Package 'randtoolbox'

October 30, 2021

Type Package

Title Toolbox for Pseudo and Quasi Random Number Generation and Random Generator Tests

Version 1.31.1

Author R port by Yohan Chalabi, Christophe Dutang, Petr Savicky and Diethelm Wuertz with some underlying C codes of (i) the SFMT algorithm from M. Matsumoto and M. Saito, (ii) the Knuth-TAOCP RNG from D. Knuth.

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

Description Provides (1) pseudo random generators - general linear congruential generators, multiple recursive generators and generalized feedback shift register (SF-Mersenne Twister algorithm and WELL generators); (2) quasi random generators - the Torus algorithm, the Sobol sequence, the Halton sequence (including the Van der Corput sequence) and (3) some generator tests - the gap test, the serial test, the poker test.
See e.g. Gentle (2003) <doi:10.1007/b97336>. The package can be provided without the rngWELL dependency on demand.
Take a look at the Distribution task view of types and tests of random number generators.
Version in Memoriam of Diethelm and Barbara Wuertz.

Depends rngWELL (>= 0.10-1)

License BSD_3_clause + file LICENSE

NeedsCompilation yes

Repository CRAN

Date/Publication 2021-10-30 04:30:02 UTC

R topics documented:

`randtoolbox-package` *General remarks on 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 at the Rmetrics workshop in 2009 in Meielisalp, Switzerland.

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

We recommend first users to take a look at the vignette 'Quick introduction of randtoolbox', whereas advanced users whould have a look at the vigentte 'A note on random number generation'.

The stable version is available on CRAN `https://CRAN.R-project.org/package=randtoolbox`, while the development version is hosted on R-forge in the rmetrics project at `https://r-forge.r-project.org/projects/rmetrics/`.

Details

Package:	randtoolbox
Type:	Package
Version:	1.31.0
Date:	2021
License:	BSD_3_clause + file LICENSE

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 $1, ..., n$ 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

$$\begin{matrix} 3 & 1 & 2 \\ 3 & 2 & 1 \end{matrix}$$

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, segments = 2^10, tdim = 2,
nbSample = 1000, echo = TRUE, ...)
```

Arguments

`rand`	a function generating random numbers. its first argument must be the 'number of observation' argument as in `runif`.
`lenSample`	numeric for the length of generated samples.
`segments`	numeric for the number of segments to which the interval `[0,1]` is split.
`tdim`	numeric for the length of the disjoint t-tuples.
`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 $c = 0, \ldots, n-1$.

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, (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 `coll.test.sparse`, `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, 2^7, 2^10, 1, 100)

# (2) exact distribution
#
coll.test(SFMT, 2^7, 2^10, 1, 100)
```

`coll.test.sparse` *the Collision test*

Description

The Collision test for testing random number generators.

Usage

```
coll.test.sparse(rand, lenSample = 2^14, segments = 2^10, tdim = 2,
nbSample = 10, ...)
```

Arguments

`rand`	a function generating random numbers. its first argument must be the 'number of observation' argument as in `runif`.
`lenSample`	numeric for the length of generated samples.
`segments`	numeric for the number of segments to which the interval `[0,1]` is split.
`tdim`	numeric for the length of the disjoint t-tuples.
`nbSample`	numeric for the number of repetitions of the test.
`...`	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).

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 $c = 0, \ldots, n-1$.

This formula cannot be used for large n since the Stirling number need $O(n \log(n))$ time to be computed. We use a Poisson approximation if $\frac{n}{k} < \frac{1}{32}$ and the exact formula otherwise.

The test is repeated `nbSample` times and the result of each repetition forms a row in the output table.

Value

A data frame with `nbSample` rows and the following columns.

`observed` the observed counts.

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

Author(s)

Christophe Dutang, Petr Savicky.

References

P. L'Ecuyer, R. Simard, S. Wegenkittl, Sparse serial tests of uniformity for random number generators. SIAM Journal on Scientific Computing, 24, 2 (2002), 652-668.

