Lecture 3

Pseudo-random number generation

Lecture Notes

by Jan Palczewski

with additions
by Andrzej Palczewski

Congruential generator

Define a sequence (s_i) recurrently

$$s_{i+1} = (as_i + b) \mod M.$$

Then

$$U_i = \frac{s_i}{M}$$

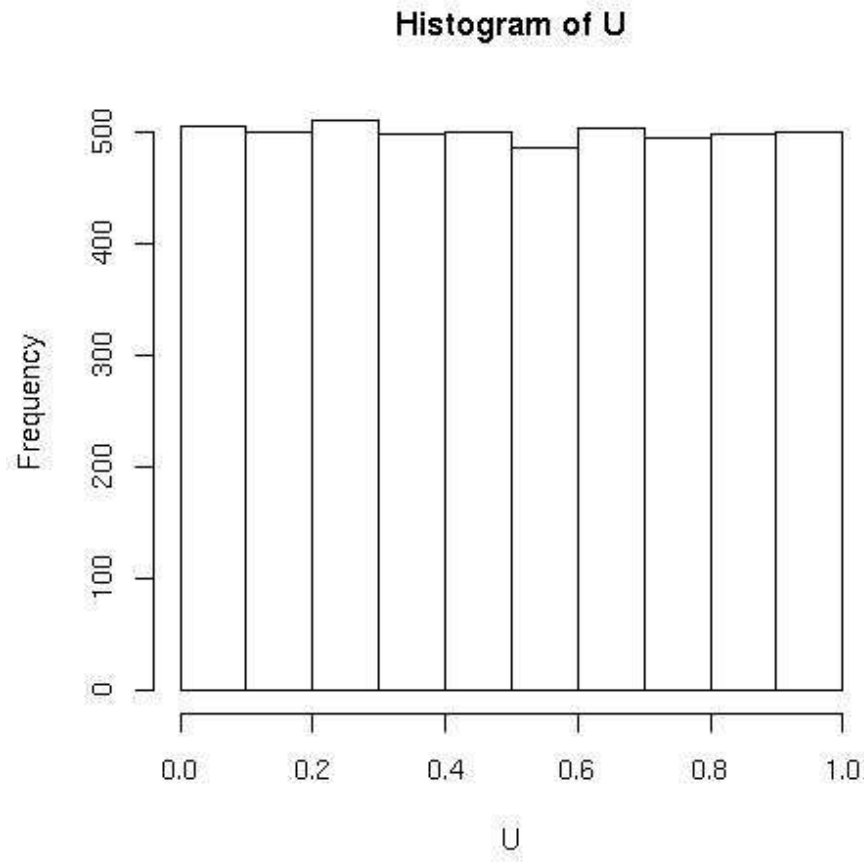
form an infinite sample from a uniform distribution on $[0, 1]$.

The numbers s_i have the following properties:

1. $s_i \in \{0, 1, 2, \ldots, M-1\}$
2. s_i are periodic with period $< M$. Indeed, since at least two values in $\{s_0, s_1, \ldots, s_M\}$ must be identical, therefore $s_i = s_{i+p}$ for some $p \leq M$.

In view of property 2 the number M should be as large as possible, because a small set of numbers make the outcome easier to predict.

But still p can be small even is M is large!



5000 uniform variates

$a = 1229$, $b = 1$, $M = 2048$, $s_0 = 1$

Are the numbers U_i obtained with the congruential linear generator uniformly distributed? It seems so, as the histogram is quite flat.

