LECTURE 8: Random numbers

October 31, 2011

1 Introduction

1.1 Background

What is a *random* number? Is 58935 "more random" than 12345?

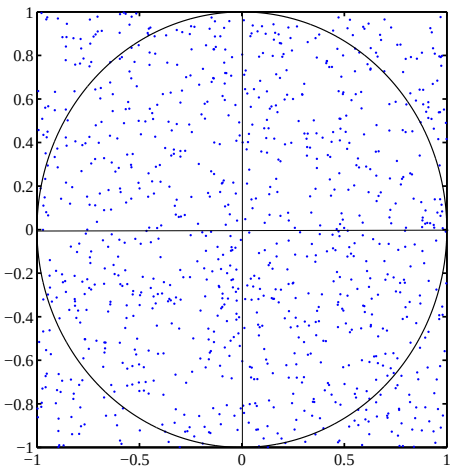


Figure 1: Randomly distributed points.

Randomness (uncorrelatedness) is an *asymptotic property* of a sequence of numbers $\{x_1, x_2, \ldots, x_n\}$ for $n >> 1$. Thus we cannot ask if 58935 is "more random" than 12345, since we cannot associate randomness with individual numbers.

There is no *unique* definition of randomness, but in principle we could try to check that all cross-correlations approach zero as $n \to \infty$. But this is a hopeless task (it involves way too much computation). In practice, randomness is checked by various random number tests.

Long sequences of random numbers are needed in numerous applications, in particular in statistical physics and applied mathematics. Examples of methods which utilize random numbers are Monte Carlo simulation techniques, stochastic optimization and cryptography. All these methods require fast and reliable random number sources.

Random numbers can be obtained from:

1. **Physical sources.** (Real random numbers.)
   Many physical processes are random by nature and such processes can be used to produce random numbers. Examples are noise in semiconductor devices or throwing a dice.

2. **Deterministic algorithms.** (Pseudorandom numbers.)
   In practice, random numbers are generated by pseudorandom number generators (RNG's). These are deterministic algorithms, and consequently the generated numbers are only "pseudo-random" and have their limitations. But for many applications, pseudorandom numbers can be successfully used to *approximate* real random numbers.

Properties of good random number generators

From now on, we use the term "random number" to mean pseudorandom numbers.

In practice, random number generator algorithms are implemented in the computer and the numbers are generated while a computer program is running. RNG algorithms produce *uniform pseudorandom numbers* $x_i \in (0, 1]$. These uniformly distributed random numbers can be used further to produce different distributions of random numbers (we will return to this later in this chapter).

A good random number generator has the following properties:

1. The numbers must have a correct *distribution*. In simulations, it is important that the sequence of random numbers is *uncorrelated* (i.e. numbers in the sequence are not related in any way). In numerical integration, it is important that the distribution is flat.

2. The sequence of random numbers must have a *long period*. All random number generators will repeat the same sequence of numbers eventually, but it is important that the sequence is sufficiently long.

3. The sequences should be *reproducible*. Often it is necessary to test the effect of certain simulation parameters and the exact same sequence of random numbers should be used to run many different simulation runs. It must also be possible to stop a simulation run and then recontinue the same run which means that the state of the RNG must be put in memory.

4. The RNG should be easy to *export* from one computing environment to another.

5. The RNG must be *fast*. Large amounts of random numbers are needed in simulations.

6. The algorithm must be *parallelizable*.

2 Random number generators

Most RNG's use integer arithmetic and the real numbers in $(0,1]$ are produced by scaling. In the following, some of the most widely used RNG's are introduced.

2.1 Linear congruential generators

Linear congruential generators (LCG) are based on the integer recursion relation

$$x_{i+1} = (ax_i + b) \bmod m$$

where integers a, b and m are constants. This generates a sequence $x_1, x_2, \ldots$ of random integers which are distributed between 0 and $m-1$ (if $b > 0$) or between 1 and $m-1$ (if $b = 0$). Each x_i is scaled to the interval $(0,1)$ by dividing by m. The parameter m is usually equal or nearly equal to the largest integer of the computer. This determines the period P of the LCG. It applies that $P \leq m$. In order to start the sequence of random numbers, a *seed* x_0 must be provided as input.

Linear congruential generators are classified into *mixed* ($b > 0$) and *multiplicative* types, which are denoted by LCG(a,b,m) and MLCG(a,m) respectively.

GGL

GGL is a uniform random number generator based on the linear congruential method. It is denoted by MLCG(16807,$2^{31}-1$) or

$$x_{i+1} = (16807 x_i) \bmod (2^{31} - 1)$$

This has been a particularly popular generator. It is available in some numerical software packages such as subroutine `RAND` in Matlab. GGL is a simple and fast RNG, but it suffers from a short period of $2^{31} - 1 \approx 2 \times 10^9$ steps.

