A note on random numbers

Christophe Dutang
ENSIMAG, Grenoble INP

June 2008

*"Nothing in Nature is random...
a thing appears random only through
the incompleteness of our knowledge."*
Spinoza, Ethics I[1].

1 Introduction

The simulation of random variables has a big number of applications from insurance to biology, and from computer science to finance. Actually the only things that are truly random are the measurement of physical phenomena such as thermal noises of semiconductor chips or radioactive sources[2].

Thus the only way to simulate some randomness on computers are carried out by deterministic algorithms. Excluding true randomness[3], there are two kinds random generation: pseudo and quasi random number generators.

The package **randtoolbox** provides R functions for pseudo and quasi random number generations, as well as statistical tests to quantify the quality of random numbers generated.

[1]quote taken from Niederreiter (1978).

[2]for more details go to `http://www.random.org/randomness/`.

[3]For true random number generation on R, use the **random** package of Eddelbuettel (2007).

2 Brief overview of random generation algoritms

In this section, we present first the pseudo random number generation and second the quasi random number generation. By "random numbers", we mean random variates of the uniform $\mathcal{U}(0,1)$ distribution. More complex distributions can be generated with uniform variates and rejection/inversion algorithms. Pseudo random number generation aims to seem random whereas quasi random number generation aims to be deterministic.

Those familiars with algorithms such as linear congruential generation, Mersenne-Twister type algorithms, and low discrepancy sequences should go directly to the next section.

2.1 Pseudo random generation

At the beginning of the nineties, there was no state-of-the-art algorithms to generate pseudo random numbers. And despite this fact, most users thought the `rand` function they used was good, because of a short period and a term to term dependence. But in 1998, Japenese mathematicians Matsumoto and Nishimura invents the first algorithm whose period ($2^{19937}-1$) exceeds the number of electron spin changes since the creation of the Universe (10^{6000} against 10^{120}).

As described in L'Ecuyer (1990), a (pseudo) random number generator (RNG) is defined by a structure (S, μ, f, U, g) where

- S a finite set of *states*,
- μ a probability distribution on S, called the *initial distribution*,
- a *transition function* $f : S \mapsto S$,
- a finite set of *output* symbols U,
- an *output function* $g : S \mapsto U$.

Then the generation of random numbers is as follows:

1. generate the initial state (called the *seed*) s_0 according to μ and compute $u_0 = g(s_0)$,
2. iterate for $i = 1, \dots,$ $s_i = f(s_{i-1})$ and $u_i = g(s_i)$.

Generally, the seed s_0 is determined using the clock machine, and so the random variates $u_0, \dots, u_n, \dots$ seems "real" i.i.d. uniform random variates. The period of a RNG is the smallest integer $p \in \mathbb{N}$, such that $\forall n \in \mathbb{N}, s_{p+n} = s_n$.

2.1.1 Linear congruential generators

There are many families of RNGs : linear congruential, multiple recursive,... and "computer operation" algorithms. Linear congruential generators have a *transfer function* of the following type

$$f(x) = (ax + c) \mod m^1,$$

[1]this representation could be easily generalized for matrix, see L'Ecuyer (1990).

where a is the multiplier, c the increment and m the modulus and $x, a, c, m \in \mathbb{N}$ (i.e. S is the set of (positive) integers). f is such that

$$x_n = (ax_{n-1} + c) \mod m.$$

Typically,c and m are chosen to be relatively prime and a such that $\forall x \in \mathbb{N}, ax \mod m \neq 0$. When $c = 0$, we find the special of Park-Miller algorithm or Lehmer algorithm (see Park & Miller (1988)). Let us note that the $n+j$th term can be easily derived from the nth term with a puts to $a^j \mod m$ (still when $c = 0$).

Finally, we generally use of the three types of *output function*: $g(x) = x/m$, $x/(m-1)$ or $x/m + 1/m^2$. Linear congruential generators are implemented in the R function `congruRand`.

2.1.2 Multiple recursive generators

The multiple recursive generators are based on the following recurrences

$$x_n = (a_1 x_{n-1} + \dots + a_k x_{n-k} c) \mod m,$$

where k is a fixed integer. Hence the nth term of the sequence depends on the k previous one. A particular case of this type of generators is when

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

which is a Fibonacci-lagged generator[3]. This generator has been invented by Knuth (2002) and is implemented in the R function `runif` (see `.Random.seed`).

2.1.3 Mersenne-Twister

These two types of generators are in the big family of matrix linear congruential generators (cf. L'Ecuyer (1990)). But until here, no algorithms exploit the binary structure of computers (i.e.

[2]the last one has the good property to generate variate in $]0,1[$.

[3]see L'Ecuyer (1990).

use binary operations). In 1994, Matsumoto and Kurita invented the TT800 generator using binary operations. But Matsumoto & Nishimura (1998) greatly improved the use of binary operations and proposed a new type of random number generators, which could be called "Mersenne-Twister type"[1] generators.

Matsumoto & Nishimura (1998) work on the finite set $N_2 = \{0,1\}$, so a variable x is represented by a vectors of ω bits (e.g. 32 bits). They use the following linear recurrence for the kth term:

$$x_{k+n} = x_{k+m} \oplus (x_k^{upp} | x_{k+1}^{low}) A,$$

where n, m are constant integers, x_k^{upp} (respectively x_k^{low}) means the upper (lower) $\omega - r$ (r) bits of x_k and A a $\omega \times \omega$ matrix of N_2. $|$ is the operator of concatenation, so $x_k^{upp} | x_{k+1}^{low}$ appends the upper $\omega - r$ bits of x_k with the lower r bits of x_{k+1}. After a right multiplication with the matrix A[2], $\oplus$ adds the result with x_{k+m} bit to bit. Once provided an initial seed $x_0, \dots, x_{n-1}$, Mersenne Twister produces random integers in $0, \dots, 2^{omega} - 1$.

All operations used in the recurrence are bitwise operations, thus it is a very fast computation compared to modulus operations used in previous algorithms. To increase the equidistribution, Matsumoto & Nishimura (1998) added a tempering step:

$$\begin{aligned} y_k &\leftarrow x_{k+n} \oplus (x_{k+n} >> u), \\ y_k &\leftarrow y_k \oplus ((y_k << s) \oplus b), \\ y_k &\leftarrow y_k \oplus ((y_k << t) \oplus c), \\ y_k &\leftarrow y_k \oplus (y_k >> l), \end{aligned}$$

where $>> u$ (resp. $<< s$) denotes a rightshift (leftshift) of u (s) bits. At last, we transform random integers to reals with the output function $g(x) = \frac{x+1}{2^\omega}$. Details of the order of the successive operations used in the Mersenne-Twister (MT) algorithm can be found at the page 7 of Matsumoto & Nishimura (1998). However, the least, we need to learn and to retain, is all these (bitwise) operations can be easily done in many computer languages (e.g in C) ensuring a very fast algorithm.

[1]the name comes from the fact this generator has a Mersenne-prime number period.

[2]whose right multiplication can be done with a bitwise rightshift operation and an addition with integer a. See the section 2 of Matsumoto & Nishimura (1998) for explanations.