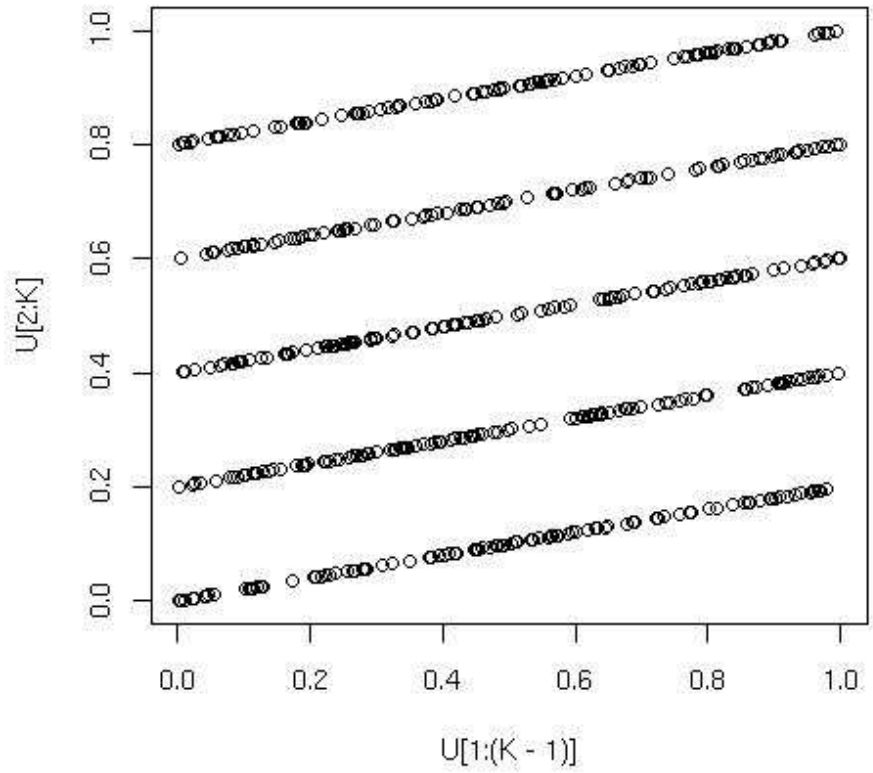
Does this really means, that the numbers U_i are good uniform deviates?

There is a question whether they are **independent**. To study this property we consider vectors built of U_i's:

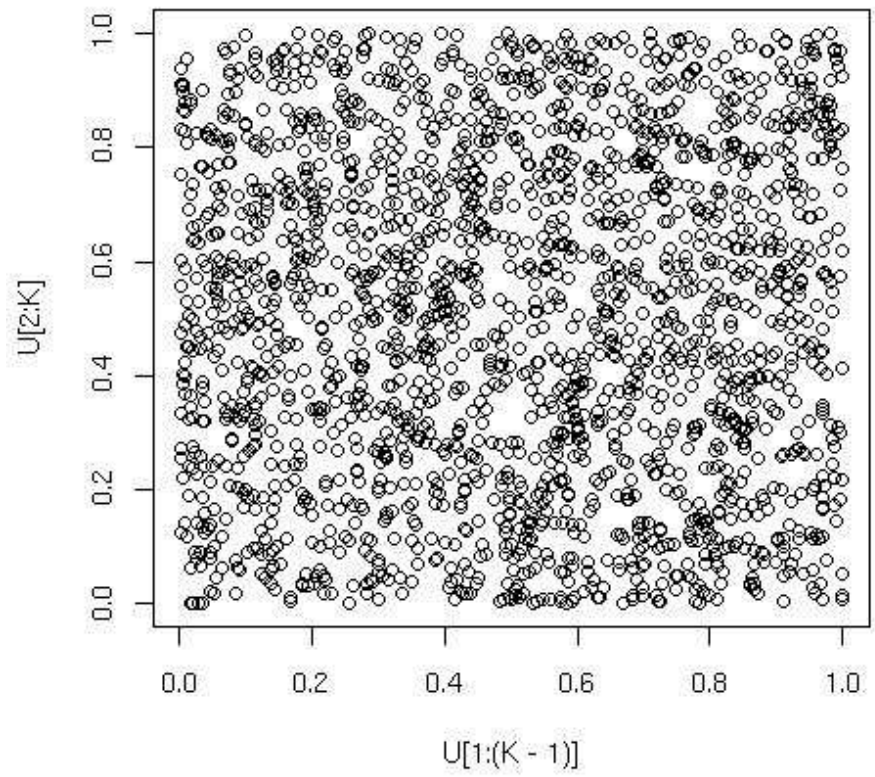
$$(U_i, U_{i+1}, \ldots, U_{i+m-1}) \in [0,1]^m$$

and analyze them with respect to distribution.
If U_i's are good uniform deviates, above random vectors are uniformly distributed over $[0,1]^m$.
It should be stressed that it is very difficult to assess random number generators!