RAND

RAND is a uniform random number generator which uses a linear congruential algorithm with a period of 2^{32}. The generator is denoted LCG(69069,1,2^{32}) or

$$x_{i+1} = (69069 x_i + 1) \bmod (2^{32})$$

Implementation

The following C code is an implementation of RAND = LCG(69069,1,2^{32}). By modifying the values of a, b and m, the same code can be used for GGL.

```
double lcg(long int *xi)    /* m=2**32 */
{
  static long int a=69069, b=1, m=4294967296;
  static double rm=4294967296.0;

  *xi = (*xi*a+b)%m;
  return (double)*xi/rm;
}
```

The following main program generates 20000 random number pairs.

```
#include <stdio.h>
#include <stdlib.h>

long int getseed();
double lcg(long int *xi);

main(int argc, char **argv)
{
  long int seed;
  int i;
  double r1, r2;

  /* Seed calculated from time */
  seed = getseed();

  /* Generate random number pairs */
  for(i=1; i<=20000; i++) {
    r1 = lcg(&seed); r2 = lcg(&seed);
    fprintf(stdout,"%g  %g\n",r1,r2);
  }

  return(0);
}
```

The seed number is calculated from time using the following code:

```
#include <sys/time.h>

long int getseed()
{
  long int i;
  struct timeval tp;
  if (gettimeofday(&tp,(struct timezone *)NULL)==0) {
    i=tp.tv_sec+tp.tv_usec;
    i=i%1000000|1;
    return i;
  } else {
    return -1;
  }
}
```

Visual test

Linear congruential generators have a serious drawback of correlations between d consecutive numbers $\{x_{i+1}, x_{i+2}, \ldots, x_{i+d}\}$ in the sequence. In d-dimensional space, the points given by these d numbers order on parallel hyperplanes. The average distance between the planes is constant. The smaller the amount of these planes, the less uniform is the distribution. Good LCG's have a large amount of these planes, and thus they can be acceptable for many uses, but one should be aware that these correlations are always present.

As an example, consider the case $d = 2$. Figure 2 shows the spatial distribution of 20000 random number pairs in two dimensions on a unit square $x, y \in [0, 1]$. The points seem to be uniformly distributed throughout the unit square.

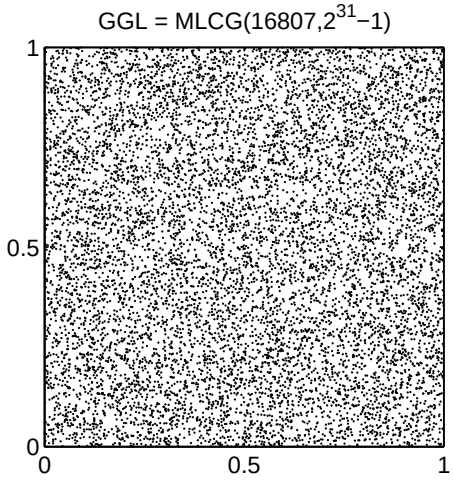


Figure 2: Spatial distribution of 20000 random number pairs on a unit square.

However, if we look at the distribution of random number pairs generated by GGL on a *thin slice* of the unit square $x \in [0, 0.001]$ and $y \in [0, 1]$, then we notice that the points do order in planes (Figure 3).

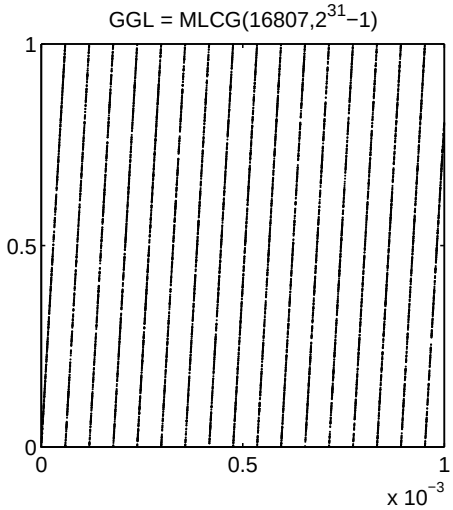


Figure 3: Spatial distribution of random number pairs on a thin slice on the unit square (GGL).

The same happens for RAND, although here the number of planes is larger (Figure 4).

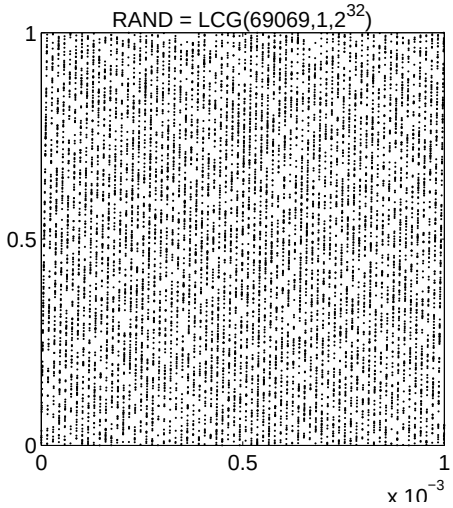


Figure 4: Spatial distribution of random number pairs on a thin slice on the unit square (RAND).

