Axel Kind
Assistant Professor
Corporate Finance Division
Department of Economics and
Business Administration (WWZ)
University of Basel
Spring 2012

Quantitative Security Analysis, Course-Nr. 21996

Solution Exercise Set 4: Monte-Carlo Simulation

1 Random Number Generation

Monte Carlo simulations are built on sequences of apparently random numbers. However, these random numbers, generated by a computer only behave like random variables from the point of view of the user. Actually, these artificial random numbers are generated by completely deterministic algorithms. That is why they are called *pseudorandom* numbers. Therefore, a random number generator is often called a *pseudorandom number generator* because it only mimics randomness.
The desired properties for a sequence of random numbers u_1, u_2,...,u_n are

(i) u_i is uniformly distributed between 0 and 1, i.e. $u_i \sim U[0,1] \forall\ i$;

(ii) u_i's are mutually independent.

Whereas property (i) is just an arbitrary normalization because uniform variables of any interval can be transformed in other distributions, property (ii) is much more important. This property implies that any random number u_i should be unpredictable from u_1, u_2,...,u_n, which is much more difficult to satisfy. The goal is to have a random number generator that is consistent with the properties (i) and (ii).
The following aspects are of importance when generating random numbers:

- Period length: Any random number sequence will eventually repeat itself. Other things equal, a generator with a longer sequence as output are preferred.
- Reproducibility: Often, it is important to repeat a simulation using exactly the same inputs. By setting the same initial seed for a linear congruental generator (introduced in Section 1.1.1) this is easily accomplished.

- Speed: Pseudorandom number generator may be called millions of times. For that reason, the algorithm must be fast.
- Portability: A random number generating algorithm should produce identical sequences of values on all computing platforms.
- Randomness: The random numbers should be statistically indistinguishable from samples of a random variable with an ideal distribution of interest (e.g. normal distribution).

Thus, in a first step, uniformly distributed (pseudo)-random numbers in the intervall $[0,1]$ are generated

$$u_i \sim U[0,1].$$

In a second step, these uniform random numbers are transformed to a distribution of choice with a function F, e.g. the normal distribution

$$Z := F(u) \sim N(0,1)$$

1.1 Uniformly Distributed Pseudorandom Numbers

In this section, we will consider different methods for random number generation. The methods covered are:

(i) Linear congruental generators

(ii) Fibonacci generators

(iii) Combining generators

1.1.1 Linear Congruental Method (LCG)

The *linear congruental generator*, introduced by Lehmer (1951), constitutes a simple algorithm for generating uniform random numbers in the intervall $[0,1]$. For some deterministic functions f and g the LCG is of the form

$$x_{i+1} = f(x_i), \qquad u_{i+1} = g(x_{i+1}) \tag{1}$$

The general form is given by

$$\begin{aligned} &\text{for } i = 1, 2, ... \\ x_{i+1} &= (ax_i + b) \bmod m \\ u_{i+1} &= x_{i+1}/m \end{aligned} \tag{2}$$

The parameters a, m and b must be integers. The starting value x_0 is called *seed* and must lie in the intervall $1 < x_0 < m - 1$. Whereas this form is called *mixed* LCG, the form given by $x_{i+1} = ax_i \bmod m$ is called a *pure* or *multiplicative* LCG. The obtained series of uniform random variables belong to the set $\{0, 1, 2, m - 1\}$.

A LCG that produces the maximum sequence of m distinct values before repeating itself (i.e. once a number is repeated, the whole sequence is repeated) is called a *full-period* LCG. To ensure a sequence of m distinct values, the integers a, m and b have to be chosen in a appropriate way. The necessary conditions are

(i) b is relatively prime with m (i.e. their only common divisor is 1)

