 521, 607, 1024 where no tempering are needed (thus possible). Order can also be 800, 19937, 21071, 23209, 44497 where a tempering stage is possible through the `temper` argument. Furthermore a possible 'b' version of WELL RNGs are possible for the following order 521, 607, 1024, 800, 19937, 23209 with the `version` argument.

All the C code for WELL generator used in this package is the code of P. L'Ecuyer (cf. <http://www.iro.umontreal.ca/~lecuyer/>), except some C code, we add, to *interface* with R.

Knuth TAOCP 2002 (double version): The Knuth-TACOP-2002 is a Fibonacci-lagged generator invented by Knuth(2002), based on the following recurrence.

$$x_n = (x_{n-37} + x_{n-100}) \bmod 2^{30},$$

In R, there is the integer version of this generator.

All the C code for this generator called `RAN_ARRAY` by Knuth is the code of D. Knuth (cf. <http://www-cs-faculty.stanford.edu/~knuth/news02.html#rng>) except some C code, we add, to *interface* with R.

Set the seed: The function `setSeed` is similar to the function `set.seed` in R. It sets the seed to the one given by the user. Do not use a seed with too few ones in its binary representation. Generally, we append our day, our month and our year of birth or append a day, a month and a year. We recall by default with use the machine time to set the seed except for quasi random number generation.

See the pdf vignette for details.

Value

SFMT, WELL, `congruRand` and `knuthTAOCP` generate random variables in $]0,1[$, $[0,1[$ and $[0,1[$ respectively. It returns a $n \times \text{dim}$ matrix, when $\text{dim} > 1$ otherwise a vector of length n .

`setSeed` sets the seed of the `randtoolbox` package (i.e. both for the `knuthTAOCP`, SFMT, WELL and `congruRand` functions).

Author(s)

Christophe Dutang and Petr Savicky

References

Knuth D. (1997), *The Art of Computer Programming V2 Seminumerical Algorithms*, Third Edition, Massachusetts: Addison-Wesley.

Matsumoto M. and Nishimura T. (1998), *Dynamic Creation of Pseudorandom Number Generators*, Monte Carlo and Quasi-Monte Carlo Methods, Springer, pp 56–69. (available online)

Matsumoto M., Saito M. (2008), *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*. (available online)

Paneton F., L'Ecuyer P. and Matsumoto M. (2006), *Improved Long-Period Generators Based on Linear Recurrences Modulo 2*, ACM Transactions on Mathematical Software. (preprint available online)

Park S. K., Miller K. W. (1988), *Random number generators: good ones are hard to find*. Association for Computing Machinery, vol. 31, 10, pp 1192-2001. (available online)

Wikipedia (2008), *a linear congruential generator*.

See Also

[.Random.seed](#) for what is done in R about random number generation and [runifInterface](#) for the `runif` interface.