2.2 Lagged Fibonacci generators

Lagged Fibonacci (LF) generators are based on the generalization of the linear congruential method. The period of a LCG can be increased by the form

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + \ldots + a_p x_{i-p}) \bmod m$$

where $p > 1$ and $a_p \neq 0$.

A lagged Fibonacci generator requires an initial set of elements $x_1, x_2, \ldots, x_r$ and then uses the integer recursion

$$x_i = (x_{i-r} \otimes x_{i-s}) \bmod m$$

in which r and s are two integer lags satisfying $r > s$ and $\otimes$ is one of the following binary operations:

$+$	addition
$-$	subtraction
$\times$	multiplication
$\oplus$	exclusive-or

The corresponding generators are termed LF(r,s,$\otimes$). The initialization requires a set of q random numbers which can be generated for example by using another RNG.

The properties of LF-generators are not very well known but a definite plus is their long period. Some evidence suggests that the exclusive-or operation should not be used.

RAN3

RAN3 is a lagged Fibonacci generator LF(55,24,-) or

$$x_i = (x_{i-55} - x_{i-24}) \bmod m$$

The algorithm has also been called a subtractive method. The period length is $2^{55} - 1$. The initialization requires 55 numbers.

RAN3 does not suffer from similar correlations as the LCG's.

2.3 Shift register generators

Shift-register generators can be viewed as the special case $m = 2$ of LF generators. Feedback shift register algorithms are based on the theory of primitive trinomials

$$P(x; p, q) = x^p + x^q + 1 \qquad (x = 0, 1)$$

Given such a primitive trinomial and p initial binary digits, a sequence of bits $b = \{b_i\}$ $(i = 0, 1, 2, \ldots)$ can be generated using the following recursion formula:

$$b_i = b_{i-p} \oplus b_{i-p+q}$$

where $p > q$ and $\oplus$ denotes the exclusive-or (gives 1 iff exactly one of the b's is 1). The exclusive-or is equivalent to addition modulo 2.

Using the recursion formula, random words W_i of size l can be formed by

$$W_i = b_i b_{i+d} b_{i+2d} \cdots b_{i+(l-1)d}$$

where d is a chosen *delay*. The resulting binary vectors are treated as random numbers. It can be shown that if p is a *Mersenne prime* (which means that $2^p - 1$ is also a prime), then the sequence of random numbers has a maximal possible period of $2^p - 1$.

In *generalized feedback shift register* (GFSR) generators, l-bit words are formed by a recursion where two bit sequences are combined using the binary operation $\oplus$:

$$W_i = W_{i-p} \oplus W_{i-q}$$

The best choices for q and p are Mersenne primes which satisfy the condition

$$p^2 + q^2 + 1 = \text{prime}$$

Examples of pairs which satisfy this condition are:

$$\begin{aligned} p &= 98 & q &= 27 \\ p &= 250 & q &= 103 \\ p &= 1279 & q &= 216, 418 \\ p &= 9689 & q &= 84, 471, 1836, 2444, 4187 \end{aligned}$$

Generalized feedback shift register generators are denoted by GFSR(p',q',$\oplus$).

R250

R250 for which $p = 250$ and $q = 103$ has been the most commonly used generator of this class. The 31-bit integers are generated by

$$x_i = x_{i-250} \oplus x_{i-103}$$

250 uncorrelated seeds (random integers) are needed to initialize R250. The latest 250 random numbers must be stored in memory. The period length is $2^{250} - 1$.

Implementation

The following C code, is an implementation of R250 = GFSR(250,103,$\oplus$).

```
void r250(int *x,double *r,int n)
{
  static int q=103,p=250;
  static double rmaxin=2147483648.0;  /* 2**31 */
  int i,k;

  for (k=1;k<=n;k++) {
    x[k+p]=x[k+p-q]^x[k];
    r[k]=(double)x[k+p]/rmaxin;
  }
  for (i=1;i<=p;i++) x[i]=x[n+i];
}
```

The following procedure takes care of the initialization and returns a single random number. (R250 fills an array of size *n* with random numbers.)

```
#define NR 1000
#define NR250 1250
#define NRp1 1001
#define NR250p1 1251

/* Combination of two LCG generators to produce seeds */
double lcgy(long int *ix);

/* Calling procedure for R250 */
double ran_number(long int *seed)
{
  double ret;
  static int firsttime=1;
  static int i,j=NR;
  static int x[NR250p1];
  static double r[NRp1];

  if (j>=NR) {
    /* Initialize if first call */
    if(firsttime==1) {
      for (i=1;i<=250;i++) x[i]=2147483647.0*lcgy(seed);
      firsttime=0;
    }
    /* Call R250 */
    r250(x,r,NR);
    j=0;
  }
  j++;
  return r[j];
}
```

Finally, a sample program which generates random number pairs in $x \in [0, 0.001]$ and $y \in [0, 1]$.

```
long int getseed();
double ran_number(long int *seed);

main(int argc, char **argv)
{
  long int seed;
  int i;
  double r1, r2;

  /* Seed calculated from time */
  seed = getseed();

  /* Generate random number pairs */
  for(i=1; i<=20000000; i++) {
    r1 = ran_number(&seed); r2 = ran_number(&seed);
    if(r1 <= 0.001)
      fprintf(stdout,"%g  %g\n",r1,r2);
  }

  return(0);
}
```