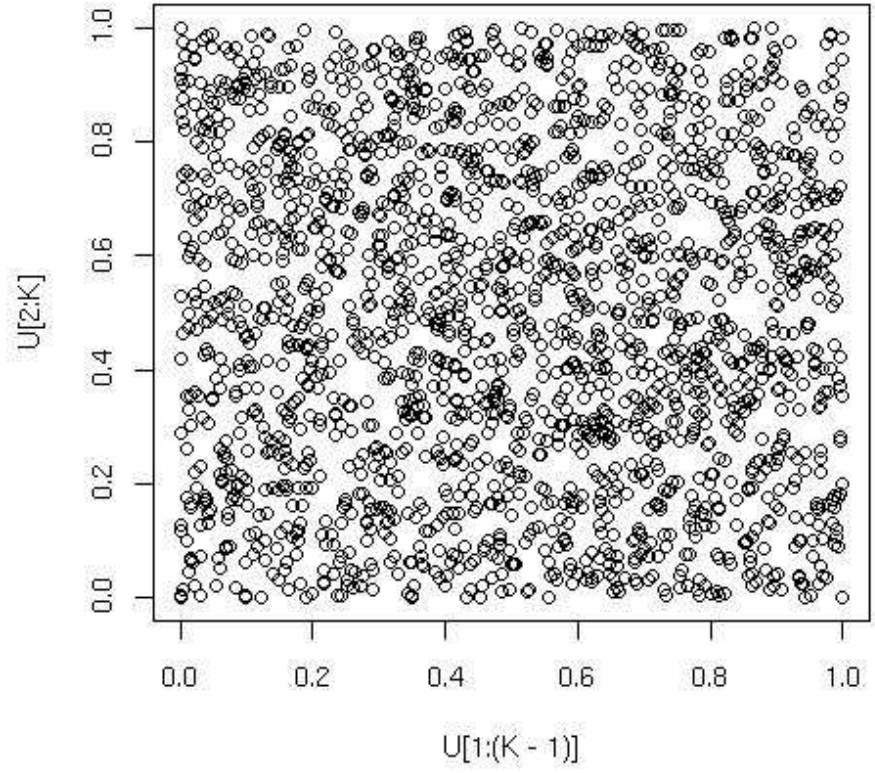
The plot of pairs (U_i, U_{i+1}) for the linear congruential generator with $a = 1229$, $b = 1$, $M = 2048$.



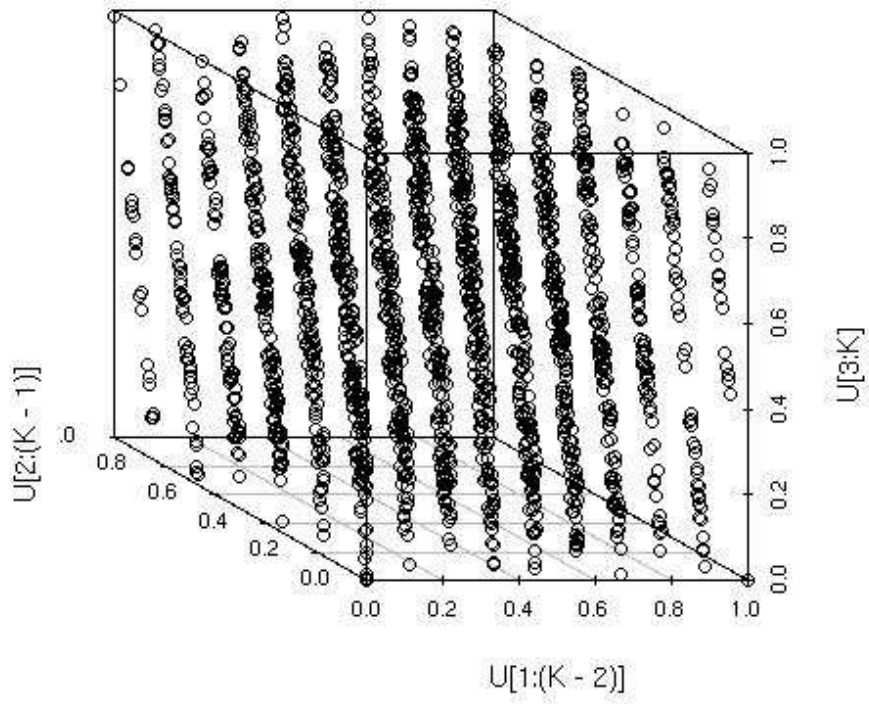
Good congruential generator with $a = 1597$, $b = 51749$ and $M = 244944$



Deceptively good congruential generator with $a = 2^{16} + 3$, $b = 0$, $M = 2^{31}$



Apparently not so good congruential generator with
$a = 2^{16} + 3$, $b = 0$, $M = 2^{31}$



In general Marsaglia showed that the m-tuples $(U_i, \ldots, U_{i+m-1})$ generated with linear congruential generators lie on relatively low number of hyperplanes in $\mathbb{R}^m$.