The set of parameters used are

- $(\omega, n, m, r) = (32, 624, 397, 31)$,
- $a = 0 \times 9908B0DF, b = 0 \times 9D2C5680, c = 0 \times EFC60000$,
- $u = 11$, $l = 18$, $s = 7$ and $t = 15$.

These parameters ensure a good equidistribution and a period of $2^{n\omega - r} - 1 = 2^{19937} - 1$.

The great advantages of the MT algorithm are a far longer period than any linear congruential generators (greater than the period of Park & Miller (1988) sequence of $2^{32} - 1$ or of Knuth (2002) around 2^{129}), a far better equidistribution (since it passed the DieHard test) as well as an VERY good computation time (since it used binary operations and not the costly real operation of modullus).

MT algorithm is already implemented in R (function `runif`). However the package `randtoolbox` provides a function to compute a ten-years improved version of Mersenne-Twister: SIMD-oriented Fast Mersenne Twister algorithm. A decade after the invention of MT, Matsumoto & Saito (2008) enhances their algorithm with the computer of today, that have Single Instruction Mutiple Data operations letting to work conceptually with 128 bits integers.

2.1.4 SF-Mersenne Twister

MT and its successor are part of the family of multiple-recursive matrix generators since they verify a multiple recursive equation with matrix constants. For MT, we have

$$x_{k+n} = x_{k+m} + x_{k+1} \begin{pmatrix} 0 & 0 \\ 0 & I_r \end{pmatrix} A + x_k \begin{pmatrix} I_{\omega-r} & 0 \\ 0 & 0 \end{pmatrix} A.$$

This recursive equation can be expressed with the following function

$$h(x_k, x_{k+1}, \dots, x_{k+n-1}) = x_m + (x_k|x_{k+1})A.$$

The recursion could be rewritten as

$$h(\omega_0, \dots, \omega_{n-1}) = \omega_m + (\omega_k|\omega_{k+1})A,$$

where ω_i denotes a 32-bit integer (i.e. horizontal vectors of N_2).

The general recurrence of SFMT generalizes MT recursion to

$$h(\omega_0, \dots, \omega_{n-1}) = \omega_0 A + \omega_m B + \omega_{n-2} C + \omega_{n-1} D,$$

where A, B, C, D are sparse matrix over N_2, ω_i could be integers of $\omega = 32, 64, 128$ bits and $n = \lceil \frac{19937}{128} \rceil = 156$. The right multiplication matrix operations can EASILY be done using SIMD opertations[1]. Hence the *transition function* of SFMT is given by

$$\begin{array}{rcl} f : (N_2^\omega)^n & \mapsto & (N_2^\omega)^n \\ (\omega_0, \dots, \omega_{n-1}) & \mapsto & (\omega_1, \dots, \omega_{n-1}, h(\omega_0, \dots, \omega_{n-1})), \end{array}$$

where $(N_2^\omega)^n$ is the *state* space.

There are various sets of parameters for SFMT which allow different periods from $2^{607} - 1$ to $2^{216091} - 1$. The advantage of SFMT over MT is the computation speed, SFMT is twice faster without SIMD operations and nearly fourth faster with SIMD operations. But SFMT has also a better equidistribution[2] and a better recovery time from zeros-excess states[3]. The function `SFMT` provides an interface to the C code of Matsumoto and Saito.

2.2 Quasi random generation

Before detailing and explaining quasi random generation, we must (quickly) explain Monte-Carlo methods, which have been introduced in the forties. In this section, we follow the approach of Niederreiter (1978). Let us work on the d-dimensional unit cube $I^d = [0,1]^d$ and with a (multivariate) bounded (Lebesgues) integrable function on I^d. Then we define the Monte Carlo approximation of integral of f over I^d by

$$\int_{I^d} f(x)dx \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i),$$

where $(x_i)_{1 \leq i \leq n}$ are independent random points from I^d. The strong law of large numbers ensures the almost surely convergence of the approximation. Furthermore, the expected integration error is bounded by $O(\frac{1}{\sqrt{n}})$, with the interesting fact it does not depend on dimension d.

The main difference between Monte-Carlo methods and quasi Monte-Carlo methods is that we no longer use random points $(x_i)_{1 \leq i \leq n}$ but deterministic points. Unlike statistical tests, numerical integration does not rely on true randomness. Let us note that quasi Monte-Carlo methods dates from the fifties, and have also been used for interpolation problems and integral equations solving.

In the following, we consider a sequence $(u_i)_{1 \leq i \leq n}$ of points in I^d, that are NOT random. As n increases, we want

$$\frac{1}{n} \sum_{i=1}^{n} f(u_i) \underset{n \to +\infty}{\longrightarrow} \int_{I^d} f(x)dx.$$

The condition on the sequence $(u_i)_i$ is to be uniformly distributed in the unit cube I^d with the following sense:

$$\forall J \subset I^d, \lim_{n \to +\infty} \frac{1}{n} \sum_{i=1}^{n} 1\!\!1_J(u_i) = \lambda_d(I),$$

where λ_d stands for the volume (i.e. the d-dimensional Lebesgue measure) and $1\!\!1_J$ the indicator function of subset J. The problem is that our discrete sequence will never constitute a "fair" distribution in I^d, since there will always be a small subset with no points.

Therefore, we need to consider a more fexible defition of uniform distribution of a sequence.

[1] see section 2.3 of Matsumoto & Saito (2008).

[2] See linear algebra arguments of Matsumoto & Nishimura (1998).

[3] states with too many zeros.

Before introducing the discrepancy, we need to define $\mathrm{Card}_E(u_1, \dots, u_n)$ as $\sum_{i=1}^n \mathbb{1}_E(u_i)$ the number of points in subset E. The discrepancy D_n of the n points $(u_i)_{1\leq i\leq n}$ in I^d is given by

$$D_n = \sup_{J\in\mathcal{J}} \left| \frac{\mathrm{Card}_J(u_1, \dots, u_n)}{n} - \lambda_d(J) \right|$$

where $\mathcal{J}$ denotes the family of all subintervals of I^d of the form $\prod_{i=1}^d [a_i, b_i]$. If we took the family of all subintervals of I^d of the form $\prod_{i=1}^d [0, b_i]$, D_n is called the star discrepancy (cf. Niederreiter (1992)).

Let us also note that the D_n discrepancy is nothing else than the L_∞-norm of points $(u_i)_{1\leq i\leq n}$ over the unit cube. A L_2-norm can be defined as well, see Niederreiter (1992) or Jäckel (2002).

The integral error is bounded by

$$\left| \frac{1}{n}\sum_{i=1}^n f(u_i) - \int_{I^d} f(x)dx \right| \leq V_d(f)D_n,$$

where $V_d(f)$ is the d-dimensional Hardy and Krause variation[1] of f on I^d (supposed to be finite). Let us notice that the integral error bound of the product of two independent quantities: the variability of function f through $V_d(f)$ and the regularity of the sequence through D_n. So, we want to minimize the discrepancy D_n.

We will not explain it but this type of result can be extented to subset J of the unit cube I^d in order to have a similar bound for $\int_J f(x)dx$.

In the literature, there were many ways to find sequences with small discrepancy, generally called low-discrepancy sequences or quasi-random points. A first approach tries to find bounds for these sequences and to search the good parameters to reach the lower bound or to decrease the upper bound. Another school tries to exploit regularity of function f to decrease the discrepancy. Sequences coming from the first school are called quasi-random points while those of the second school are called good lattice points.