Examples

```
require(rngWELL)

# (1) the Park Miller sequence
#

# Park Miller sequence, i.e. mod = 2^31-1, mult = 16807, incr=0
# the first 10 seeds used in Park Miller sequence
# 16807          1
# 282475249     2
# 1622650073    3
# 984943658     4
# 1144108930    5
# 470211272     6
# 101027544     7
# 1457850878    8
# 1458777923    9
# 2007237709   10
setSeed(1)
congruRand(10, echo=TRUE)

# the 9998+ th terms
# 925166085     9998
# 1484786315   9999
# 1043618065  10000
# 1589873406  10001
# 2010798668  10002
setSeed(1614852353) #seed for the 9997th term
congruRand(5, echo=TRUE)
```

```
# (2) the SF Mersenne Twister algorithm
SFMT(1000)

#Kolmogorov Smirnov test
#KS statistic should be around 0.037
ks.test(SFMT(1000), punif)

#KS statistic should be around 0.0076
ks.test(SFMT(10000), punif)

#different mersenne exponent with a fixed parameter set
#
SFMT(10, mexp = 607, usepset = FALSE)
SFMT(10, mexp = 1279, usepset = FALSE)
SFMT(10, mexp = 2281, usepset = FALSE)
SFMT(10, mexp = 4253, usepset = FALSE)
SFMT(10, mexp = 11213, usepset = FALSE)
SFMT(10, mexp = 19937, usepset = FALSE)
SFMT(10, mexp = 44497, usepset = FALSE)
SFMT(10, mexp = 86243, usepset = FALSE)
SFMT(10, mexp = 132049, usepset = FALSE)
SFMT(10, mexp = 216091, usepset = FALSE)

#use different sets of parameters [default when possible]
#
for(i in 1:7) print(SFMT(1, mexp = 607))
for(i in 1:7) print(SFMT(1, mexp = 2281))
for(i in 1:7) print(SFMT(1, mexp = 4253))
for(i in 1:7) print(SFMT(1, mexp = 11213))
for(i in 1:7) print(SFMT(1, mexp = 19937))

#use a fixed set and a fixed seed
#should be the same output
setSeed(08082008)
SFMT(1, usepset = FALSE)
setSeed(08082008)
SFMT(1, usepset = FALSE)

# (3) withtorus argument
#

# one third of outputs comes from Torus algorithm
u <- SFMT(1000, with=1/3)
# the third term of the following code is the first term of torus sequence
print(u[666:670] )

# (4) WELL generator
#

# 'basic' calls
# WELL512
```

```
WELL(10, order = 512)
# WELL1024
WELL(10, order = 1024)
# WELL19937
WELL(10, order = 19937)
# WELL44497
WELL(10, order = 44497)
# WELL19937 with tempering
WELL(10, order = 19937, temper = TRUE)
# WELL44497 with tempering
WELL(10, order = 44497, temper = TRUE)

# tempering vs no tempering
setSeed4WELL(08082008)
WELL(10, order =19937)
setSeed4WELL(08082008)
WELL(10, order =19937, temper=TRUE)

# (5) Knuth TAOCP generator
#
knuthTAOCP(10)
knuthTAOCP(10, 2)

# (6) How to set the seed?
# all example is duplicated to ensure setSeed works

# congruRand
setSeed(1302)
congruRand(1)
setSeed(1302)
congruRand(1)
# SFMT
setSeed(1302)
SFMT(1, usepset=FALSE)
setSeed(1302)
SFMT(1, usepset=FALSE)
# BEWARE if you do not set usepset to FALSE
setSeed(1302)
SFMT(1)
setSeed(1302)
SFMT(1)
# WELL
setSeed(1302)
WELL(1)
setSeed(1302)
WELL(1)
# Knuth TAOCP
setSeed(1302)
knuthTAOCP(1)
setSeed(1302)
knuthTAOCP(1)
```

```

# (7) computation times on my macbook, mean of 1000 runs
#

## Not run:
# algorithm time in seconds for n=10^6
# classical Mersenne Twister    0.066
# SF Mersenne Twister          0.044
# WELL generator 0.065
# Knuth TAOCP 0.046
# Park Miller                   0.108
n <- 1e+06
mean( replicate( 1000, system.time( runif(n), gcFirst=TRUE)[3]) )
mean( replicate( 1000, system.time( SFMT(n), gcFirst=TRUE)[3]) )
mean( replicate( 1000, system.time( WELL(n), gcFirst=TRUE)[3]) )
mean( replicate( 1000, system.time( knuthTAOCP(n), gcFirst=TRUE)[3]) )
mean( replicate( 1000, system.time( congruRand(n), gcFirst=TRUE)[3]) )

## End(Not run)

```

quasi.randtoolbox *Toolbox for quasi random number generation*

Description

the Torus algorithm, the Sobol and Halton sequences.

Usage

```

halton(n, dim = 1, init = TRUE, normal = FALSE, usetime = FALSE)
sobol(n, dim = 1, init = TRUE, scrambling = 0, seed = 4711, normal = FALSE)
torus(n, dim = 1, prime, init = TRUE, mixed = FALSE, usetime = FALSE, normal=FALSE)

```

Arguments

n	number of observations. If length(n) > 1, the length is taken to be the required number.
dim	dimension of observations default 1.
init	a logical, if TRUE the sequence is initialized and restarts, otherwise not. By default TRUE.
normal	a logical if normal deviates are needed, default FALSE
scrambling	an integer value, if 1, 2 or 3 the sequence is scrambled otherwise not. If 1, Owen type type of scrambling is applied, if 2, Faure-Tezuka type of scrambling, is applied, and if 3, both Owen+Faure-Tezuka type of scrambling is applied. By default 0.

seed	an integer value, the random seed for initialization of the scrambling process. By default 4711. On effective if <code>scrambling>0</code> .
prime	a single prime number or a vector of prime numbers to be used in the Torus sequence. (optional argument).
mixed	a logical to use the mixed Torus algorithm, default FALSE.
usetime	a logical to use the machine time to start the Torus sequence, default TRUE. if FALSE, the Torus sequence start from the first term.