This is a major disadvantage of linear congruential generators.

An alternative way to generate uniform deviates is by using **Fibonacci generators**.

Fibonacci generators

The original Fibonacci recursion motivates the following general approach to generate pseudo-random numbers

$$s_i = s_{i-n} \text{ op } s_{i-k}.$$

Here $0 < k < n$ are the lags and **op** can be one of the following operators:

- $+$ addition $\mod M$,
- $-$ subtraction $\mod M$,
- $*$ multiplication $\mod M$.

To initialize (seed) these generators we have to use another generator to supply first n numbers $s_0, s_1, \ldots, s_{n-1}$. In addition, in subtraction generators we have to control that if $s_i < 0$ for some i the result has to be shifted $s_i := s_i + M$.

What are ”good generators”?

Good generators are those generators that pass a large number of statistical tests, see, e.g.,

P. L’Ecuyer and R. Simard – TestU01: A C Library for Empirical Testing of Random Number Generators, *ACM Transactions on Mathematical Software*, Vol. 33, article 22, 2007

TestU01 is a software library, implemented in C, and offering a collection of utilities for the empirical statistical testing of uniform random number generators.

These tests are freely accessible on the web page http://www.iro.umontreal.ca/ simardr/testu01/tu01.html

”Minimal Standard” generators

W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery – *Numerical Recipes in C* offers a number of portable random number generators, which passed all new theoretical tests, and have been used successfully.

The simplest of these generators, called *ran0*, is a standard congruential generator

$$s_{i+1} = a s_i \mod M,$$

with $a = 7^5 = 16807$ and $M = 2^{31} - 1$, is a basis for more advanced generators *ran1* and *ran2*. There are also better generators like *ran3* and *ran4*.
Of these generators only *ran3* possesses sufficiently good properties to be used in financial calculations.
It can be difficult or impossible to implement these generators directly in a high-level language, since usually integer arithmetics is limited to 32 bits.

To avoid this difficulty Schrage proposed an algorithm for multiplying two 32-bit integers modulo a 32-bit constant, without using any intermediates larger than 32 bits. Schrage's algorithm is based on an approximate factorization of M

$$M = aq + r, \text{ i.e. } q = [M/a], \; r = M \mod a.$$

If $r < q$, and $0 < z < M - 1$, it can be shown that both $a(z \mod q)$ and $r[z/q]$ lie in the range $[0, M - 1]$. Then

$$az \mod M = \begin{cases} a(z \mod q) - r[z/q], & \text{if it is } \geq 0, \\ a(z \mod q) - r[z/q] + M, & \text{otherwise.} \end{cases}$$

For *ran0* Schrage's algorithm is uses with the values $q = 127773$ and $r = 2836$.

The Mersenne Twister

The Mersenne Twister of Matsumoto and Nishimura which appeared in late 90-ties is now mostly used in financial simulations.
It has a period of $2^{19937} - 1$ and the correlation effect is not observed for this generator up to dimension 625.
Mersenne Twister is now implemented in most commercial packages. In particular, it is a standard generator in Matlab, Octave, R-project, S-plus.

Generation of uniform pseudo-random numbers

The generator of pseudo-random numbers with uniform distribution on interval $[0, 1]$ in Octave can be called by one of the commands:

```
rand (x)
rand (n, m)
rand ("state", v)
```

The version `rand (x)` returns `x`-dimensional square matrix of uniformly distributed pseudo-random numbers.
For two scalar arguments, `rand` takes them to be the number of rows and columns.

The state of the random number generator can be queried using the form

```
v = rand ("state")
```

This returns a column vector v of length 625. Later, the random number generator can be restored to the state v using the form

```
rand ("state", v)
```

The state vector may be also initialized from an arbitrary vector of length ≤ 625 for v. By default, the generator is initialized from `/dev/urandom` if it is available, otherwise from CPU time, wall clock time and the current fraction of a second.

`rand` includes a second random number generator, which was the previous generator used in Octave. If in some circumstances it is desirable to use the old generator, the keyword "seed" is used to specify that the old generator should be used.

The generator in such situation has to be initialized as in

```
rand ("seed", v)
```

which sets the seed of the generator to `v`.

It should be noted that most random number generators coming with C++ work fine, they are usually congruential generators, so you should be aware of their limitations.