[1]Interested readers can find the definition page 966 of Niederreiter (1978). In a sentence, the Hardy and Krause variation of f is the supremum of sums of d-dimensional delta operators applied to function f.

2.2.1 Quasi-random points and discrepancy

Until here, we do not give any example of quasi-random points. In the unidimensional case, an easy example of quasi-random points is the sequence of n terms given by $(\frac{1}{2n}, \frac{3}{2n}, \dots, \frac{2n-1}{2n})$. This sequence has a discrepancy $\frac{1}{n}$, see Niederreiter (1978) for details.

The problem with this finite sequence is it depends on n. If we want different points numbers, we need to recompute the whole sequence. In the following, we will work the first n points of an infinite sequence in order to use previous computation if we increase n.

We introduce the notion of discrepancy on a finite sequence $(u_i)_{1\leq i\leq n}$. In the example, we are able to calculate exactly the discrepancy. With infinite sequence, this is no longer possible. Thus, we will try to estimate asymptotic equivalents of discrepancy.

The discrepancy of the average sequence of points is governed by the law of the iterated logarithm :

$$\limsup_{n\to+\infty} \frac{\sqrt{n}D_n}{\sqrt{\log\log n}} = 1,$$

which leads to the following asymptotic equivalent for D_n:

$$D_n = O\left(\frac{\log\log n}{\sqrt{n}}\right).$$

2.2.2 Van der Corput sequences

An example of quasi-random points which are low discrepancy is the the (unidimensional) Van der Corput sequences. The big advantage of

Van der Corput sequence is that they use p-adic fractions easily computable on the binary structure of computers.

Let p be a prime number. Every integer n can decompose the p basis, i.e. there exists some integer k such that

$$n = \sum_{j=1}^{k} a_j p^j.$$

Thus, we can define the radical-inverse function of integer n as

$$\phi_p(n) = \sum_{j=1}^{k} \frac{a_j}{p^{j+1}}.$$

And finally, the Van der Corput sequence is given by $(\phi_p(0), \phi_p(1), \dots, \phi_p(n), \dots) \in [0, 1[$. First terms of those sequence for prime numbers 2 and 3 are given in table 1.

	n in p-basis			$\phi_p(n)$		
n	$p=2$	$p=3$	$p=5$	$p=2$	$p=3$	$p=5$
0	0	0	0	0	0	0
1	1	1	1	0.5	0.333	0.2
2	10	2	2	0.25	0.666	0.4
3	11	10	3	0.75	0.111	0.6
4	100	11	4	0.125	0.444	0.8
5	101	12	10	0.625	0.777	0.04
6	110	20	11	0.375	0.222	0.24
7	111	21	12	0.875	0.555	0.44
8	1000	22	13	0.0625	0.555	0.64

Table 1: Van der Corput first terms

2.2.3 Halton sequences

The d-dimensional version of the Van der Corput sequence is known as the Halton sequence. The nth term of the sequence is define as

$$(\phi_{p_1}(n), \dots, \phi_{p_d}(n)) \in I^d,$$

where $p_1, \dots, p_d$ are pairwise relatively prime bases. The discrepancy of the Halton sequence is asymptotically $O\left(\frac{\log(n)^d}{n}\right)$.

The following Halton theorem gives us better discrepancy estimate of infinite sequences. For any dimension $d \geq 1$, there exists an infinite sequence of points in I^d such that the discrepancy

$$D_n = O\left(\frac{\log(n)^{d-1}}{n}\right)^1.$$

Therefore, we have a significant guarantee there exists quasi-random points which are outperforming than traditional Monte-Carlo methods.

2.2.4 Faure sequences

The Faure sequences is also based on the decomposition of integers into prime-basis but they have two differences: it uses only one prime number for basis and it permutes vector elements from one dimension to another.

The basis prime number is chosen as the smallest prime number greater than the dimension d, i.e. 3 when $d = 2$, 5 when $d = 3$ or 4 etc... In the Van der Corput sequence, we decompose integer n into the p-basis:

$$n = \sum_{j=1}^{k} a_j p^j.$$

Let $a_{1,j}$ be integer a_j used for the decomposition of n. Now we define a recursive permutation of a_j:

$$\forall 2 \leq D \leq d, a_{D,j} = \sum_{j=i}^{k} C_j^i a_{D-1,j} \mod p,$$

where C_i^j denotes standard combination $\frac{i!}{j!(i-j)!}$. Then we take the radical-inversion $\phi_p(a_{D,1}, \dots, a_{D,k})$ defined as

$$\phi_p(a_1, \dots, a_k) = \sum_{j=1}^{k} \frac{a_j}{p^{j+1}},$$

which is the same as above for n defined by the $a_{D,i}$'s.

[1]if the sequence has at least two points, cf. Niederreiter (1978).

Finally the (d-dimensional) Faure sequence is defined by

$$(\phi_p(a_{1,1},\dots,a_{1,k}),\dots,\phi_p(a_{d,1},\dots,a_{d,k}))\in I^d.$$

In the bidimensional case, we work in 3-basis, first terms of the sequence are listed in table 2.

n	$a_{13}a_{12}a_{11}$[1]	$a_{23}a_{22}a_{21}$	$\phi(a_{13}..)$	$\phi(a_{23}..)$
0	0	0	0	0
1	1	1	1/3	1/3
2	2	2	2/3	2/3
3	10	12	1/9	7/9
4	11	10	4/9	1/9
5	12	11	7/9	4/9
6	20	21	2/9	5/9
7	21	22	5/9	8/9
8	22	20	8/9	2/9

Table 2: Faure first terms

2.2.5 Kronecker sequences

Another kind of low-discrepancy sequence uses irrational number and fractional part. The fractional part of a real x is given by $\{x\}=x-\lfloor x\rfloor$. The infinite sequence $(n\{\alpha\})_{n\leq 0}$ has a bound for its discrepancy

$$D_n\leq C\frac{1+\log n}{n}.$$

This family of infinite sequence $(n\{\alpha\})_{n\leq 0}$ is called the Kronecker sequence.

A special case of the Kronecker sequence is the Torus algorithm where irrational number α is a square root of a prime number. The nth term of Torus algorithm is defined by

$$(n\{\sqrt{p_1}\},\dots,n\{\sqrt{p_d}\})\in I^d,$$

where $(p_1,\dots,p_d)$ are prime numbers, generally the first d prime numbers. With the previous inequality, we can derive an estimate of the Torus algorithm discrepancy:

$$O\left(\frac{1+\log n}{n}\right).$$

[1] we omit commas for simplicity.

2.2.6 Mixed pseudo quasi random sequences

Sometimes we want to use quasi-random sequences as pseudo random ones, i.e. we want to keep the good equidistribution of quasi-random points but without the term-to-term dependence.

One way to solve this problem is to use pseudo random generation to mix outputs of a quasi-random sequence. For example in the case of the Torus sequence, we have repeat for $1\leq i\leq n$

- draw an integer n_i from Mersenne-Twister in $\{0,\dots,2^\omega-1\}$
- then $u_i=\{n_i\sqrt{p}\}$

2.2.7 Good lattice points

In the above methods we do not take into account a better regularity of the integrand function f than to be of bounded variation in the sense of Hardy and Krause. Good lattice point sequences try to use the eventual better regularity of f.