Details

The currently available generator are given below.

Torus algorithm: The k th term of the Torus algorithm in d dimension is given by

$$u_k = (\text{frac}(k\sqrt{p_1}), \dots, \text{frac}(k\sqrt{p_d}))$$

where p_i denotes the i th prime number, frac the fractional part (i.e. $\text{frac}(x) = x - \text{floor}(x)$). We use the 100 000 first prime numbers from <http://primes.utm.edu/>, thus the dimension is limited to 100 000. If the user supplies prime numbers through the argument `prime`, we do NOT check for primality and we cast numerics to integers, (i.e. `prime=7.1234` will be cast to `prime=7` before computing Torus sequence). The Torus sequence starts from $k = 1$ when initialized with `init = TRUE` and no not depending on machine time `usetime = FALSE`. This is the default. When `init = FALSE`, the sequence is not initialized (to 1) and starts from the last term. We can also use the machine time to start the sequence with `usetime = TRUE`, which overrides `init`.

scrambled Sobol sequences Calculates a matrix of uniform and normal deviated Sobol low discrepancy numbers. Optional scrambling of the sequence can be selected. The Sobol sequence restarts and is initialized when `init = TRUE` and otherwise not.

Halton sequences Calculates a matrix of uniform or normal deviated halton low discrepancy numbers. Let us note that Halton sequence in dimension is the Van Der Corput sequence. The Halton sequence starts from $k = 1$ when initialized with `init = TRUE` and no not depending on machine time `usetime = FALSE`. This is the default. When `init = FALSE`, the sequence is not initialized (to 1) and starts from the last term. We can also use the machine time to start the sequence with `usetime = TRUE`, which overrides `init`.

See the pdf vignette for details.

Value

`torus`, `halton` and `sobol` generates random variables in $]0,1[$. It returns a $n \times \text{dim}$ matrix, when `dim>1` otherwise a vector of length `n`.

Author(s)

Christophe Dutang and Diethelm Wuertz

References

Bratley P., Fox B.L. (1988); *Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software 14, 88–100.

Joe S., Kuo F.Y. (1998); *Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator*.

Planchet F., Jacquemin J. (2003), *L'utilisation de methodes de simulation en assurance*. Bulletin Francais d'Actuariat, vol. 6, 11, 3-69. (available online)

See Also

[pseudo.randtoolbox](#) for pseudo random number generation, [.Random.seed](#) for what is done in R about random number generation.

Examples

```
# (1) the Torus algorithm
#
torus(100)

# example of setting the seed
setSeed(1)
torus(5)
setSeed(6)
torus(5)
#the same
setSeed(1)
torus(10)

#no use of the machine time
torus(10, use=FALSE)

#Kolmogorov Smirnov test
#KS statistic should be around 0.0019
ks.test(torus(1000), punif)

#KS statistic should be around 0.0003
ks.test(torus(10000), punif)

#the mixed Torus sequence
torus(10, mix=TRUE)
par(mfrow = c(1,2))
acf(torus(10^6))
acf(torus(10^6, mix=TRUE))

#usage of the init argument
torus(5)
torus(5, init=FALSE)

#should be equal to the combination of the two
#previous call
torus(10)
```

```
# (2) Halton sequences
#

# uniform variate
halton(n = 10, dim = 5)

# normal variate
halton(n = 10, dim = 5, normal = TRUE)

#usage of the init argument
halton(5)
halton(5, init=FALSE)

#should be equal to the combination of the two
#previous call
halton(10)

# some plots
par(mfrow = c(2, 2), cex = 0.75)
hist(halton(n = 5000, dim = 1), main = "Uniform Halton",
     xlab = "x", col = "steelblue3", border = "white")

hist(halton(n = 5000, dim = 1, norm = TRUE), main = "Normal Halton",
     xlab = "x", col = "steelblue3", border = "white")

# (3) Sobol sequences
#

# uniform variate
sobol(n = 10, dim = 5, scrambling = 3)

# normal variate
sobol(n = 10, dim = 5, scrambling = 3, normal = TRUE)

# some plots
hist(sobol(5000, 1, scrambling = 2), main = "Uniform Sobol",
     xlab = "x", col = "steelblue3", border = "white")

hist(sobol(5000, 1, scrambling = 2, normal = TRUE), main = "Normal Sobol",
     xlab = "x", col = "steelblue3", border = "white")

#usage of the init argument
sobol(5)
sobol(5, init=FALSE)

#should be equal to the combination of the two
#previous call
sobol(10)

# (4) computation times on my macbook, mean of 1000 runs
#
```

```
## Not run:
# algorithm time in seconds for n=10^6
# Torus algo 0.058
# mixed Torus algo          0.087
# Halton sequence 0.878
# Sobol sequence 0.214
n <- 1e+06
mean( replicate( 1000, system.time( torus(n), gcFirst=TRUE)[3]) )
mean( replicate( 1000, system.time( torus(n, mixed=TRUE), gcFirst=T)[3]) )
mean( replicate( 1000, system.time( halton(n), gcFirst=TRUE)[3]) )
mean( replicate( 1000, system.time( sobol(n), gcFirst=TRUE)[3]) )

## End(Not run)
```