Generation of non-uniformly distributed random deviates

- Idea 1: Discrete distributions
- Idea 2: Inversion of the distribution function
- Idea 3: Transformation of random variables

Idea 1: Discrete distributions

Coin toss distribution: $\mathbb{P}(X = 1) = 0.5$, $\mathbb{P}(X = 0) = 0.5$

(1) Generate $U \sim \mathcal{U}(0, 1)$
(2) If $U \leq 0.5$ then Z = 1, otherwise Z = 0

Z has a coin toss distribution.

Discrete distribution: $\mathbb{P}(X = a_i) = p_i$, $i = 1, 2, \ldots, n$

(1) Compute $c_k = \sum_{i=1}^{k} p_i$
(2) Generate $U \sim \mathcal{U}(0, 1)$
(3) Find smallest k such that $U \leq c_k$. Put $Z = a_k$

Z has a given discrete distribution.

Idea 2: Inversion of the distribution function

Proposition. *Let $U \sim \mathcal{U}(0,1)$ and F be a continuous and strictly increasing cumulative distribution function. Then $F^{-1}(U)$ is a sample of F.*

Theoretically this approach seems to be fine. The only thing we really need to do is to generate uniformly distributed random numbers.

Works well for: exponential distribution, uniform distribution on various intervals, Cauchy distribution.

Beasley-Springer-Moro algorithm for normal variate uses inversion of the distribution function with high accuracy (3×10^{-9}).
In interval $0.5 \leq y \leq 0.92$ the algorithm uses the formula

$$F^{-1}(y) \approx \frac{\sum_{n=0}^{3} a_n (y-0.5)^{2n+1}}{1+\sum_{n=0}^{3} b_n (y-0.5)^{2n}},$$

and for $y \geq 0.92$ the formula

$$F^{-1}(y) \approx \sum_{n=0}^{8} c_n \Big(\log\big(-\log(1-y)\big)\Big)^n.$$

Idea 3: Transformation of random variables

Proposition. *Let X be a random variable with density function f on the set $A = \{x \in \mathbb{R}^n | f(x) > 0\}$. Assume that the transformation $h : A \to B = h(A)$ is invertible and that the inverse h^{-1} is continuously differentiable. Then $Y = h(X)$ has the density*

$$y \mapsto f(h^{-1}(y)) \cdot \left| \det \left(\frac{dh^{-1}}{dy}(y) \right) \right|,$$

for all $y \in B$.

Multidimensional normal distribution $\mathcal{N}(\mu, \Sigma)$ on $\mathbb{R}^p$ has a density

$$f(x) = \frac{1}{(2\pi)^{p/2}} \frac{1}{(\det \Sigma)^{1/2}} \exp\Big(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu) \Big).$$

Here μ is a p-dimensional vector of expected values, and Σ is a $p \times p$-dimensional covariance matrix.

If $X \sim \mathcal{N}(\mu, \Sigma)$, then

- $X = (X_1, X_2, \ldots, X_p)$,
- $\mu = \mathbb{E}X = (\mathbb{E}X_1, \ldots, \mathbb{E}X_p)$
- $\Sigma = (\Sigma_{ij})_{i,j=1,\ldots,p}$ is a square matrix, where

$$\Sigma_{ij} = cov(X_i, X_j) = \mathbb{E}[(X_i - \mu_i)(X_j - \mu_j)]$$

- $\Sigma_{ij} = \Sigma_{ji}$,
- $\Sigma_{ii} = var(X_i)$

We are going to apply this result for the case where $A = [0,1]^2$ and $f(x) = 1$ for all $x \in A$ (i.e. we start with a two-dimensional uniformly distributed random variable) and choose the transformation

$$x \mapsto h(x) = \begin{pmatrix} \sqrt{-2\ln x_1}\cos(2\pi x_2), \\ \sqrt{-2\ln x_1}\sin(2\pi x_2). \end{pmatrix}$$

The inverse of this transformation is given by

$$y \mapsto h^{-1}(y) = \begin{pmatrix} \exp(-\|y\|^2/2), \\ \arctan(y_2/y_1)/2\pi. \end{pmatrix}$$

We compute the determinant of the derivative at $y = h(x)$ as follows:

$$\det\Big(\frac{dh^{-1}}{dy}(y)\Big) = -\frac{1}{2\pi}\exp\Big(-\frac{1}{2}(y_1^2+y_2^2)\Big)$$