If f is 1-periodic for each variable, then the approximation with good lattice points is

$$\int_{I^d}f(x)dx\approx\frac{1}{n}\sum_{i=1}^{n}f(\frac{i}{n}g),$$

where $g\in\mathbb{Z}^d$ is suitable d-dimensional lattice point. To impose f to be 1-periodic may seem too brutal. But there exists a method to transform f into a 1-periodic function while preserving regularity and value of the integrand (see Niederreiter (1978) page 983).

We have the following theorem for good lattice points. For every dimension $d\geq 2$ and integer $n\geq 2$, there exists a lattice points $g\in\mathbb{Z}^d$ which coordinates relatively prime to n such that the discrepancy D_n of points $\{\frac{1}{n}g\},\dots,\{\frac{n}{n}g\}$ satisfies

$$D_s<\frac{d}{n}+\frac{1}{2n}\left(\frac{7}{5}+2\log m\right)^d.$$

Numerous studies of good lattice points try to find point g which minimizes the discrepancy. Korobov test g of the following form $(1, m, \dots, m^{d-1})$ with $m \in \mathbb{N}$. Bahvalov tries Fibonnaci numbers $(F_1, \dots, F_d)$. Other studies look directly for the point $\alpha = \frac{g}{n}$ e.g. $\alpha = (p^{\frac{1}{d+1}}, m, \dots, p^{\frac{d}{d+1}})$ or some cosinus functions. We let interested readers to look for detailed information in Niederreiter (1978).

3 Description of the random generation functions

In this section, we detail the R functions implemented in `randtoolbox` and give examples.

3.1 Pseudo random generation

For pseudo random generation, R provides many algorithms through the function `runif` parametrized with `.Random.seed`. We encourage readers to look in the corresponding help pages for examples and usage of this function. Let us just say `runif` use the Mersenne-Twister algorithm by default and other generators such as Wichmann-Hill, Marsaglia-Multicarry or Knuth-TAOCP-2002[1].

3.1.1 congruRand

The `randtoolbox` package provides two pseudo-random generators functions : `congruRand` and `SFMT`. `congruRand` computes linear congruential generators, see sub-section 2.1.1. By default, it computes the Park & Miller (1988) sequence, so it needs only the observation number argument. If we want to generate 10 random numbers, we type

```
> congruRand(10)
```

```
 [1] 0.6485238 0.7387460 0.1032293
 [4] 0.9747305 0.2957171 0.1177271
 [7] 0.6395649 0.1672669 0.2548319
[10] 0.9590469
```

One will quickly note that two calls to `congruRand` will not produce the same output. This is due to the fact that we use the machine time to initiate the sequence at each call. But the user can set the *seed* with the function `setRandSeed`:

```
> setRandSeed(1)
> congruRand(10)
```

```
 [1] 7.826369e-06 1.315378e-01
 [3] 7.556053e-01 4.586501e-01
 [5] 5.327672e-01 2.189592e-01
 [7] 4.704462e-02 6.788647e-01
 [9] 6.792964e-01 9.346929e-01
```

One can follow the evolution of the nth integer generated with the option `echo=TRUE`.

```
> setRandSeed(1)
> congruRand(10, echo = TRUE)
```

```
1 th integer generated : 16807
2 th integer generated : 282475249
3 th integer generated : 1622650073
4 th integer generated : 984943658
5 th integer generated : 1144108930
6 th integer generated : 470211272
7 th integer generated : 101027544
8 th integer generated : 1457850878
9 th integer generated : 1458777923
10 th integer generated : 2007237709
 [1] 7.826369e-06 1.315378e-01
 [3] 7.556053e-01 4.586501e-01
 [5] 5.327672e-01 2.189592e-01
 [7] 4.704462e-02 6.788647e-01
 [9] 6.792964e-01 9.346929e-01
```

The integers used for the 10 first terms are listed in table 3.

[1]see Wichmann & Hill (1982), Marsaglia (1994) and Knuth (2002)

n	x_n	n	x_n
1	16807	6	470211272
2	282475249	7	101027544
3	1622650073	8	1457850878
4	984943658	9	1458777923
5	1144108930	10	2007237709

Table 3: 10 first integers of Park & Miller (1988) sequence

We can check that the 9998th term of the sequence. From CITATION, we know for sure that integers of the Park-Miller sequence are 925166085, 1484786315, 1043618065, 1589873406, 2010798668. The `congruRand` generates

```
> setRandSeed(1614852353)
> congruRand(5, echo = TRUE)
```

```
1 th integer generated : 925166085
2 th integer generated : 1484786315
3 th integer generated : 1043618065
4 th integer generated : 1589873406
5 th integer generated : 2010798668
[1] 0.4308140 0.6914075 0.4859725
[4] 0.7403425 0.9363511
```

with 1614852353 being the 9997th term.

However, we are not limited to the Park-Miller sequence. If we change the modulus, the increment and the multiplier, we get other random sequences. For example,

```
> setRandSeed(12)
> congruRand(5, mod = 2^8, mult = 25,
+      incr = 16, echo = TRUE)
```

```
1 th integer generated : 60
2 th integer generated : 236
3 th integer generated : 28
4 th integer generated : 204
5 th integer generated : 252
[1] 0.234375 0.921875 0.109375
[4] 0.796875 0.984375
```

Those values are correct according to Planchet et al. (2005) (page 119).

Here is a list of RNGs computable with `congruRand`:

RNG	`mod`	`mult`	`incr`
Knuth - Lewis	2^{32}	1664525	$1.01e9$[1]
Lavaux - Jenssens	2^{48}	31167285	1
Haynes	2^{64}	$6.36e17$[2]	1
Marsaglia	2^{32}	69069	0
Park - Miller	$2^{32}-1$	16807	0

Table 4: some linear RNGs

One may wonder why we implement such a short-period algorithm since we know the Mersenne-Twister algorithm. It is provided to make comparisons with other algorithms. The Park-Miller should *not* be viewed as a "good" random generator.

Finally, `congruRand` function has a `dim` argument to generate `dim`- dimensional vectors of random numbers. The nth vector is build with d consecutive numbers of the RNG sequence (i.e. $(u_{n+1}, \dots, u_{n+d})$).

3.1.2 SFMT

The SF- Mersenne Twister algorithm is described in sub-section 2.1.4. Usage of `SFMT` function implementing the SF-Mersenne Twister algorithm is the same. First argument `n` is the number of random variates, second argument `dim` the dimension.

```
> SFMT(10)
> SFMT(5, 2)
```

```
 [1] 0.82755719 0.41069176 0.47995492
 [4] 0.06337025 0.63308225 0.07371121
```

[1] 1013904223
[2] 6364122338467930005

```
 [7] 0.75031772 0.39574669 0.56246988
[10] 0.78700141
```

```
          [,1]       [,2]
[1,] 0.3142382 0.50739370
[2,] 0.3066768 0.94906450
[3,] 0.1136327 0.06708027
[4,] 0.7164584 0.42288663
[5,] 0.1820726 0.82182600
```

We must precise that we do *not* implement the SFMT algorithm, we "just" use the C code of Matsumoto & Saito (2008). For the moment, we do not fully use the strength of their code. For example, we do not use block generation for multivariate generation as well as SIMD operations.