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 `coll.test`, `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.sparse(runif)

# (2) exact distribution
#
coll.test.sparse(SFMT, lenSample=2^7, segments=2^5, tdim=2, nbSample=10)

## Not run:
#A generator with too uniform distribution (too small number of collisions)
#produces p-values close to 1
set.generator(name="congruRand", mod="2147483647", mult="742938285", incr="0", seed=1)
coll.test.sparse(runif, lenSample=300000, segments=50000, tdim=2)

#Park-Miller generator has too many collisions and produces small p-values
set.generator(name="congruRand", mod="2147483647", mult="16807", incr="0", seed=1)
coll.test.sparse(runif, lenSample=300000, segments=50000, tdim=2)

## End(Not run)
```

`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 $U_1, \ldots, U_n$.

The frequency test works on a serie `seq` of ordered contiguous integers $(s_1, \ldots, s_d)$, 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 uniformely distributed in $\{s_1, \ldots, s_d\}$). 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

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

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, (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 `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(10000), 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 $U_1, \ldots, U_n$.

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, (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`, `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])
```

getWELLState — *Get the state of a WELL generator implemented in randtoolbox package.*

Description

Get the state of a WELL generator implemented in randtoolbox package to be used in function `rngWELLScriptR()`.

Usage

```
getWELLState()
```

Value

A 0,1-matrix, whose columns represent 32-bit integers.

See Also

`rngWELLScriptR`

order.test — *the Order test*

Description

The Order test for testing random number generators.

Usage

```
order.test(u, d = 3, 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 $U_1, \ldots, U_n$.

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, (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`, `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 , nbcard = 5, echo = TRUE)
```

Arguments

u	sample of random numbers in]0,1[.
echo	logical to plot detailed results, default TRUE
nbcard	a numeric for the number of cards, we assume that the length of u is a multiple of nbcard.

Details

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

Let us note k the card number (i.e. nbcard). The poker test computes a serie of 'hands' in $\{0, \ldots, k-1\}$ 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

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

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)
```

pseudoRNG *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.

The list of supported generators consists of generators available via direct functions and generators available via `set.generator()` and `runif()` interface. Most of the generators belong to both these groups, but some generators are only available directly (SFMT) and some only via `runif()` interface (Mersenne Twister 2002). This help page describes the list of all the supported generators and the functions for the direct access to those, which are available in this way. See `set.generator()` for the generators available via `runif()` interface.

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 kth 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 $0 <= a < m$ and $0 <= c < m$. The default setting is the Park Miller sequence with $a = 16807$, $m = 2^31 - 1$ and $c = 0$.

Knuth TAOCP 2002 (double version): The Knuth-TACOP-2002 is a Fibonnaci-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. https://www-cs-faculty.stanford.edu/~uno/programs.html) except some C code, we add, to *interface* with R.

Mersenne Twister 2002 generator: The generator suggested by Makoto Matsumoto and Takuji Nishimura with the improved initialization from 2002. See http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html for more information on the generator itself. This generator is available only via `set.generator()` and `runif()` interface. Mersenne Twister generator used in base R is the same generator (the recurrence), but with a different initialization and the output transformation. The implementation included in randtoolbox allows to generate the same random numbers as in Matlab, see examples in `set.generator()`.

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 pesudorandom 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 diffrent parameters for each generation is prefered. 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.

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.

Set the generator: Some of the generators are available using `runif()` interface. See `set.generator()` for more information.

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 $nxdim$ matrix, when `dim`>1 otherwise a vector of length `n`.

`congruRand` may raise an error code if parameters are not correctly specified: `-1` if the multiplier is zero; `-2` if the multiplier is greater or equal than the modulus; `-3` if the increment is greater or equal than the modulus; `-4` if the multiplier times the modulus minus 1 is greater than `2^64-1` minus the increment; `-5` if the seed is greater or equal than the modulus.

`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 2017 macbook (my 2007 macbook), mean of 1000 runs
#

## Not run:
# algorithm time in seconds for n=10^6
# classical Mersenne Twister  0.028155 (0.066)
# SF Mersenne Twister         0.008223 (0.044)
# WELL generator       0.006407 (0.065)
# Knuth TAOCP          0.002923 (0.046)
# Park Miller          0.015635 (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)
```

---

quasiRNG *Toolbox for quasi random number generation*

Description

the Torus algorithm, the Sobol and Halton sequences.