(ii) $a - 1$ is a multiple of any prime number that divides m

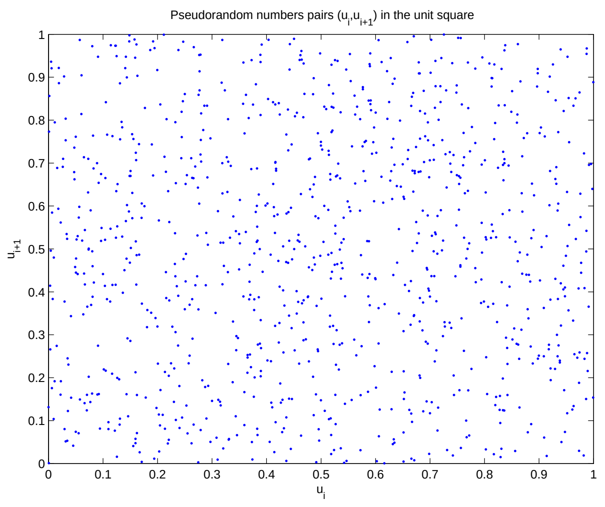
(iii) $a - 1$ is divisible by 4 if m is

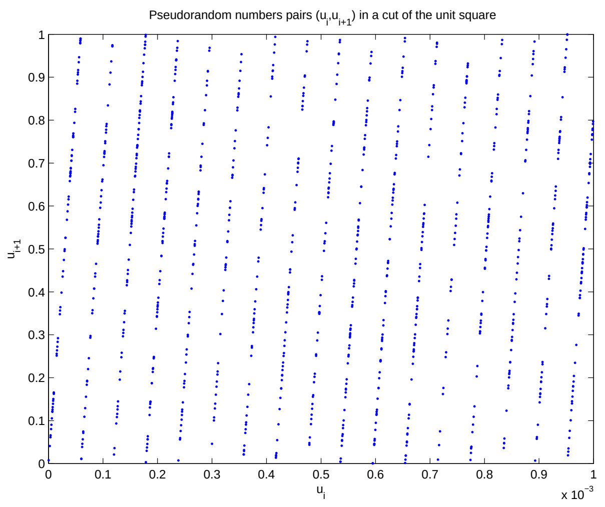
Exercises LCG Method

1. Generate uniformly distributed random numbers by applying the LCG-method given in Equation 2 with parameter values $a = 16807$, $b = 0$, $m = 2^{31} - 1$. These are the same parameters as used by MATLAB's *multiplicative* congruential generator. If the random numbers are independent and uniformly distributed, then (u_i, u_{i+1}) will be uniformly distributed over the unit square, (u_i, u_{i+1}, u_{i+2}), over the unit cube $[0,1]^3$, and $(u_i, u_{i+1}, ..., u_{i+d})$ over the d-tuple $[0,1]^d$.
   Show in a first plot the consecutive pseudorandom number pairs (u_i, u_{i+1}) in the unit square with the number of random values of $n = 1000$. In as second plot, show the pseudorandom number pairs (u_i, u_{i+1}) but only over the intervall $[0, 0.001]$ on the x-axis with $n = 1000000$. What do you observe?

```
function u = LCGFun(n,seed,a,b,m)
%
% u = LCGFun(n,seed,a,b,m)
%
% This function generates uniformly distributed random variables with the
% linear congruential generator.
%
% INPUT:      n:              number of uniformly distributed random
%                             variables to be generated
%             seed:           starting value
%             a, b, and m:    integer values for LCG, where m denotes
%                             the modulus
%
% OUTPUT:     u:              vector (n*1) of uniformly distributed random
%                             numbers
%
% EXAMPLE:  u = LCGFun(1000,1,16807,0,2^31-1);

x = nan(n,1);
u = nan(n-1,1);
x(1) = seed;
% Random number generation
for i = 2:n
    x(i) = mod(a*x(i-1)+b,m);
    u(i-1) = x(i)/m;
end
```

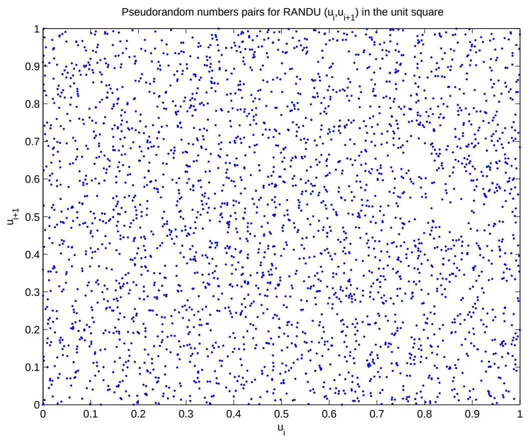
```
% This m-script plots the LCG uniform random numbers in the unit square and
% in a small window of the unit square
close all
%% Parameter values for LCG
a = 16807;
b = 0;
m = 2^31-1;
seed = 1;

%% Plot random number pairs in the unit square
% Number of uniform random variables
n = 1000;
u = LCGFun(n,seed,a,b,m);
figure;
plot(u(1:n-2),u(2:n-1),'.')
title('Pseudorandom numbers pairs (u_{i},u_{i+1}) in the unit square');
xlabel('u_{i}')
ylabel('u_{i+1}');
saveas(gca,'LCGCode.eps','psc2');

%% Plot random number pairs in a small window of the unit square
% Number of uniform random variables
n = 1000000;
u = LCGFun(n,seed,a,b,m);
figure;
plot(u(1:n-2),u(2:n-1),'.')
title('Pseudorandom numbers pairs (u_{i},u_{i+1}) in a cut of the unit square');
xlabel('u_{i}')
ylabel('u_{i+1}');
ax = axis;
axis([ax(1) 0.001 ax(3) ax(4)])
saveas(gca,'LCGCodeNarrowRange.eps','psc2');
```