3.2 Quasi-random generation

In a near future, randtoolbox package will have more than one quasi-random sequence. Currently, we only have the Torus sequence with the function torus. Its usage is the same.

```
> torus(10)
```

```
 [1] 0.41421356 0.82842712 0.24264069
 [4] 0.65685425 0.07106781 0.48528137
 [7] 0.89949494 0.31370850 0.72792206
[10] 0.14213562
```

These numbers are fractional parts of $\sqrt{2}, 2\sqrt{2}, 3\sqrt{2}, \dots,$ see sub-section 2.2.1 for details.

```
> torus(5, use = TRUE)
```

```
[1] 0.1375012 0.5517149 0.9659286
[4] 0.3801417 0.7943554
```

The optional argument useTime can be used to the machine time or not to initiate the seed. If we do not use the machine time, two calls of torus produces obviously the same output.

If we want the random sequence with prime number 7, we just type:

```
> torus(5, p = 7)
```

```
[1] 0.6457513 0.2915026 0.9372539
[4] 0.5830052 0.2287566
```

As described in sub-section 2.2.6, one way to deal with serial dependence is to mix the Torus algorithm with a pseudo random generator. The torus function offers this operation thanks to argument mixed (the Torus algorithm is mixed with SFMT).

```
> torus(5, mixed = TRUE)
```

```
[1] 0.8823128 0.3098669 0.3084612
[4] 0.9642344 0.2850962
```

In order to see the difference between, we can plot the empirical autocorrelation function acf.

```
> par(mfrow = c(2, 1))
> acf(torus(10^5))
> acf(torus(10^5, mix = TRUE))
```

3.3 Visual comparisons

Before doing some serious statistical tests, we can make visual comparisons of how random numbers fill the unit square.

```
> par(mfrow = c(2, 1))
> plot(SFMT(1000, 2))
> plot(torus(10^3, 2))
```

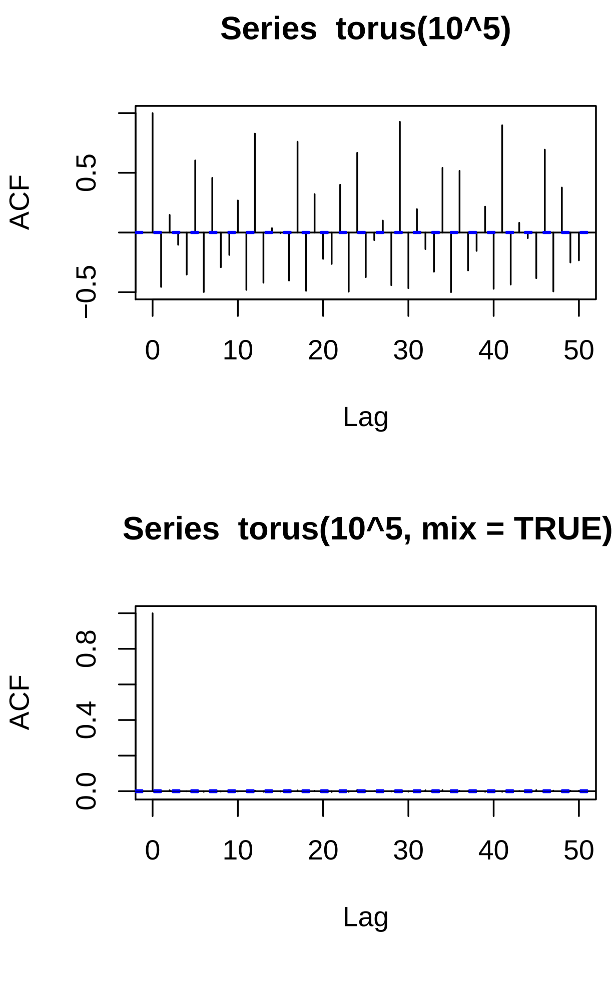


Figure 1: Auto-correlograms

3.4 Applications of QMC methods

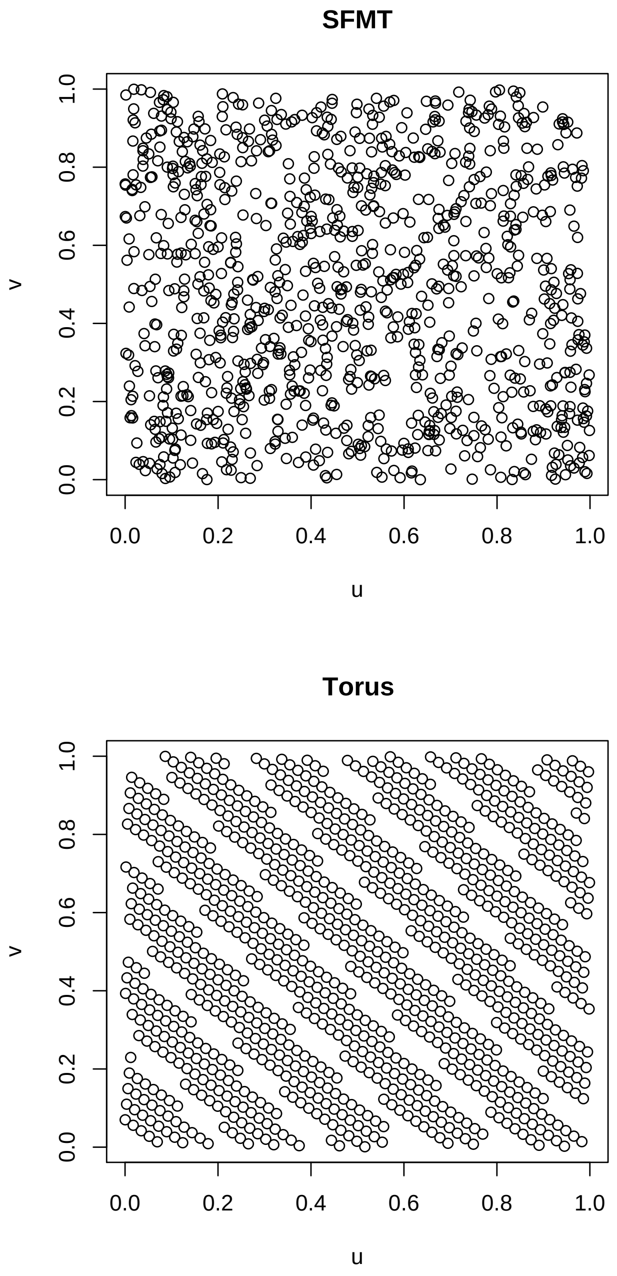
In this sub-section, we will present one financial application of QMC methods. Firstly, we want to price a vanilla European call. In the framework of a geometric Brownian motion for the underlying asset, Those options are already implemented in the package `fOptions` of `Rmetrics` bundle[1].

The payoff of this classical option is

$$f(S_T) = (S_T - K)_+,$$

where K is the strike price. A closed formula for this call was derived by Black & Scholes (1973).