Problems

The following figure shows that R250 does not exhibit similar pair correlations as the LCG generators,

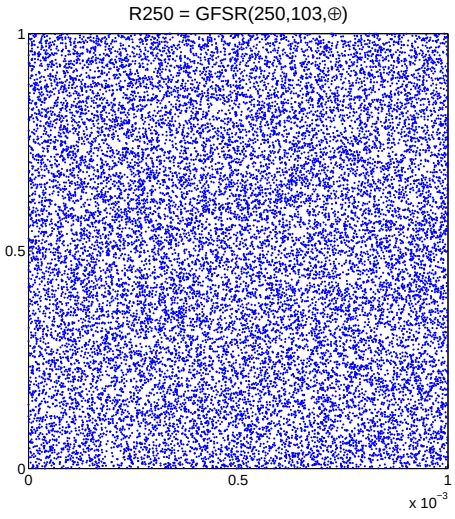


Figure 5: Distribution of random number points from R250.

But R250 does have strong *triple correlations*:

$$\langle x_i \, x_{i-250} \, x_{i-103} \rangle \neq 0$$

In addition, R250 *fails in important physical applications* such as random walks or simulations of the Ising model. An efficient way of reducing the correlations is to decimate the sequence by taking every kth number only ($k = 3, 5, 7, \ldots$).

2.4 Combination generators

Having seen that all pseudorandom number sequences exhibit some sort of dependencies, it seems natural that *shuffling* a sequence or *combining* two separate sequences might reduce the correlations. The combination sequence $\{z_i\}$ is defined by

$$z_i = x_i \otimes y_i$$

where x_i and y_i are from some (good) generators and $\otimes$ denotes a binary operation (+,-,$\times$,$\oplus$).

RANMAR

One of the best known and tested combination generator is RANMAR. (Presently the best RNG is Mersenne twister - see e.g. Wikipedia.) RANMAR is a combination of two different RNG's. The first one is a lagged Fibonacci generator

$$x_i = \begin{cases} x_{i-97} - x_{i-33} & \text{if } x_{i-97} \geq x_{i-33} \\ x_{i-97} - x_{i-33} + 1 & \text{otherwise} \end{cases}$$

Only 24 most significant bits are used for single precision reals. The second part of the generator is a simple arithmetic sequence for the prime modulus $2^{24} - 3 = 16777213$. The sequence is defined as

$$y_i = \begin{cases} y_i - c & \text{if } y_i \geq c \\ y_i - c + d & \text{otherwise} \end{cases}$$

where $c = 7654321/16777216$ and $d = 16777213/16777216$.

The final random number z_i is produced by combining x_i and y_i as follows:

$$z_i = \begin{cases} x_i - y_i & \text{if } x_i \geq y_i \\ x_i - y_i + 1 & \text{otherwise} \end{cases}$$

The total period of RANMAR is about 2^{144}.

Implementation

See course homepage `http://www.lce.hut.fi/teaching/S-114.1100/`.

Usage

When RANMAR is used, it must be first initialized by the call

```
crmarin(seed);
```

where `seed` is an integer seed.

The following call calls RANMAR which fills the vector `crn` of length `len` with uniformly distributed random numbers.

```
cranmar(crn,len);
```

RANMAR is a very *fast* generator. Using an array length of 100 numbers, generating 10^6 random numbers took 1.76 seconds on a Linux/Intel-Pentium4(1.8GHz) machine.

RANMAR has also performed well in several tests, and should thus be suitable for most applications (although testing is always recommended for each application separately).

3 Testing of random number generators

No single test can prove that a random number generator is suitable for all applications. It is always possible to construct a test where a given RNG fails (since the numbers are not truly random but generated by a deterministic algorithm).

3.1 Classification of test methods

1. **Theoretical tests** are based on theoretical properties of the algorithms. These tests are exact but they are often very difficult to perform and only asymptotic. Thus important correlations between consecutive sets of numbers in the sequence are not measured.

2. **Empirical tests** are based on testing the algorithms and their implementations in practice (i.e. by running programs and producing sequences of random numbers). These tests are versatile and can be tailored to measure particular correlations. They are also suitable for all algorithms. The downside is that it is often difficult to say how extensive testing is sufficient.

   Empirical tests can be further divided into *standard tests* (statistical tests) or *application specific tests* (measuring physical quantities). The tests can be implemented either for numbers or for bits.

3. **Visual tests** can be used to locate global or local deviations from randomness. As an example, pairs of random numbers can be used to plot points in a unit square. Visual tests can also be performed on bit level.

3.2 How to select a random number generator?

- *Good strategy in practice:* Check your application with at least two different random number generators and make sure that the results agree within the required accuracy.
- *Never use RNG's in commercial libraries!!* (These are usually of very low quality.)
- *Never use RNG's as black boxes* (without testing).