Therefore, the density function of $Y = h(X)$, where $X \sim U([0,1]^2)$ equals

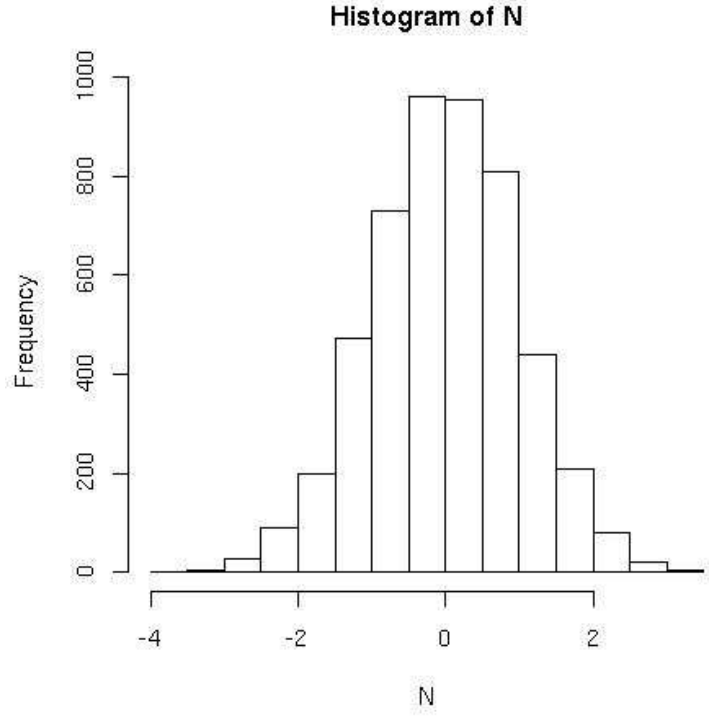
$$f\big(h^{-1}(y)\big)\cdot\Big|\det\Big(\frac{dh^{-1}}{dy}(y)\Big)\Big| = 1\cdot\frac{1}{2\pi}\exp\Big(-\frac{1}{2}(y_1^2+y_2^2)\Big).$$

This is obviously the density function of the two-dimensional standard normal distribution.
We therefore obtain that $h(X)$ is 2-dimensional standard normally distributed, whenever X is uniformly distributed on $[0,1]^2$.