[1]created by Wuertz et al. (2007*b*).



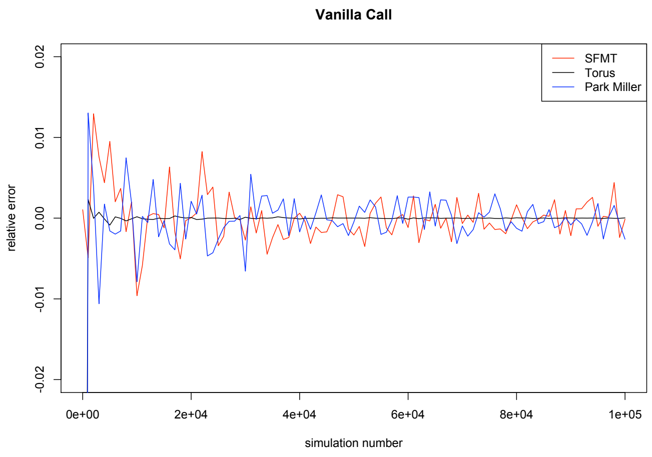
The Monte Carlo method to price this option is quite simple

1. simulate $s_{T,i}$ for $i = 1 \ldots n$ from starting point S_0,
2. compute the mean of the discounted payoff $\frac{1}{n}\sum_{i=1}^{n} e^{-rT}(s_{T,i} - K)_+$.

With parameters ($S_0 = 100$, $T = 1$, $r = 5\%$,

$K = 60$, $\sigma = 20\%$), we get the following relative error as a function of number of simulation n.

We test two pseudo-random generators (namely Park Miller and SF-Mersenne Twister) and one quasi-random generator (Torus algorithm). No code will be shown, see the file `qmc.R` in the package source. But we use a step-by-step simulation for the Brownian motion simulation and the inversion function method for Gaussian distribution simulation (default in R).



As showed on the supra figure, the convergence of Monte Carlo price for the Torus algorithm is extremly fast. Whereas for SF-Mersenne Twister and Park Miller prices, the convergence is very slow.

Secondly, we want to price a barrier option: a down-out call i.e. an Downward knock-Out Call[1]. These kind of options are path-dependent, i.e. we need to simulate a whole trajectory of the underlying asset on $[0, T]$.

In the same framework of a geometric Brownian motion, there exists a closed formula for DOCs (see Rubinstein & Reiner (1991)). Those options are already implemented in the package `fExoticOptions` of `Rmetrics` bundle[2].

[1]DOC is disactived when the underlying asset hits the barrier.

[2]created by Wuertz et al. (2007*a*).

The payoff of a DOC option is

$$f(S_T) = (S_T - K)_+ \mathbb{1}_{(\tau_H > T)},$$

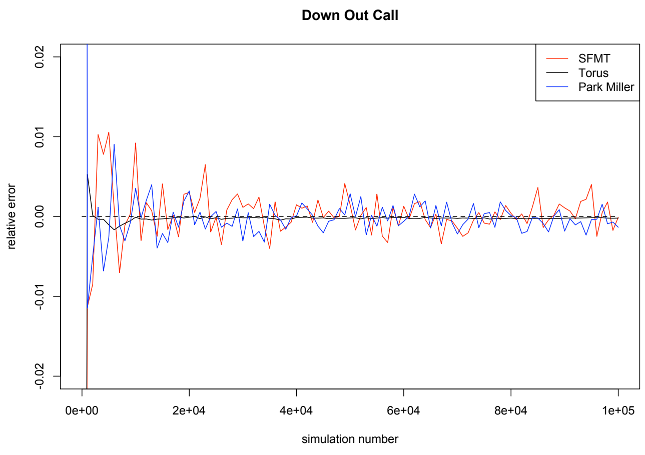
where K is the strike price, T the maturity, τ_H the stopping time associated with the barrier H and S_t the underlying asset at time t.

As the price is needed on the whole period $[0, T]$, we produc as follows:

1. start from point s_{t_0},
2. for simulation $i = 1 \dots n$ and time index $j = 1 \dots d$
   - simulate $s_{t_j,i}$,
   - update disactivation boolean D_i
3. compute the mean of the discounted payoff $\frac{1}{n}\sum_{i=1}^{n} e^{-rT}(s_{T,i} - K)_+ \overline{D}_i$,

where n is the simulation number, d the point number for the grid of time and $\overline{D}_i$ the opposite of boolean D_i.

In the following, we set $T = 1$, $r = 5\%$, $S_0 = 100$, $H = K = 50$, $d = 250$ and $\sigma = 20\%$. We test crude Monte Carlo methods with Park Miller and SF-Mersenne Twister generators and a quasi-Monte Carlo method with (multidimensional) Torus algoritm on the figure below.



One may wonder how the Torus algorithm is still the best (on this example). We use the d-dimensional Torus sequence. Thus for time t_j, the simulated underlying assets $(s_{t_j,i})_i$ are computed with the sequence $(i\{\sqrt{p_j}\})_i$. Thanks to the linear independence of the Torus sequence over the rationals[1], we guarantee a non-correlation of Torus quasi-random numbers.

However, these results do *not* prove the Torus algorithm is always better than traditional Monte Carlo. The results are sensitive to the barrier level H, the strike price X (being in or out the money has a strong impact), the asset volatility σ and the time point number d.

4 Random generation tests

Tests of random generators aim to check the output $u_1, \dots, u_n, \dots$ could be considered as independent and identically distributed (i.i.d.) uniform variates. There are two kinds of tests of the uniform distribution: first on the interval $]0,1[$, second on the binary set $\{0,1\}$. In this note, we only describe tests for $]0,1[$ outputs (see L'Ecuyer & Simard (2007) for details about these two kind of tests).

Some RNG tests can be two-level tests, i.e. we do not work directly on the RNG output u_i's but on a function of the output such as the spacings (coordinate difference of the sorted sample).

4.1 Test on one sequence of n numbers

4.1.1 Goodness of Fit

Goodness of Fit tests compare the empirical distribution $\mathbb{F}_n$ of u_i's with a specific distribution ($\mathcal{U}(0,1)$ here).The most known test are Kolmogorov-Smirnov, Crámer-von Mises and Anderson-Darling tests. They use different norms to quantify the difference between $\mathbb{F}_n$ and $F_{\mathcal{U}_{(0,1)}}$.

[1]i.e. for $k \neq j, \forall i, (i\{\sqrt{p_j}\})_i$ and $(i\{\sqrt{p_k}\})_i$ are linearly independent over $\mathbb{Q}$.

- Kolmogorov-Smirnov statistic is

$$K_n = \sqrt{n} \sup_{x\in\mathbb{R}} \left| \mathbb{F}_n(x) - F_{\mathcal{U}_{(0,1)}}(x) \right|,$$

- Crámer-von Mises statistic is

$$W_n^2 = n \int_{-\infty}^{+\infty} \Big(\mathbb{F}_n(x) - F_{\mathcal{U}_{(0,1)}}(x)\Big)^2 dF_{\mathcal{U}_{(0,1)}}(x),$$

- and Anderson-Darling statistic is

$$A_n^2 = n \int_{-\infty}^{+\infty} \frac{\Big(\mathbb{F}_n(x) - F_{\mathcal{U}_{(0,1)}}(x)\Big)^2 dF_{\mathcal{U}_{(0,1)}}(x)}{F_{\mathcal{U}_{(0,1)}}(x)(1 - F_{\mathcal{U}_{(0,1)}}(x))}.$$

Those statistics can be evaluated empirically thanks to the sorted sequence of u_i's. But we will not detail any further those tests, since according to L'Ecuyer & Simard (2007) they are not powerful for random generation testing.