4 Different distributions of random numbers

4.1 Fundamentals of probability

Definition. Let X be a random variable $X \in [a,b]$. The cumulative probability distribution function $F(x)$ is defined to be the probability that $X < x$:

$$F(x) \equiv P(X \leq x) = \text{prob}(X \leq x)$$

For continuous variables, this can be expressed as

$$F(x) = \int_a^x p(x')dx'$$

where p is the probability density

$$p(x)dx \equiv \text{prob}(X \in [x, x+dx])$$

Normalization condition is

$$\int_a^b p(x)dx = 1$$

Two random variables are *statistically independent* or *uncorrelated* if

$$p(x_i, x_2) = p(x_1)p(x_2)$$

The mth moment of $p(x)$ is defined by

$$\langle x^m \rangle \equiv \int_a^b x^m p(x)dx$$

The first moment is the *mean* or *average*:

$$\bar{x} \equiv \langle x \rangle = \int_a^b xp(x)dx$$

The *variance* σ^2 is obtained from

$$\sigma^2 \equiv \langle x - \bar{x} \rangle^2 = \langle x^2 \rangle - \langle x \rangle^2$$

The *standard deviation* σ is given by the square root of the variance:

$$\sigma = \sqrt{\langle x^2 \rangle - \langle x \rangle^2}$$

4.2 Examples of important distributions

Uniform distribution

$$p(x) = \frac{1}{b-a} \qquad (a \leq x \leq b)$$

$$\text{Mean} \qquad \bar{x} = \frac{a+b}{2}$$

$$\text{Variance} \qquad \sigma^2 = \frac{(b-a)^2}{12}$$

In the special case where $x \in [0,1]$

$$\langle x^m \rangle = \int_0^1 x^m dx = \frac{1}{m+1}$$

Calculating the mth moments for $m = 1,2,3,\ldots$ is the simplest way to check whether a given distribution is uniform or not.

Gaussian distribution (Normal distribution)

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\bar{x})^2/2\sigma^2}$$

where $\bar{x}$ is the mean and σ^2 is the variance. These two parameters fully define the Gaussian distribution.

Exponential distribution

$$p(x) = \frac{1}{\theta} e^{-x/\theta} \qquad (x > 0)$$

The mean of the distribution is $\bar{x} = \theta$ (also called the decay constant) and the variance is $\sigma^2 = \theta^2$.

4.3 Transformation of probability densities

In some situations, we need random numbers which are not uniformly distributed but for example follow the Gaussian distribution.

The most general way to perform the conversion is as follows:

1. Sample y from a uniform distribution $y \in [0,1]$.
2. Compute $x = F^{-1}(y)$ where $F(x) = \int_a^x p(x')dx'$ is the cumulative distribution function. The variable x follows the distribution $p(x)$ as desired.

Example. The Lorentzian distribution.

$$p(x) = \frac{1}{\pi}\frac{1}{1+x^2} \qquad (-\infty < x < \infty)$$

1. Sample $y \in [0,1]$.
2. Compute x from

$$y = F(x) = \int_{-\infty}^{x} \frac{1}{\pi}\frac{1}{1+x'^2}dx' = \frac{1}{2} + \frac{1}{\pi}\arctan(x)$$

$$\Rightarrow x = F^{-1}(y) = \tan\left[\pi(y-\frac{1}{2})\right]$$

4.4 Gaussian random numbers

A fast and efficient way to generate Gaussian distributed random numbers from uniformly distributed $x_i \in [0,1]$ is the **Box-Muller algorithm**.

1. Draw two different random numbers x_1 and x_2 from a uniform distribution ($x_1, x_2 \in [0,1]$).
2. Construct

$$y_1 = \sqrt{-2\ln x_1}\,\cos(2\pi x_2)$$
$$y_2 = \sqrt{-2\ln x_1}\,\sin(2\pi x_2)$$

3. Multiply y_1 and y_2 by σ_1 and σ_2 to get the desired variances σ_1^2 and σ_2^2.
4. Goto 1.

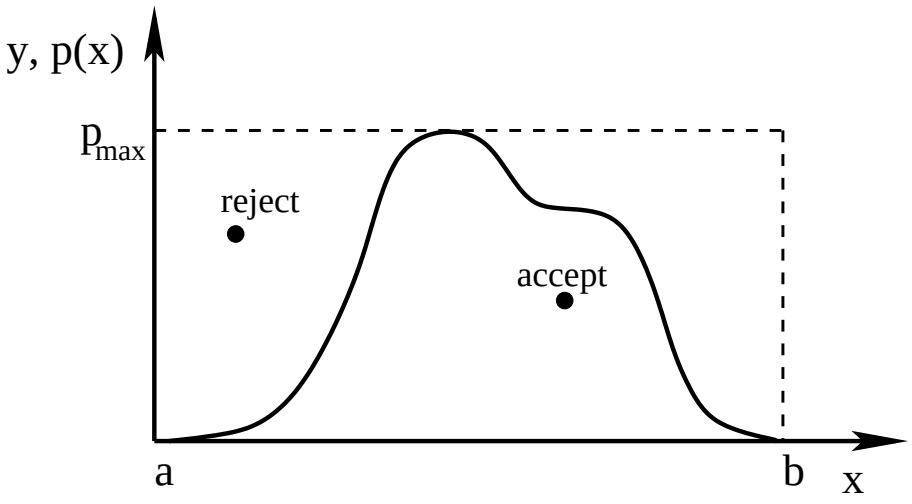
Obviously the quality of the random numbers generated by this algorithm depends on the quality of x_1 and x_2. It must also be checked that $x_i > \varepsilon$ where $\varepsilon(\approx 10^{-6})$ is the smallest number that causes an *overflow* in $-\ln\varepsilon$.