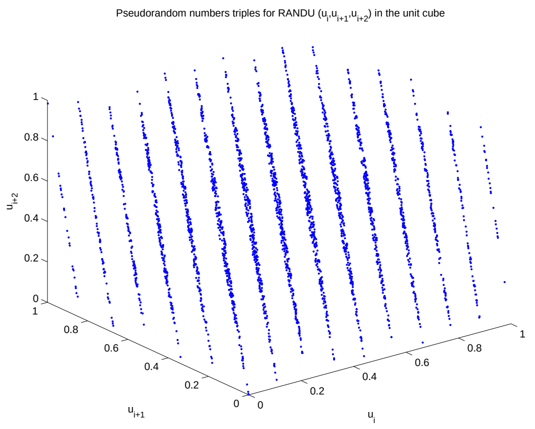




2. In the 1960's, RANDU, distributed by IBM, was a popular random generator employing the *multiplicative* LCG with $a = 65539$, $b = 0$, and $m = 2^{31}$. First, plot the pseudorandom number pairs (u_i, u_{i+1}) in the unit square with $n = 3000$. What do you think, do these numbers mimic randomness well? Second, plot (u_i, u_{i+1}, u_{i+2}) over the unit cube $[0,1]^3$. What do you think now of RANDU?

```
% This m-script plots the LCG uniform random numbers for RANDU in the unit
% square and in a small window of the unit square
close all
%% Parameter values for LCG
a = 65539;
b = 0;
m = 2^31;
seed = 1;
n = 3000;
%% Plot random number pairs in the unit square
u = LCGFun(n,seed,a,b,m);
figure;
plot(u(1:n-2),u(2:n-1),'.')
title('Pseudorandom numbers pairs for RANDU (u_{i},u_{i+1}) in the unit square');
xlabel('u_{i}')
ylabel('u_{i+1}');
saveas(gca,'LCGRANDUUnitSquare.eps','psc2');
%% Plot random number triples in a cut of the unit square
u = LCGFun(n,seed,a,b,m);
figure;
plot3(u(1:n-3),u(2:n-2),u(3:n-1),'.')
title('Pseudorandom numbers triples for RANDU (u_{i},u_{i+1},u_{i+2}) in the unit cube');
xlabel('u_{i}')
ylabel('u_{i+1}');
zlabel('u_{i+2}');
saveas(gca,'LCGRANDUUnitCubic.eps','psc2');
```





Marsaglia (1968) shows that all multiplicative congruential random number generators have a defect that makes them unsuitable for many Monte Carlo problems. This is due to the fact that in a cube of d dimensions all the points will lie in a relatively small number of parallel hyperplanes. This is demonstrated in the second plot that shows that the point triples are all located on a small number of 15 hyperplanes in the unit cube $[0,1]^3$. Because of Marsaglia's finding, the LCG method on its own is not a suitable generator of random numbers.

1.1.2 Fibonacci Generators

Further methods to generate random numbers are Fibonacci generators, which belong to the group of congruential generators as well. The idea of these generators is to use the Fibonacci sequence. An algorithm for a simple Fibonacci generator is given below

$$\begin{aligned} \text{for } i &= 2, 3, ... \\ x_{i+1} &= (x_{i-1} + x_{i-2}) \bmod m \\ u_i &= x_i/m \end{aligned} \tag{3}$$