runifInterface

Functions for using runif() and rnorm() with randtoolbox generators

Description

These functions allow to set some of the random number generators from randtoolbox package so that it can be used in the standard R functions, which use random numbers, for example `runif()`, `rnorm()`, `sample()` and also `set.seed()`.

Usage

```
set.generator <- function(name=c("congruRand", "WELL", "default"),
  parameters=NULL, seed=NULL, ..., only.dsc=FALSE)
  put.description(description)
  get.description()
```

Arguments

name	a character string for the RNG name.
parameters	a numeric or character vector describing a RNG from the family specified by the name parameter.
seed	a number to be used as a seed
...	arguments describing the components of the vector parameters, if argument parameters is NULL.
only.dsc	a logical. If TRUE the description of a RNG is created, but the generator is not initialized.
description	a list describing a generator as created by <code>set.generator()</code> or <code>get.description()</code>

Details

Random number generators provided by R extension packages are set using `RNGkind("user-supplied")`. The package **randtoolbox** assumes that this function is not called by the user directly. Instead, it is called from the functions `set.generator()` and `put.description()` used for setting some of a larger collection of the supported generators.

Random number generators in **randtoolbox** are represented at the R level by a list containing mandatory components `name`, `parameters`, `state` and possibly an optional component `authors`. The function `set.generator()` internally creates this list from the user supplied information and then runs `put.description()` on this list in order to really initialize the generator for the functions `runif()` and `set.seed()`. If `set.generator()` is called with the parameter `only.dsc=TRUE`, then the generator is not initialized and only its description is created. If the generator is initialized, then the function `get.description()` may be used to get the actual state of the generator, which may be stored and used later in `put.description()` to continue the sequence of the random numbers from the point, where `get.description()` was called. This may be used, for example, to alternate between the streams of random numbers generated by different generators.

Value

`set.generator()` with the parameter `only.dsc=TRUE` and `get.description()` return the list describing a generator. `put.description()` with the parameter `only.dsc=TRUE` (the default) and `put.description()` return `NULL`.

Author(s)

Petr Savicky and Christophe Dutang

See Also

`RNGkind`

Examples

```
RNGkind() # [1] "Mersenne-Twister" "Inversion"

#parameters for Park Miller congruential generator
paramParkMiller <- c(mod=2^31-1, mult=16807, incr=0)
set.generator(name="congruRand", parameters=paramParkMiller, seed=1)

RNGkind() # [1] "user-supplied" "Inversion"

#description of the RNG set by set.generator(), i.e. Park Miller
print(ParkMiller <- get.description())

#generate 10 random points from the Park-Miller sequence
x1 <- runif(10)

#the seed has changed
get.description()
```

```

# the Knuth Lewis RNG
paramKnuthLewis <- c(mod="4294967296", mult="1664525", incr="1013904223")
set.generator(name="congruRand", parameters= paramKnuthLewis, seed=1)

#description of the current RNG, i.e. Knuth Lewis
KLwithseed1 <- get.description()

x2 <- runif(10)

#reinitiate the RNG setting
put.description(ParkMiller)

#the same as x1
x1 == runif(10)

#set WELL RNGs
set.generator("WELL", seed=12345, order=1024, version="a")
get.description()

#get back to the original R setting
set.generator("default")
RNGkind()

```

serial.test

the Serial test

Description

The Serial test for testing random number generators.