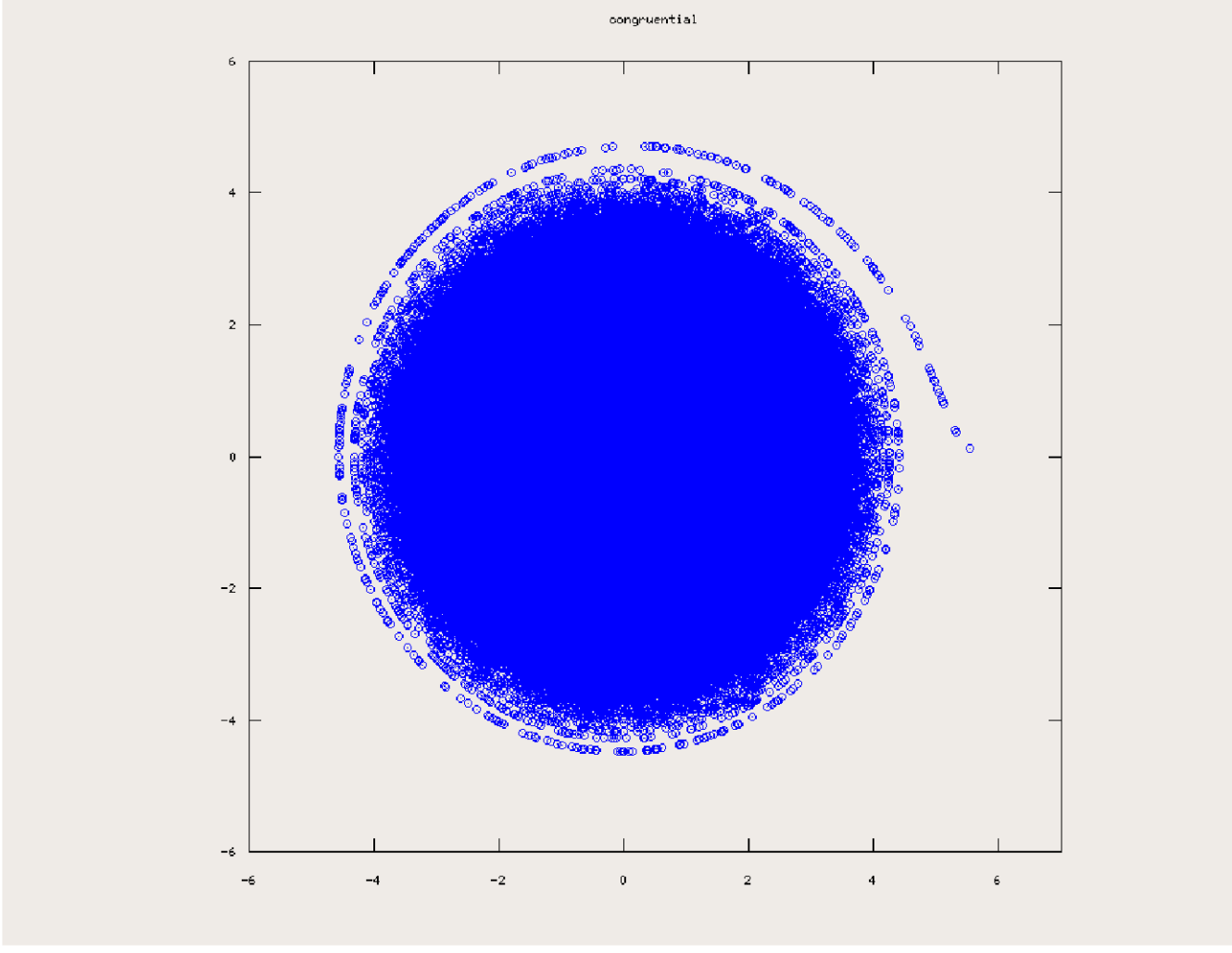
Box-Muller algorithm

Creates $Z \sim \mathcal{N}(0,1)$:

1. Generate $U_1 \sim \mathcal{U}[0,1]$ and $U_2 \sim \mathcal{U}[0,1]$,
2. $\theta = 2\pi U_2$, $\rho = \sqrt{-2\log U_1}$,
3. $Z_1 := \rho\cos\theta$ is a normal variate (as well as $Z_2 := \rho\sin\theta$).



Box-Muller with congruential generator



Naeve effect

In 1973, H. R. Naeve discovered an undesirable interaction between simple multiplicative congruential pseudo-random number generators and the Box-Muller algorithm. The pairs of points generated by the Box-Muller method fall into a small range (rectangle) around zero. The Naeve effect disappears for the polar method.
Unfortunately, number theorists suspect that effects similar to the Naeve effect may occur for other pseudo-random generators.
In summary, since there are highly accurate algorithms for the inverse cumulative normal probability function, use those rather than the Box-Muller algorithm.

Marsaglia algorithm

The Box-Muller algorithm has been improved by Marsaglia in a way that the use of trigonometric functions can be avoided. It is important, since computation of trigonometric functions is very **time-consuming**.

Algorithm: polar method (creates $Z \sim \mathcal{N}(0,1)$):

1. *repeat* generate $U_1, U_2 \sim \mathcal{U}[0,1]$;
   $V_1 = 2U_1 - 1$, $V_2 = 2U_2 - 1$;
   until $W := V_1^2 + V_2^2 < 1$.
2. $Z_1 := V_1 \sqrt{-2\ln(W)/W}$
   $Z_2 := V_2 \sqrt{-2\ln(W)/W}$
   are both normal variates.

Generation of $\mathcal{N}(\mu, \sigma^2)$ distributed pseudo-random variables

(1) Compute $Z \sim \mathcal{N}(0, 1)$

(2) $Z_1 = \mu + \sigma Z$

Z_1 is a pseudo-random number from a normal distribution $\mathcal{N}(\mu, \sigma^2)$.

How to generate a sample from a multidimensional normal distribution?

Theorem. *Let Z be a vector of p independent random variables each with a standard normal distribution $\mathcal{N}(0,1)$. There exists a matrix A such that*

$$\mu + AZ \sim \mathcal{N}(\mu, \Sigma).$$

Our aim is to find such a matrix A. We know how to generate a sequence of independent normally distributed random variables. Using matrix A we can transform it into a sequence of multidimensional normal variates.

The covariance matrix Σ is positive definite. One can show that there is exactly one lower-triangular matrix A with positive diagonal elements, s.t.

$$\Sigma = A \cdot A^T,$$

where A^T denotes a transpose of a matrix A.