Exercise Fibonacci Generators

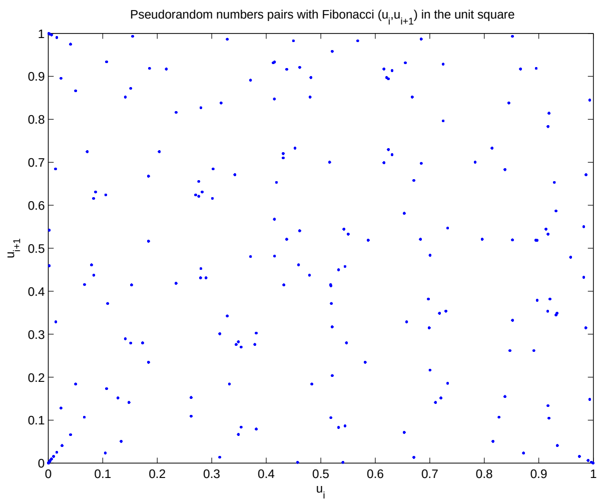
Implement the algorithm for the simple Fibonacci generator with $m = 2179$, $n = 3000$, and $x_1 = x_2 = 1$ and plot the random number pairs (u_i, u_{i+1}) in the unit square. You will observe less than 3000 distinct values in your plot. How many distinct values are generated by this algorithm?

```
function u = Fibonacci(n,seed1,seed2,m)
%
% u = Fibonacci(n,seed1,seed2,m)
%
% This function generates uniformly distributed random variables with the
% linear congruential generator.
%
% INPUT:            n:              number of uniformly distributed random
%                                   variables to be generated
%                   seed1,seed2:    initial seeds
%                   m:              modulus
%
% OUTPUT:           u:      vector (nx1) of uniformly distributed random
%                           numbers
%
% EXAMPLE:  u = Fibonacci(2000,1,1,2179)

x = nan(n,1);
u = nan(n,1);
x(1) = seed1;
x(2) = seed2;
u(1) = x(1)/m;
u(2) = x(2)/m;
for i = 3:n
    x(i) = mod(x(i-1)+x(i-2),m);
    u(i) = x(i)/m;
end
```

```
% Plot Fibonacci/number of distinct values
clc
close all
% Parameters
m = 2179;
n = 3000;
seed1 = 1;
seed2 = 1;
% Random number generation
u = Fibonacci(n,seed1,seed2,m);
% Plot Fibonacci
figure;
plot(u(1:n-2),u(2:n-1),'.');
title('Pseudorandom numbers pairs with Fibonacci (u_{i},u_{i+1}) in the unit square');
xlabel('u_{i}')
ylabel('u_{i+1}');
saveas(gca,'FibonacciUnitSquare.eps','psc2');
% Number of distinct values
nbrDistinctValues = length(unique(u));

fprintf('Number of distinct values: %1.0f\n',nbrDistinctValues);
```

```
Number of distinct values: 144
```

For that reason, we observe only 144 instead of 3000 points in the unit square.



1.1.3 Combining Generators

One way to enhance the linear congruential generators is to combine two or more LCG's through summation. A combination of generators allows to generate much longer sequences than any sequence generated by one of its components. Assume we have J generators, the jth having parameters a_j, m_j:

$$x_{j,i+1} = a_j x_{j,i} \bmod m_j, \qquad u_{j,i+1} = x_{j,i+1}/m_j \tag{4}$$

As a simple example consider two multiplicative generators to generate the sequences x_1, x_2 with moduli m_1 and m_2 where $m_1 > m_2$. This sequences can now be added or subtracted and we are able to obtain an enhanced sequence of uniformly distributed pseudorandom numbers with the following generator:

$$x_i = (x_{1,i} + x_{2,i}) \bmod m_1, \qquad u_i = x_i/m_1 \tag{5}$$

Another possibility to enhance the basic linear congruential generator uses a high-order recursion called *multiple recursive generator* (MRG):

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + ... + a_k x_{i-k}) \bmod m, \qquad u_i = x_i/m \tag{6}$$

The seed for MRG is given by the initial values for x_{k-1}, x_{k-2}, ..., x_0.