Usage

```
serial.test(u , d = 8, echo = TRUE)
```

Arguments

u	sample of random numbers in]0,1[.
echo	logical to plot detailed results, default TRUE
d	a numeric for the dimension, see details. When necessary we assume that d is a multiple of the length of u.

Details

We consider a vector u , realisation of i.i.d. uniform random variables U_1, \dots, U_n .

The serial test computes a serie of integer pairs (p_i, p_{i+1}) from the sample u with $p_i = \lfloor u_i d \rfloor$ (u must have an even length). Let n_j be the number of pairs such that $j = p_i \times d + p_{i+1}$. If $d=2$, we

count the number of pairs equals to 00, 01, 10 and 11. Since all the combination of two elements in $\{0, \dots, d-1\}$ are equiprobable, the chi-squared statistic is

$$S = \sum_{j=0}^{d-1} \frac{n_j - n/(2d^2)}{n/(2d^2)}^2.$$

Value

a list with the following components :

`statistic` the value of the chi-squared statistic.

`p.value` the p-value of the test.

`observed` the observed counts.

`expected` the expected counts under the null hypothesis.

`residuals` the Pearson residuals, $(\text{observed} - \text{expected}) / \sqrt{\text{expected}}$.

Author(s)

Christophe Dutang.

References

Planchet F., Jacquemin J. (2003), *L'utilisation de methodes de simulation en assurance*. Bulletin Francais d'Actuariat, vol. 6, 11, 3-69. (available online)

L'Ecuyer P. (2001), *Software for uniform random number generation distinguishing the good and the bad*. Proceedings of the 2001 Winter Simulation Conference. (available online)

L'Ecuyer P. (2007), *Test U01: a C library for empirical testing of random number generators*. ACM Trans. on Mathematical Software 33(4), 22.

See Also

other tests of this package [freq.test](#), [gap.test](#), [poker.test](#), [order.test](#) and [coll.test](#) [ks.test](#) for the Kolmogorov Smirnov test and [acf](#) for the autocorrelation function.

Examples

```
# (1)
#
serial.test(runif(1000))
print( serial.test( runif(1000000), d=2, e=FALSE) )

# (2)
#
serial.test(runif(5000), 5)
```

Index

*Topic **distribution**

auxiliary, 2
pseudo.randtoolbox, 13
quasi.randtoolbox, 19
runifInterface, 22

*Topic **htest**

coll.test, 4
freq.test, 6
gap.test, 7
order.test, 10
poker.test, 11
serial.test, 25

*Topic **package**

randtoolbox-package, 2
.Random.seed, 16, 20

acf, 5, 7, 8, 11, 12, 26

auxiliary, 2

choose, 4

coll.test, 4, 7, 8, 11, 12, 26

congruRand(pseudo.randtoolbox),
13

freq.test, 5, 6, 8, 11, 12, 26

gap.test, 5, 7, 7, 11, 12, 26

get.description(runifInterface),
22

get.primes, 9

halton(quasi.randtoolbox), 19

knuthTAOCP(pseudo.randtoolbox),
13

ks.test, 5, 7, 8, 11, 12, 26

order.test, 5, 7, 8, 10, 12, 26

permut(auxiliary), 2

poker.test, 5, 7, 8, 11, 11, 26

pseudo.randtoolbox, 13, 20

put.description(runifInterface),
22

quasi.randtoolbox, 19

randtoolbox

(randtoolbox-package), 2

randtoolbox-package, 2

runif.halton(quasi.randtoolbox),
19

runif.sobol(quasi.randtoolbox),
19

runifInterface, 16, 22

serial.test, 5, 7, 8, 11, 12, 25

set.generator(runifInterface), 22

setSeed(pseudo.randtoolbox), 13

SFMT(pseudo.randtoolbox), 13

sobol(quasi.randtoolbox), 19

stirling(auxiliary), 2

torus, 9

torus(quasi.randtoolbox), 19

WELL(pseudo.randtoolbox), 13