4.5 Rejection method

To generate random numbers that follow an arbitrary distribution $p(x)$ which is not analytically invertible (or known), the rejection method should be used.

The algorithm for the rejection method is given by:

1. Generate x and y in the box $x \in [a,b]$ and $y \in [0, p_{max}]$, where p_{max} is the maximum value of the probability distribution $p(x)$.
2. If $y \leq p(x)$, accept x.
3. Goto 1.


y, p(x)
p_{max}
reject
accept
a
b
x

5 Using random numbers

5.1 Example of a simple test

A simple test that can be used to check that your RNG implementation works as it should is to perform a so-called *moment test*. As seen before, the moments of the uniform distribution are known analytically:

$$\langle x^k \rangle = \frac{1}{k+1}$$

If we generate random numbers which should be uniformly distributed, then the moments calculated from these numbers should be approximately equal to the analytical values within statistical fluctuations.

Figure 6 shows how the mean value of the random numbers (the first moment $\langle x \rangle$) for 100 independent measurements using RANMAR. The 'measured' values fluctuate around the correct value 0.5. The data are for averages over $N = 1000$ and $N = 100000$ random numbers. Naturally, the fluctuations decrease as the number of averaged numbers grows.

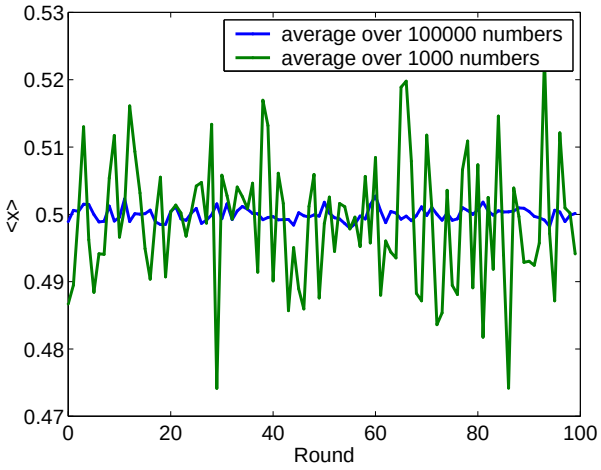


Figure 6: Measured mean value of a random number set ($N = 1000$ or $N = 100000$) for a sequence of 100 independent measurements.

The following data shows examples of the calculated values of the first five moments for $N = 10^4, 10^5$ and 10^6. Note that for each N, the values correspond to a single 'measurement' of the nth moment ($n = 1-5$). Figure 6 shows a sequence of 100 'measurements' of $\langle x \rangle$.

```
         nth moment       exact value       deviation

 N= 10000
 n=1    0.502453222656   0.500000000000   0.002453222656
 n=2    0.335839453125   0.333333333333   0.002506119792
 n=3    0.252290014648   0.250000000000   0.002290014648
 n=4    0.202056579590   0.200000000000   0.002056579590
 n=5    0.168516821289   0.166666666667   0.001850154622
```

```
        nth moment       exact value       deviation

N = 100000
n=1    0.500623476562   0.500000000000   0.000623476563
n=2    0.333602617188   0.333333333333   0.000269283854
n=3    0.250080078125   0.250000000000   0.000080078125
n=4    0.200001738281   0.200000000000   0.000001738281
n=5    0.166641953125   0.166666666667   0.000024713542

N = 1000000
n=1    0.500471968750   0.500000000000   0.000471968750
n=2    0.333628937500   0.333333333333   0.000295604167
n=3    0.250205343750   0.250000000000   0.000205343750
n=4    0.200141953125   0.200000000000   0.000141953125
n=5    0.166772375000   0.166666666667   0.000105708333
```

Central limit theorem

For any independently measured values $M_1, M_2, \ldots, M_m$ which come from the same (sufficiently short-ranged) distribution $p(x)$, the average

$$\langle M \rangle = \frac{1}{m} \sum_{i=1}^{m} M_i$$

will asymptotically follow a *Gaussian distribution* (normal distribution) whose mean is $\langle M \rangle$ (equal to the mean of the parent distribution $p(x)$) and variance is $1/\sqrt{N}$ times that of $p(x)$.