Usage

```
torus(n, dim = 1, prime, init = TRUE, mixed = FALSE, usetime = FALSE,
                  normal = FALSE, mexp = 19937, start = 1)
sobol(n, dim = 1, init = TRUE, scrambling = 0, seed = NULL, normal = FALSE,
                   mixed = FALSE, method = "Fortran", mexp = 19937, start = 1,
                   maxit = 10)
halton(n, dim = 1, init = TRUE, normal = FALSE, usetime = FALSE,
                    mixed = FALSE, method = "C", mexp = 19937, start = 1)
```

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 to the start value, 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 scrambling=1, Owen type type of scrambling is applied, if scrambling=2, Faure-Tezuka type of scrambling, is applied, and if scrambling=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 (only for sobol with scrambling>0). If NULL, set to 4711.
prime	a single prime number or a vector of prime numbers to be used in the Torus sequence. (optional argument).
mixed	a logical to combine the QMC algorithm with the SFMT algorithm, default FALSE.

usetime	a logical to use the machine time to start the Torus sequence, default FALSE, i.e. when usetime=FALSE the Torus sequence start from the first term. usetime is only used when mixed=FALSE.
method	a character string either "C" or "Fortran". Note that mixed=TRUE is only available when method="C".
mexp	an integer for the Mersenne exponent of SFMT algorithm, only used when mixed=TRUE.
start	an integer to initiliaze the sequence, default to 1, only used when init=TRUE.
maxit	a positive integer used to control inner loops both for generating randomized seed and for controlling outputs (when needed).

Details

The currently available generator are given below. Whatever the sequence, when normal=TRUE, outputs are transformed with the quantile of the standard normal distribution qnorm. If init=TRUE, the default, unscrambled and unmixed-SFMT quasi-random sequences start from start. If start != 0 and normal=FALSE, we suggest to use 0 as recommended by Owen (2020). One must handle the starting value (0) correctly if a quantile function of a not-lower-bounded distribution is used.

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

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

where p_i denotes the ith prime number, $frac$ the fractional part (i.e. $frac(x) = x - floor(x)$). We use the 100 000 first prime numbers from https://primes.utm.edu/, thus the dimension is limited to 100 000. If the user supplys 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 so 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 or a randomized when mixed = TRUE.

(scrambled) Sobol sequences Computes uniform Sobol low discrepancy numbers. The sequence starts from $k = 1$ when initialized with init = TRUE (default). When scrambling > 0, a scrambling is performed or when mixed = TRUE, a randomized seed is performed. If some number of Sobol sequences are generated outside [0,1) with scrambling, the seed is randomized until we obtain all numbers in [0,1). One version of Sobol sequences is available the current version in Fortran (method = "Fortran") since method = "C" is under development.

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 (default) and no not depending on machine time usetime = FALSE. 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. Two versions of Halton sequences are available the historical version in Fortran (method = "Fortran") and the new version in C (method = "C"). If method = "C", mixed argument can be used to randomized the Halton sequences.

See the pdf vignette for details.

Value

`torus`, `halton` and `sobol` generates random variables in [0,1). It returns a $nx dim$ 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 Quaisrandom Seqence Generator*.

Owen A.B. (2020), *On dropping the first Sobol' point*, https://arxiv.org/abs/2008.08051.