This decomposition is called the **Cholesky decomposition**.

There are numerical methods how to compute the Cholesky decomposition of a positive definite matrix, but we don't discuss this here.

Algorithm for generation of $\mathcal{N}(\mu, \Sigma)$ distributed pseudo-random variables.

1. Calculate the Cholesky decomposition $AA^T = \Sigma$.

2. Calculate $Z \sim \mathcal{N}(0, I)$ componentwise by $Z_i \sim \mathcal{N}(0, 1)$, $i = 1, \ldots, n$, for instance with Marsaglia polar algorithm.

3. $\mu + AZ$ has the desired distribution $\sim \mathcal{N}(\mu, \Sigma)$.

Constant correlation Brownian motion

We call a process $W(t) = (W_1(t), \ldots, W_d(t))$ a standard d-dimensional Brownian motion when the coordinate processes $W_i(t)$ are standard one-dimensional Brownian motions with W_i and W_j independent for $i \neq j$.

Let μ be a vector in $\mathbb{R}^d$ and Σ an $d \times d$ positive definite matrix. We call a process X a Brownian motion with drift μ and covariance Σ if X has continuous sample paths and independent increments with

$$X(t+s) - X(s) \sim \mathcal{N}(t\mu, t\Sigma).$$

Process X is called a constant correlation Brownian motion (which means that Σ is constant).

Simulation

For Monte Carlo simulations with correlated Brownian motion we have to know how to generate samples from $\mathcal{N}(\delta t\mu, \delta t\Sigma)$.

Since δt is a known constant we can reduce the problem to generating samples from distribution $\mathcal{N}(\mu, \Sigma)$.

Methods of sample generation:

- Cholesky decomposition (discussed earlier),
- PCA construction.

Cholesky decomposition

Algorithm for generation of $\mathcal{N}(\mu, \Sigma)$ distributed pseudo-random variables:

1. Calculate the Cholesky decomposition $AA^T = \Sigma$.
2. Calculate $Z \sim \mathcal{N}(0, I)$ componentwise by $Z_i \sim \mathcal{N}(0, 1)$, $i = 1, \ldots, n$.
3. $\mu + AZ$ has the desired distribution $\sim \mathcal{N}(\mu, \Sigma)$.

PCA construction

Principal Component Analysis (PCA) is the method of analyzing spectrum of the covariance matrix.

Since Σ is symmetric positive definite matrix, it has d positive eigenvalues and d eigenvectors which span the space $\mathbb{R}^d$. In addition the following formula holds

$$\Sigma = \Gamma \Lambda \Gamma^T,$$

where Γ is the matrix of d eigenvectors of Σ and Λ is the diagonal matrix of eigenvalues of Σ.

Theorem. *Let Z be a vector of d independent random variables each with a standard normal distribution $\mathcal{N}(0,1)$. Then*

$$\mu + \Gamma \Lambda^{1/2} Z \sim \mathcal{N}(\mu, \Sigma),$$

where $\Sigma = \Gamma \Lambda \Gamma^T$ is the spectral decomposition of matrix Σ.

Let us observe that due to positivity of eigenvalues $\Lambda^{1/2}$ is well defined.

In general Cholesky decomposition and PCA construction do not give the same results.

Indeed

$$\Sigma = AA^T = \Gamma\Lambda\Gamma^T = \Gamma\Lambda^{1/2}\Lambda^{1/2}\Gamma^T$$

and

$$\Sigma(A^T)^{-1} = A = \Gamma\Lambda^{1/2}\Lambda^{1/2}\Gamma^T(A^T)^{-1}$$

holds. But in general

$$\Lambda^{1/2}\Gamma^T(A^T)^{-1} \neq I,$$

where I is the identity matrix. Hence

$$AZ \neq \Gamma\Lambda^{1/2}Z$$

even though both sides generate correlated deviates.

Computer implementation

In a Cholesky factorization matrix A is lower triangular. It makes calculations of AZ particularly convenient because it reduces the calculations complexity by a factor of 2 compared to the multiplication of Z by a full matrix $\Gamma\Lambda^{1/2}$. In addition error propagates much slower in Cholesky factorization.

Cholesky factorization is more suitable to numerical calculations than PCA.

Why use PCA?

The eigenvalues and eigenvectors of a covariance matrix have statistical interpretation that is some times useful. Examples of such a usefulness are in some variance reduction methods.