We can use this result to analyze the errors in the calculated values of any of the moments. Take for example the second moment and denote $M = \langle x^2 \rangle$. The central limit theorem states that the errors should follow the normal distribution and the width of this distribution should behave as $1/\sqrt{N}$. Thus let us take a set of m independent 'measurements' of the second moment, each consisting of an average obtained from N random numbers. From each measurement we obtain a single value M_α.

The average of all m measurements is

$$\langle M \rangle = \frac{1}{m} \sum_{\alpha=1}^{m} M_\alpha$$

and the standard deviation (variance) is given by

$$\sigma^2 = \langle M^2 \rangle - \langle M \rangle^2$$

Figure 7 shows the variance of $M = \langle x^2 \rangle$ obtained from $m = 1000$ measurements, each consisting of an average obtained using N random numbers.

We see that the variance behaves like $1/N^{1/2}$, which means that the second moment obeys the scaling of the central limit theorem. This tells us that, at least in what comes to the second moment, the random numbers generated by RANMAR seem to follow the uniform distribution.

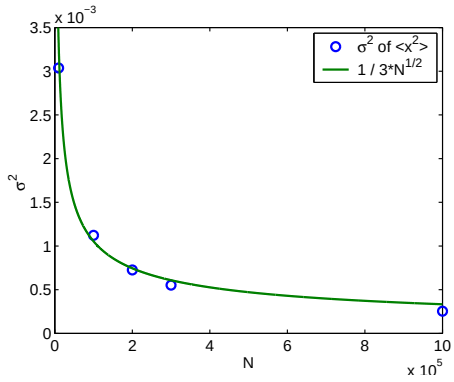


Figure 7: Variance of $M = \langle x^2 \rangle$ obtained from $m = 1000$ measurements, each consisting of an average obtained using N random numbers.

5.2 Uniformly distributed random points inside a circle

One must be careful in using random numbers because there are several incorrect ways of using random number generators. As an illustration of an incorrect way of using random numbers, we consider the problem of generating uniformly distributed random points inside a circle.

The equation of a circle with radius R is $x^2 + y^2 = R^2$. For simplicity, we set $R = 1$. One way to generate points inside the circle is to generate points inside the square $x, y \in [-1, 1]$ and discard those which are not inside the circle.

The following segment of C code does this:

```
/* Initialise ranmar */
crmarin(seed);

/* Generate random number vectors */
for(call=0;call<rounds; call++) {

    /* Load random number vectors */
    cranmar(crn,len);

    /* Generate points inside a square */
    for(i=0; i<len; i+=2) {

      /* Two uniform random numbers in [-1,1] */
      r1= 2.0*(crn[i]-0.5);
      r2= 2.0*(crn[i+1]-0.5);

        /* Select points inside the circle */
        if(r1*r1 + r2*r2 <= 1.0) {
          fprintf(outfile,"%lf  %lf\n",r1,r2);
          num++;
        }
      /* Stop when 2000 points have been found */
      if(num==2000) break;
    }
}
```

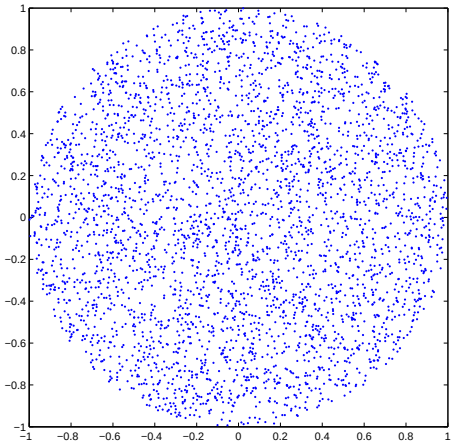


Figure 8: Distribution of 2000 points inside a circle.

Figure 8 shows the distribution of 2000 points inside the circle.

If we do not want to waste the random numbers by discarding points, we could think about the following way which forces the points to lie inside the circle. Using the following conversion from polar to Cartesian coordinates

$$x = r_1 \cos(2\pi r_2)$$
$$y = r_1 \sin(2\pi r_2)$$

where r_1 and r_2 are uniform random numbers in $[0,1]$. All the points (x,y) lie inside the circle $x^2 + y^2 = 1$, but the distribution is *not* uniform!

Figure 9 shows the distribution of points obtained using this algorithm. The points are strongly clustered near the center of the circle.

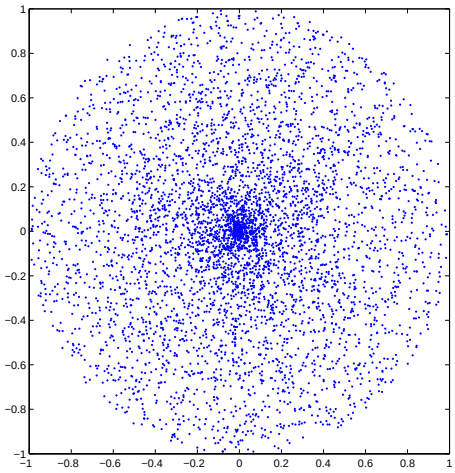


Figure 9: Nonuniform distribution of points inside a circle.

A correct way to generate a uniform distribution of points inside a circle is to use the given recipe for doing a *transformation between probability densities*.