Exercise Combining Generators

Program an m-function to handle combinations of several LCG's as given in Equation 4 and 5. Then, generate uniform random numbers with a combination of two LCG with $a_1 = 40014$, $a_2 = 40692$, $b_1 = b_2 = 0$, $m_1 = 2147483563$, and $m_2 = 2147483399$.

```
function u = CombinedGenerator(a,m,seed,n)
%
% u = CombinedGenerator(a,m,seed,n)
%
% This function generates uniformly distributed random variables with a
% combination of several linear congruential generator.
%
% INPUT:            a:      vector of a's
%                   m:      vector of moduli
%                   seed:   vector of seeds
%                   n:      number of uniformly distributed numbers
%
% OUTPUT:           u:      vector (n*1) of uniformly distributed random
%                           numbers
%
% EXAMPLE:  u = CombinedGenerator([40014;40692],[2147483563;2147483399],[4;1],10000)

% Parameters
nbrGen = length(a);
x = nan(n,nbrGen);
% Assign seed values
for j = 1:nbrGen
    x(1,j) = seed(j);
end
% Get the largest modulus
maxModulus = max(m);

% Calculate the x_j's for the x
for i = 2:n
    for j = 1:nbrGen
        x(i,j) = mod(a(j)*x(i-1,j),m(j));
    end
end
x = mod(sum(x,2),maxModulus);
% Calculate the unifomly distributed numbers
u = x/maxModulus;
```

2 Generating Univariate Normals

Now, assume that we have obtained a sample of independent uniformly distributed random numbers in the interval $[0, 1]$. Mostly, we will need the normal distribution to generate random sample paths in Monte Carlo option pricing. Therefore, methods are required that transform the uniformly distributed random numbers into samples of the normal distribution. Normal random number generators can be classified into four basic categories:

(i) cumulative density function inversion,

(ii) transformation,

(iii) rejection, and

(iv) recursive methods.

Alternatively, we can classify the methods as *exact* or *approximate*. Whereas exact methods would produce perfect Gaussian random numbers if implemented in an ideal environment, approximate methods produce outputs that are approximately normally distributed even if the arithmetic applied is perfect. The first algorithm we will look at is the *Box-Muller algorithm* that belongs to the basic category of transformation methods and is classified as exact. The second algorithm is related to the Box-Muller method and is called the *Polar algorithm*. This algorithm belongs to the rejection methods and is an exact method as well.

2.1 Box-Muller Method

By applying this method two standard normal distributed numbers, z_1 and z_2 are generates from two uniform random numbers u_1, u_2 by using the following calculations:

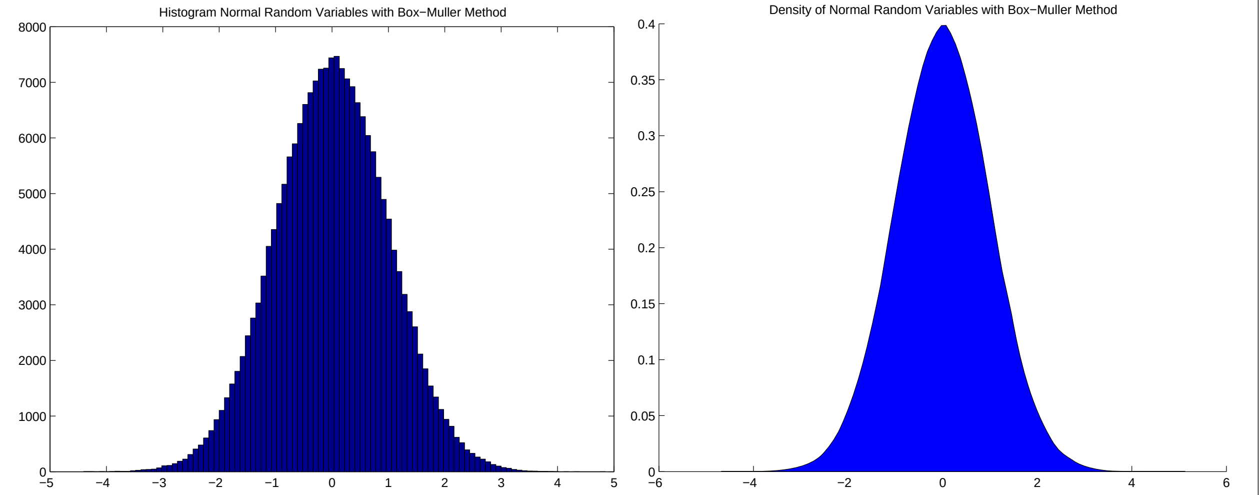
$$z_1 = \sqrt{-2ln(u_1)cos(2\pi u_2)} \quad \text{and} \quad z_2 = \sqrt{-2ln(u_1)sin(2\pi u_2)} \tag{7}$$