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

`pseudoRNG` 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, mixed=TRUE)
```

```
## Not run:
  par(mfrow = c(1,2))
  acf(torus(10^6))
  acf(torus(10^6, mixed=TRUE))

## End(Not run)

#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 = 500, dim = 1), main = "Uniform Halton",
  xlab = "x", col = "steelblue3", border = "white")

hist(halton(n = 500, 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(500, 1, scrambling = 2), main = "Uniform Sobol",
  xlab = "x", col = "steelblue3", border = "white")
```

```
hist(sobol(500, 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 2017 macbook (my 2007 macbook), mean of 1000 runs
#

## Not run:
# algorithm time in seconds for n=10^6
# Torus algo   0.012 (0.058)
# mixed Torus algo      0.018 (0.087)
# Halton sequence 0.180 (0.878)
# Sobol sequence 0.027 (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)
```

`rngWELLScriptR` *An implementation of the recurrence of WELL generators in R language*

Description

An implementation of the recurrence of WELL generators in R language for testing purposes. It is too slow to be used for random number generation.

Usage

```
rngWELLScriptR(n, s, generator, includeState=FALSE)
```

Arguments

`n`	Integer. The length of the output sequence.
`s`	An 0,1-matrix representing the state of the required WELL generator as obtained by `getWELLState()`.
`generator`	Character string. Name of the generator from the list `"512a"`,`"521a"`,`"521b"`,`"607a"`,`"607b"`,`"800a"`,

includeState Logical. Controls, whether the output should contain the final internal state additionally to the numerical output or not.

Value

If includeState=FALSE, a numeric vector of length n containing the numerical output of the generator. If includeState=TRUE, a list with components x (the numerical output) and state (the final internal state of the generator).

See Also

getWELLState

Examples

```
set.generator("WELL", order=512, version="a", seed=123456)
s <- getWELLState()
x <- runif(500)
y <- rngWELLScriptR(500, s, "512a")
all(x == y)
# [1] TRUE
```

---

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 to be used instead of the standard generator in the functions, which use random numbers, for example runif(), rnorm(), sample() and set.seed().

Usage

```
set.generator(name=c("WELL", "MersenneTwister", "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 specific RNG from the family specified by the name.
seed	A number, whose value is an integer between 0 and 2^32 -1, to be used as a seed.

`...`	Arguments describing named components of the vector of parameters, if argument `parameters` is missing or `NULL`.
`only.dsc`	Logical. If `TRUE`, a description of the specified RNG is returned and the generator is not initialized.
`description`	A list describing a specific RNG as created by `set.generator(,only.dsc=TRUE)` or `get.description()`.

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 and is called from the functions `set.generator()` and `put.description()`.

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, which initializes the generator. 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 in an R object 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 to generate several independent streams of random numbers generated by different generators.

The component `parameters` is a character or a numeric vector, whose structure is different for different types of the generators. This vector may be passed to `set.generator()`, if it is prepared before its call, however, it is also possible to pass its named components via the `...` parametr of `set.generator()` and the vector `parameters` is created internally. If the vector `parameters` is not supplied and the arguments in `...` are not sufficient to create it, an error message is produced.

Linear congruential generators: Currently disabled.

Parameters for the linear congruential generators (`name="congruRand"`) are integers represented either as a character or a numeric vector. The components are

mod The modulus.

mult The multiplier.

incr The increment.

WELL generators: Parameters for the WELL generators is a character vector with components

order The number of bits in the internal state. Possible values are 512, 521, 607, 800, 1024, 19937, 21701, 23209, 44497.

version The version letter "a", "b", or "c" to be appended to the order.

The concatenation of `order` and `version` should belong to `"512a","521a","521b","607a","607b","800a","800b","1`
When order and version are specified in `...` parametr of `set.generator()`, then the parameter `order` is optional and if missing, it is assumed that the parameter `version` contains also the number of bits in the internal state and the combination belongs to the list above.

Mersenne Twister 2002 generator: Parameters for the Mersenne Twister 2002 is a character vector with components

initialization Either "init2002" or "array2002". The initialization to be used.

resolution Either 53 or 32. The number of random bits in each number.