4.1.2 The gap test

The gap test investigates for special patterns in the sequence $(u_i)_{1\leq i\leq n}$. We take a subset $[l,u] \subset [0,1]$ and compute the 'gap' variables with

$$G_i = \begin{cases} 1 & \text{if } l \leq U_i \leq u \\ 0 & \text{otherwise.} \end{cases}$$

The probability p that G_i equals to 1 is just the $u - l$ (the Lebesgue measure of the subset). The test computes the length of zero gaps. If we denote by n_j the number of zero gaps of length j.

The chi-squared statistic of a such test is given by

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

where $p_j = (1-p)^2 p^j$ stands for the probability the length of gaps equals to j and m the max number of lengths. We fix m to be at least

$$\left\lfloor \frac{\log(10^{-1}) - 2\log(1-p) - log(n)}{\log(p)} \right\rfloor,$$

in order to have lengths whose appearance probabilitie is at least 0.1.

4.1.3 The order test

The order test looks for another kind of patterns. For a d-tuple, if the coordinates are ordered equiprobably. For example with $d = 3$, we should have an equal number of vectors $(u_i, u_{i+1}, u_{i+2})_i$ such that

- $u_i < u_{i+1} < u_{i+2}$,
- $u_i < u_{i+2} < u_{i+1}$,
- $u_{i+1} < u_i < u_{i+2}$,
- $u_{i+1} < u_{i+2} < u_i$,
- $u_{i+2} < u_i < u_{i+1}$
- and $u_{i+1} < u_{i+2} < u_i$.

For some d, we have $d!$ possible orderings of coordinates, which have the same probability to appear $\frac{1}{d!}$. The chi-squared statistic for the order test for a sequence $(u_i)_{1\leq i\leq n}$ is just

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

where n_j's are the counts for different orders and $m = \frac{n}{d}$.

4.1.4 The frequency test

The frequency test works on a serie of ordered contiguous integers ($J = [i_1, \dots, i_l] \cap \mathbb{Z}$). If we denote by $(n_i)_{1\leq i\leq n}$ the sample number of the set I, the expected number of integers equals to $j \in J$ is

$$\frac{1}{i_l - i_1 + 1} \times n,$$

which is independent of j. From this, we can compute a chi-squared statistic

$$S = \sum_{j=1}^{l} \frac{(\text{Card}(n_i = i_j) - m)^2}{m}.$$

4.2 Tests based on multiple sequences

Under the i.i.d. hypothesis, a vector of output values $u_i, \dots, u_{i+t-1}$ is uniformly distributed over the unit hypercube $[0, 1]^t$. Tests based on multiple sequences partition the unit hypercube into cells and count the number of points in each cell.

4.2.1 The serial test

The most intuitive way to split the unit hypercube $[0, 1]^t$ into $k = d^t$ subcubes. It is achieved by splitting each dimension into $d > 1$ pieces. The volume (i.e. a probability) of each cell is just $\frac{1}{k}$.

The associated chi-square statistic is defined as

$$S = \sum_{j=1}^{m} \frac{(N_j - \lambda)^2}{n\lambda},$$

where N_j denotes the counts and $\lambda = \frac{n}{k}$ their expectation.

4.2.2 The collision test

The philosophy is still the same: we want to detect some pathological behavior on the unit hypercube $[0, 1]^t$. A collision is defined as when a point $v_i = (u_i, \dots, u_{i+t-1})$ falls in a cell where there are already points v_j's. 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[1] 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. As L'Ecuyer et al. (2002) we use a Gaussian approximation if $\lambda = \frac{n}{k} > \frac{1}{32}$ and $n \geq 2^8$, a Poisson approximation if $\lambda < \frac{1}{32}$ and the exact formula otherwise.

[1]they are defined by ${}_2S_n^k = k \times {}_2S_{n-1}^k + {}_2S_{n-1}^{k-1}$ and ${}_2S_n^1 = {}_2S_n^n = 1$. For example go to wikipedia.

The normal approximation assumes C follows a normal distribution with mean $m = n - k + k\left(\frac{k-1}{k}\right)^n$ and variance very complex (see L'Ecuyer & Simard (2007)). Whereas the Poisson approximation assumes C follows a Poisson distribution of parameter $\frac{n^2}{2k}$.

4.2.3 The ϕ-divergence test

There exists generalizations of these tests where we take a function of counts N_j, which we called ϕ-divergence test. Let f be a real valued function. The test statistic is given by

$$\sum_{j=0}^{k-1} f(N_j).$$

We retrieve the collision test with $f(x) = (x-1)_+$ and the serial test with $f(x) = \frac{(x-\lambda)^2}{\lambda}$. Plenty of statistics can be derived, for example if we want to test the number of cells with at least b points, $f(x) = \mathbb{1}_{(x=b)}$. For other statistics, see L'Ecuyer et al. (2002).

4.2.4 The poker test

The poker test is a test where cells of the unit cube $[0,1]^t$ do not have the same volume. If we split the unit cube into d^t cells, then by regrouping cells with left hand corner having the same number of distinct coordinates we get the poker test. In a more intuitive way, let us consider a hand of k cards from k different cards. The probability to have exactly c different cards is

$$P(C = c) = \frac{1}{k^k} \frac{k!}{(k-c)!} \, {}_2S_k^c,$$

where C is the random number of different cards and ${}_2S_n^d$ the second-kind Stirling numbers. For a demonstration, go to Knuth (2002).

5 Description of RNG test functions

In this section, we will give usage examples of RNG test functions, in a similar way as section 3 illustrates section 2.

5.1 Test on one sequence of n numbers

Goodness of Fit tests are already implemented in R with the function `ks.test` for Kolmogorov-Smirnov test and in package `adk` for Anderson-Darling test. In the following, we will focus on one-sequence test implemented in `randtoolbox`.

5.1.1 The gap test

The function `gap.test` implements the gap test as described in sub-section 4.1.2. By default, lower and upper bound are $l = 0$ and $u = 0.5$, just as below.

```
> gap.test(runif(1000))


          Gap test


chisq stat = 15, df = 10
, p-value = 0.14


(sample size : 1000)


length observed freq theoretical freq


1          119            125
2          56            62
3          38             31
4          8            16
5          10             7.8
```

```
6            5           3.9
7            3           2.0
8            3           0.98
9            1           0.49
10            1           0.24
11            0           0.12
```

If you want $l = 1/3$ and $u = 2/3$ with a SFMT sequence, you just type

```
> gap.test(SFMT(1000), 1/3, 2/3)
```

5.1.2 The order test

The Order test is implemented in function `order.test` for d-uples when $d = 2, 3, 4, 5$. A typical call is as following

```
> order.test(runif(4000), d = 4)

         Order test

chisq stat = 16, df = 15
, p-value = 0.35

 (sample size : 1000)

observed number  37 55 44 41 40 39
 45 35 37 42 39 38 43 56 36 46 38 41
 34 41 47 43 39 44

expected number  42
```

Let us notice that the sample length must be a multiple of dimension d, see sub-section 4.1.3.