The problem is defined in terms of the polar coordinates r and θ for which $0 \leq r \leq R$ and $0 \leq \theta \leq 2\pi$. We saw that a direct conversion from polar to cartesian coordinates gives a nonuniform distribution that is clustered near the center of the circle. On the other hand, the distribution is uniform with respect to the variable θ. Thus, we need a method that gives us a uniform distribution with respect to the radius r. This can be done as follows.

The probability density of the desired distribution can be written as a function of the polar coordinate r:

$$p(r) = C\,2\pi r$$

The constant C is determined from the normalization condition:

$$\int_0^R C\,2\pi r dr = 1 \quad \Rightarrow \quad C = \frac{1}{\pi R^2} \quad \Rightarrow \quad p(r) = \frac{2r}{R^2}$$

The cumulative distribution is thus

$$F(r) = \int_0^r p(r')dr' = \frac{r^2}{R^2}$$

We can now use the inverse of F to obtain the radius r of uniformly distributed points inside the circle:

$$s_1 = \frac{r^2}{R^2}$$

This gives

$$r = F^{-1}(s_1) = R\sqrt{s_1}$$

where s_1 is uniformly distributed in [0,1].

The x and y coordinates of the points are now obtained from the transformation between polar and cartesian coordinates:

$$x = r\cos(2\pi s_2)$$
$$y = r\sin(2\pi s_2)$$

where r is given by the formula above and s_2 is another uniformly distributed random number in [0,1].

Generalization of this aproach to a 3D case is straightforward. For example, generating uniformly distributed random numbers inside a sphere can be done by a similar conversion between the spherical and cartesian coordinates. The cumulative distribution function F is given by

$$F(r,\theta,\phi) = \frac{3}{4\pi R^3}\int_0^r \int_0^\theta \int_0^\phi r'^2 \sin\theta' dr' d\theta' d\phi'$$

where $0 \leq r \leq R$, $0 \leq \theta \leq \pi$ and $0 \leq \phi \leq 2\pi$.

The cumulative distribution function G of three uniform random variables s_1, s_2 and s_3 is given by

$$G(s_1,s_2,s_3) = \int_0^{s_1}\int_0^{s_2}\int_0^{s_3} ds_1' ds_2' ds_3'$$

You can do the transformation by equating F and G part by part. This gives

$$\int_0^{s_1} ds_1{}' = s_1 = \frac{3}{R}\int_0^{r} r'^2 dr'$$
$$\int_0^{s_2} ds_2{}' = s_2 = \frac{1}{2}\int_0^{\theta} \sin\theta' d\theta'$$
$$\int_0^{s_3} ds_3{}' = s_3 = \frac{1}{2\pi}\int_0^{\phi} d\phi'$$

Finally, the transformation between spherical and cartesian coordinates is given by

$$x = r\sin\theta\cos\phi$$
$$y = r\sin\theta\sin\phi$$
$$z = r\cos\theta$$

The following segment of C code does this in the 2D case (`2*len*rounds = 2000`):

```
/* Initialise ranmar */
crmarin(seed);

/* Generate random number vectors */
for(call=0;call<rounds; call++) {

    /* Load random number vectors */
    cranmar(crn,len);

    /* Generate points inside a circle */
    for(i=0; i<len; i+=2) {
      /* Two uniform random numbers */
      s1 = crn[i];
      s2 = crn[i+1];
      r = sqrt(s1);
      /* Two random numbers inside the unit circle */
      x= r*cos(2*PI*s2);
      y= r*sin(2*PI*s2);
      fprintf(outfile,"%lf  %lf\n",x,y);
    }
}
```

The resulting points are distributed uniformly inside the unit circle as is shown in Fig. 10.

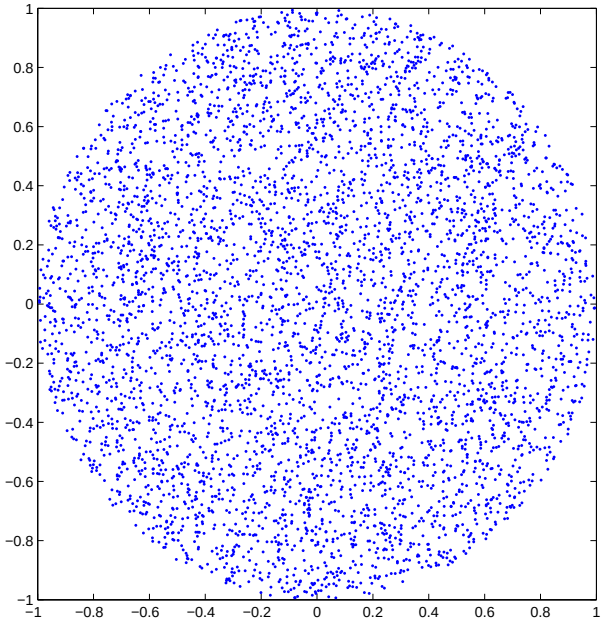


Figure 10: Uniform distribution of random points inside a circle.