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` and `.Random.seed` for random number generation in R.

Examples

```
#set WELL19937a
set.generator("WELL", version="19937a", seed=12345)
runif(5)

#Store the current state  and generate 10 random numbers
storedState <- get.description()
x <- runif(10)

## Not run:
#Park Miller congruential generator
set.generator(name="congruRand", mod=2^31-1, mult=16807, incr=0, seed=12345)
runif(5)
setSeed(12345)
congruRand(5, dim=1, mod=2^31-1, mult=16807, incr=0)

# the Knuth Lewis RNG
set.generator(name="congruRand", mod="4294967296", mult="1664525", incr="1013904223", seed=1)
runif(5)
setSeed(1)
congruRand(5, dim=1, mod=4294967296, mult=1664525, incr=1013904223)

## End(Not run)

#Restore the generator from storedState and regenerate the same numbers
put.description(storedState)
x == runif(10)

# generate the same random numbers as in Matlab
set.generator("MersenneTwister", initialization="init2002", resolution=53, seed=12345)
runif(5)
# [1] 0.9296161 0.3163756 0.1839188 0.2045603 0.5677250
# Matlab commands rand('twister', 12345); rand(1, 5) generate the same numbers,
# which in short format are   0.9296    0.3164    0.1839    0.2046    0.5677

#Restore 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 $U_1, \ldots, U_n$.

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 $\{0, \ldots, d-1\}$ are equiprobable, the chi-squared statistic is

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

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, (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`, `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)
```

`soboltestfunctions` *Some test functions for Sobol sequences*

Description

Some test functions for Sobol sequences

Usage

```
sobol.directions(polycoef, deg, prevmj, nbpoint, echo=FALSE)
sobol.basic(n, polycoef, deg, prevmj, echo=FALSE, output=c("real", "integer"))
sobol.R(n, d, echo=FALSE)

int2bit(x)
bit2int(x)
bit2unitreal(x)
```

Arguments

n	number of observations.
d	dimension of observations.
polycoef	binary coefficients of the primitive polynomial.
deg	degree of the primitive polynomial.
prevmj	matrix where each column is the binary representation of integers m_j
nbpoint	number of additionnal integers m_j.
echo	a logical to show some traces.
output	a character string either "real" or "integer" to specify whether the radical inverse function is computed output="real" or not output="integer".
x	an integer to convert in base 2 or its binary representation

Details

sobol.directions computes the direction numbers used when computing Sobol sequences.

sobol.basic compute the Sobol sequence in one dimension according to a primitive polynomial and specified integers m_j.

int2bit computes the binary representation of the integer part of a real, bit2int computes an integer from its binary representation. bit2unitreal computes the radical inverse function in base 2.

Value

a vector of length n for sobol.basic or a matrix for sobol.directions.

Author(s)

Christophe Dutang

References

Glasserman P., (2003); *Monte Carlo Methods in Financial Engineering*, Springer.

See Also

quasiRNG for quasi random number generation.

Examples

```
#page 306 of Glassermann
p13 <- int2bit(13)
prevterm <- sapply(c(1,3,3), int2bit)
colnames(prevterm) <- apply(prevterm, 2, bit2int)
sobol.directions(p13, 3, prevterm, 2, echo=FALSE)


sobol.basic(15, p13, 3, c(1,3,3), echo=FALSE)
```

```
sobol.basic(15, p13, 3, c(1,3,3), echo=FALSE, output="i")


#page 307 of Glassermann
trueval <- c(16, 24,  8, 12, 28, 20,  4, 30, 14,
6, 22, 18,  2, 10, 26,  5, 21, 29, 13,  9, 25, 17,
1, 27, 11,  3, 19, 23,  7, 15, 31)/32
cbind(sobol.basic(31, p13, 3, prevterm, echo=FALSE, output="r") , trueval)
```

Index