Exercise Box-Muller Method

Generate two sequences u_1 and u_2 each of $n = 100'000$ uniformly distributed random variables by using the LCG-method with $m = 2^{31} - 1$, $a = 16807$, and $b = 0$. For the first sequence use seed 1 and for the second 23324. Subsequently, transform these uniform variables into normal distributed random numbers, z_1 and z_2, by applying Equation 7. Merge z_1 and z_2 to z and plot a histogram with 100 bins by using `hist(z,100)`. In a second plot, show the probability density function of the z-variables by applying `ksdensity(z)`. Furthermore, perform a *one-sample Kolmogorov-Smirnov test* (`kstest(z)`) and a *Jarque-Bera test* (`jbtest(z)`) to compare the values in the data vector z to a (standard-) normal distribution.

```
function [z1 z2] = BoxMuller(u1,u2)
%
% [z1 z2] = BoxMuller(u1,u2)
%
% This function transforms uniformly distributed random numbers into
% standard normal distributed variables with the Box-Muller method.
%
% INPUT:        u1,u2: two vectors of equal length of uniform distributed random variables
%
% OUTPUT:       z1,z2: two vectors of standard normal distributed random variables

r = -2*log(u1);
v = 2*pi.*u2;
z1 = sqrt(r).*cos(v);
z2 = sqrt(r).*sin(v);


function TestNormalDistribution(z)
% Test on normal distribution
clc
% One-sample Kolmogorov-Smirnov test
[h,p] = kstest(z);
fprintf('One-sample Kolmogorov-Smirnov test\n');
if h == 0
    fprintf('h = %1.0f (p-value=%1.4f): The standard normal assumption cannot be rejected for z\n',h,p);
else
    fprintf('h = %1.0f: The standard normal assumption is rejected for z\n',h);
end
% Jarque-Bera test
fprintf('Jarque-Bera test\n');
h = jbtest(z);
if h == 0
    fprintf('h = %1.0f: The standard normal assumption cannot be rejected for z at the 5%% level.\n',h);
else
    fprintf('h = %1.0f: The standard normal assumption is rejected for z at the 5%% level\n',h);
end

% Box-Muller
% - histogram plot
% - densitiy plot
% - normality test
close all
clc
n = 100000;
u1 = LCGFun(n,1,16807,0,2^31-1);
u2 = LCGFun(n,1,23324,0,2^31-1);
[z1 z2] = BoxMuller(u1,u2);
% Merge the two vectors
z = [z1;z2];
% Create histogram of normal random numbers
figure;
hist(z,100);
title('Normal Random Variables with Box-Muller Method');
saveas(gca,'histogramBoxMuller.eps','psc2');
figure;
% Create density of normal random numbers
[f,xi] = ksdensity(z);
hold on
fill(xi,f,'b')
title('Density of Normal Random Variables with Box-Muller Method');
hold off
saveas(gca,'densityBoxMuller.eps','psc2');

% Test on normal distribution
TestNormalDistribution(z)
```

```
One-sample Kolmogorov-Smirnov test
h = 0 (p-value=0.8892): The standard normal assumption cannot be rejected for z
Jarque-Bera test
h = 0: The standard normal assumption cannot be rejected for z at the 5% level.
```



By employing the built-in function `randn(m,n)`[1] in MATLAB, we obtain standard normally distributed numbers as achieved in the previous exercise. The command `z=randn(1000000,1)` generates 1 Mio. standard normally distributed random numbers. You obtain the histogram by typing the command `hist(z,100)` in the command window.

2.2 Polar Method

The Box-Muller methods calls *sqrt*, *log*, *cos* and *sin* for creating two normal distributed numbers. Marsaglia reduced the computing time by avoiding the evaluation of the sine and cosine function. This method, called the *(Marsaglia)-polar algorithm*[2] differs from the basic method in employing acceptance-rejection sampling in the unit disc and then transforms these points to normal variables. Therefore, there is no need to use trigonometric functions. The procedure is as follows:

a. Generate u_1, $u_2 \sim U[0,1]$.

b. Calculate $v_1 = 2u_1 - 1$ and $v_2 = 2u_2 - 1$.

c. If $V = v_1^2 + v_2^2 \leq 1$ then calculate $z_1 = u_1\sqrt{-2\frac{ln(V)}{V}}$ and $z_2 = u_1\sqrt{-2\frac{ln(V)}{V}}$

The method's algorithm involves finding a random point in the unit circle by generating uniformly distributed points in the unit square $[0,1]^2$ and rejecting any points

[1]Beginning with MATLAB 5, the *ziggurat* algorithm is used by `randn`. This algorithm belongs to the rejection methods and is exact.