5.1.3 The frequency test

The frequency test described in sub-section 4.1.4 is just a basic equi-distribution test in $[0, 1]$ of the generator. We use a sequence integer to partition the unit interval and test counts in each sub-interval.

```
> freq.test(runif(1000), 1:4)

         Frequency test

chisq stat = 3, df = 3
, p-value = 0.39

 (sample size : 1000)

observed number  259 247 265 229

expected number  250
```

5.2 Tests based on multiple sequences

Let us study the serial test, the collision test and the poker test.

5.2.1 The serial test

Defined in sub-section 4.2.1, the serial test focuses on the equidistribution of random numbers in the unit hypercube $[0, 1]^t$. We split each dimension of the unit cube in d equal pieces. Currently in function `serial.test`, we implement $t = 2$ and d fixed by the user.

```
> serial.test(runif(3000), 3)

         Serial test

chisq stat = 4.9, df = 8
, p-value = 0.77
```

```
 (sample size : 3000)


observed number  169 159 175 172
 157 160 186 167 155


expected number  167
```

In newer version, we will add an argument t for the dimension.

5.2.2 The collision test

The exact distribution of collision number costs a lot of time when sample size and cell number are large (see sub-section 4.2.2). With function `coll.test`, we do not yet implement the normal approximation.

The following example tests Mersenne-Twister algorithm (default in R) and parameters implying the use of the exact distribution:

```
> coll.test(runif, 2^7, 2^10)


         Collision test


chisq stat = 20, df = 16
, p-value = 0.22


 exact distribution
(sample number : 1000/sample size : 128
 / cell number : 1024)


collision observed expected
number    count    count


  1       4      2.3
  2       15      10
  3       28      29
  4       74      62
  5       115      102
  6       148      138
  7       139      156
  8       130      151
  9       131      126
  10       92      93
  11       60      61
  12       36      36
  13       13      19
  14       10      9
  15       2      3.9
  16       1      1.5
  17       2      0.56
```

When the cell number is far greater than the sample length, we use the Poisson approximation. For example with `congruRand` generator we have

```
> coll.test(congruRand, 2^8, 2^14)


         Collision test


chisq stat = 57, df = 7
, p-value = 5.6e-10


 Poisson approximation
(sample number : 1000/sample size : 256
 / cell number : 16384)


collision observed expected
number    count    count


  0       177      135
  1       212      271
  2       270      271
  3       183      180
  4       129      90
  5       26      36
  6       0      12
  7       3      3.4
```

5.2.3 The poker test

Finally the function `poker.test` implements the poker test as described in sub-section 4.2.4. We implement for any "card number" denoted by k. A typical example follows

```
> poker.test(SFMT(10000))

        Poker test

chisq stat = 0.92, df = 4
, p-value = 0.92

 (sample size : 10000)

observed number  3 193 979 752 73

expected number  3.2 192 960 768 77
```

6 Calling the functions from other packages

In this section, we briefly present what to do if you want to use this package in your package. This section is mainly taken from package `expm` available on R-forge.

Package authors can use facilities from **randtoolbox** in two ways:

- call the R level functions (e.g. `torus`) in R code;
- if random number generators are needed in C, call the routine `torus`,...

Using R level functions in a package simply requires the following two import directives:

```
Imports: randtoolbox
```

in file `DESCRIPTION` and

```
import(randtoolbox)
```

in file `NAMESPACE`.

Accessing C level routines further requires to prototype the function name and to retrieve its pointer in the package initialization function `R_init_`*pkg*, where *pkg* is the name of the package.

For example if you want to use `torus` C function, you need

```
void (*torus)(double *u, int nb, int dim,
int *prime, int ismixed, int usetime);

void R_init_pkg(DllInfo *dll)
{
torus = (void (*) (double, int, int,
int, int, int) \
R_GetCCallable("randtoolbox", "torus");
}
```

See file `randtoolbox.h` to find headers of RNGs. The definitive reference for these matters remains the *Writing R Extensions* manual, page 20 in sub-section "specifying imports exports" and page 64 in sub-section "registering native routines".

References

Black, F. & Scholes, M. (1973), 'The pricing of options and corporate liabilities', *Journal of Political Economy* **81**(3). 11

Eddelbuettel, D. (2007), *random: True random numbers using random.org.*
URL: *http://www.random.org* 1

Jäckel, P. (2002), *Monte Carlo methods in finace*, John Wiley & Sons. 5

Knuth, D. E. (2002), *The Art of Computer Programming: seminumerical algorithms*, Vol. 2, 3rd edition edn, Massachusetts: Addison-Wesley. 2, 3, 8, 15

L'Ecuyer, P. (1990), 'Random numbers for simulation', *Communications of the ACM* **33**, 85–98. 2

L'Ecuyer, P. & Simard, R. (2007), 'Testu01: A c library for empirical testing of random number generators', *ACM Trans. on Mathematical Software* **33**(4), 22. 13, 15

L'Ecuyer, P., Simard, R. & Wegenkittl, S. (2002), 'Sparse serial tests of uniformity for random number generations', *SIAM Journal on scientific computing* **24**(2), 652–668. 14, 15

Marsaglia, G. (1994), 'Some portable very-long-period random number generators', *Computers in Physics* **8**, 117–121. 8

Matsumoto, M. & Nishimura, T. (1998), 'Mersenne twister: A 623-dimensionnally equidistributed uniform pseudorandom number generator', *ACM Trans. on Modelling and Computer Simulation* **8**(1), 3–30. 3, 4

Matsumoto, M. & Saito, M. (2008), *SIMD-oriented Fast Mersenne Twister: a 128-bit pseudorandom number generator*, Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer. 3, 4, 10

Niederreiter, H. (1978), 'Quasi-monte carlo methods and pseudo-random numbers', *Bulletin of the American Mathematical Society* **84**(6). 1, 4, 5, 6, 7, 8

Niederreiter, H. (1992), *Random Number Generation and Quasi-Monte Carlo Methods*, SIAM, Philadelphia. 5

Park, S. K. & Miller, K. W. (1988), 'Random number generators: good ones are hard to find.', *Association for Computing Machinery* **31**(10), 1192–2001. 2, 3, 8, 9

Planchet, F., Thérond, P. & Jacquemin, J. (2005), *Modèles Financiers En Assurance*, Economica. 9

Rubinstein, M. & Reiner, E. (1991), 'Unscrambling the binary code', *Risk Magazine* **4**(9). 12

Wichmann, B. A. & Hill, I. D. (1982), 'Algorithm as 183: An efficient and portable pseudo-random number generator', *Applied Statistics* **31**, 188–190. 8

Wuertz, D., many others & see the SOURCE file (2007*a*), *fExoticOptions: Rmetrics - Exotic Option Valuation.*
URL: *http://www.rmetrics.org* 12

Wuertz, D., many others & see the SOURCE file (2007*b*), *fOptions: Rmetrics - Basics of Option Valuation.*
URL: *http://www.rmetrics.org* 11