[2]Older versions of MATLAB (before MATLAB 5) used the polar algorithm for the `randn(m,n)` function.

outside of the circle. This algorithm ensures accurate tails in the distribution. However, despite the improvement relative to the Box-Muller method, this algorithm is moderately expensive due to the rejection of about 21% of the uniform numbers and the calculation of the square root and the logarithm.

Exercise Polar Method

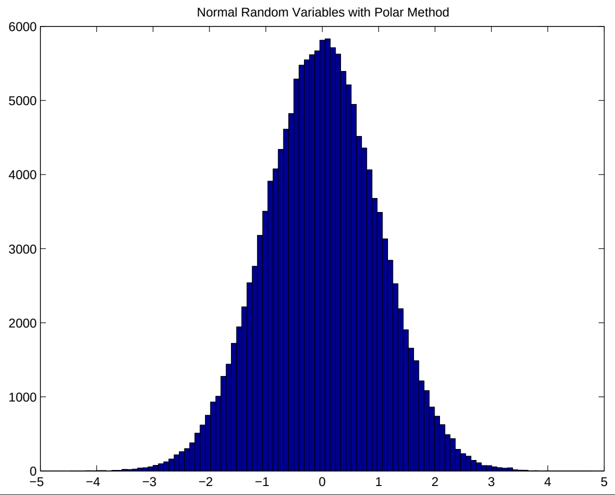
1. Implement the Polar-algorithm. Use the MATLAB built-in function `rand(m,n)`[3] to generate $n = 100'000$ uniformly distributed random numbers for the two sequences u_1, u_2 inside the *Polar*-function. As input variable for your function, you only need the number of uniformly distributed variables for the sequences u_1, u_2. Plot a histogram with 100 bins and test the data on normality with the Kolmogorov-Smirnov and the Jarque-Bera test. Calculate the number of normal distributed random variables generated by this algorithm from the uniform random numbers.

```
function z = Polar(n)
%
% z = Polar(n)
%
% This function transforms uniformly distributed random numbers (generated
% by the built-in-function rand) into standard normal distributed variables
% with the Polar method.
%
% INPUT:        n: number of uniform distributed random variables in each
%                  sequence of u
%
% OUTPUT:       z: vector of standard normal distributed random variables

% Use rand to generate unifomly distributed random numbers
u1 = rand(n,1);
u2 = rand(n,1);

v1 = 2*u1 - 1;
v2 = 2*u2 - 1;

V = (v1.*v1)+(v2.*v2);

index = V <= 1;

V = V(index);
z1 = v1(index).*sqrt(-2*log(V)./V);
z2 = v2(index).*sqrt(-2*log(V)./V);

z = [z1;z2];
```

[3]To observe the default random number algorithm that is used in MATLAB, you can type `defaultStream=RandStream.getDefaultStream` and `get(defaultStream)` in the command window. The default generator used by default stream at MATLAB 7.8.0 startup is *Mersenne twister* (Keyword: mt19937ar). Furthermore, we observe that the method to transform uniform random variables to random variables of any common probability distributions (e.g. normal and exponential) is the *ziggurat* algorithm proposed by Marsaglia and Tsang, which is a rejection sampling algorithm. This algorithm needs only multiplications, i.e. no square root and logarithm have to be calculated anymore.

```
% Polar method
% - histogram plot
% - densitiy plot
% - normality test
close all

n = 100000;
z = Polar(n);
lenZ = length(z);
nbrOfNormals = lenZ/(2*n);
fprintf('The Polar-algorithm returns %1.2f*2n random numbers.\n',nbrOfNormals);
% Create histogram of normal random numbers
figure;
hist(z,100);
title('Normal Random Variables with Polar Method');
saveas(gca,'histogramPolar.eps','psc2');

% Test on normal distribution
TestNormalDistribution(z)
```

```
The Polar-algorithm returns 0.79*2n random numbers.
One-sample Kolmogorov-Smirnov test
h = 0 (p-value=0.8281): The standard normal assumption cannot be rejected for z
Jarque-Bera test
h = 0: The standard normal assumption cannot be rejected for z at the 5% level.
```


Normal Random Variables with Polar Method