POLYTECHNIC UNIVERSITY
Department of Computer and Information Science

PSEUDORANDOM NUMBER GENERATORS

K. Ming Leung

Abstract: The generation of pseudo-random numbers on modern computers is discussed. We will explore some of the properties of these numbers. A brief introduction how to use these numbers in computer simulation is presented.

Directory

Last Revision Date: January 25, 2004

Table of Contents

1. Introduction

Random numbers are used in many computer applications:

1. Computer simulation - internet traffic, customers arriving at a bank to be serviced
2. computer programming - as source of data for testing the effectiveness of computer algorithms
3. sampling - to provide insight into some questions without examining all possible cases
4. decision making - completely unbiased decision making is useful in some computer algorithms
5. numerical analysis - some numerical problems can be solved using random numbers
6. recreational games

After all most things in nature tend to be somewhat stochastic. The configuration of phone calls handled by a given phone company at any given time is certainly rather random. One certainly cannot

expect the same configuration to occur at any other time. Even cars made at a given manufacturing plant are to a certain degree different.

2. A wish-list for random numbers

For many applications, we need to use floating-point numbers that lie within the range 0 to 1, with any one number in the range just as likely as any other. These numbers are uniformly distributed from 0 to 1. They are referred to as uniform deviates or variates. We will denote them by u.

A computer is a deterministic machine and so can never produce numbers that are truly random. [Actually being truly random is not a very well-define concept.] A computer can only represent a finite number of numbers and so as one keeps on generating more and more "random" numbers, eventually the numbers have to repeat themselves. Thus the generators of these "random" numbers are often referred to as pseudorandom number generators.

Ideally we want the numbers generated to have the following characteristics:

1. numbers should be distributed uniformly between 0 and 1 without large "gaps"
2. sequence of numbers generated should be as independent from

each other as possible

3. mean or average of the numbers generated should be as close to $\frac{1}{2}$ as possible
4. the variance should be as close to $\frac{1}{12}$ as possible
5. should have little cyclic variations, i.e. free from the following problems:
   (a) autocorrelation between numbers
   (b) numbers successively higher of lower than adjacent numbers
   (c) several numbers above the mean followed by several numbers below the mean
6. numbers should have a long cycle
7. numbers should be replicable, i.e. the same starting condition should yield the same sequence of numbers (for debugging and comparison reasons)
8. routine that generates these numbers should be extremely fast but should require very little memory
9. the generator should be portable

3. Linear congruential generators

Most pseudorandom numbers generators use an iterative scheme for integers, I,:

$$I_{n+1} = \Phi(I_n), \quad n = 0, 1, \ldots \tag{1}$$

starting with a given seed I_0. Most functions for Φ are undesirable for generating pseudorandom numbers. A very common choice is a linear function where modular arithmetics is to be performed. These generators are called linear congruential generators. The general scheme has the form

$$I_{n+1} = (aI_n + c) \mod m, \quad n = 0, 1, \ldots, \tag{2}$$

where a (the multiplier), c (the increment or shift) and m (the modulus) are 3 integer constant parameters. The uniform deviates, u, are produced mathematically by dividing by m using floating-point operation.

$$u_n = I_n/m. \tag{3}$$

3.1. Remarks on linear congruential generators

Linear congruential generators have the following general properties:

1. The scheme is very simple. It is reasonably fast and requires little memory.
2. The choice of values for a, c, m and I_0 sensitively affects the statistical properties of the numbers generated and the maximum cycle length.
3. Uniform deviates generated do not continuously fill up the line segment from 0 to 1. They are discrete and can only assume values from the set $0, \frac{1}{m}, \frac{2}{m}, \ldots, \frac{m-1}{m}$. The smallest possible gap size is $\frac{1}{m}$.
4. One can generate at most m distinct numbers before repeating the same sequence of numbers, i.e. the maximum possible cycle length is m. So m is often chosen to be the largest integer that can be represented. For example on a 32-bit computer, one bit is used for the sign bit and so the largest integer that can be represented is $2^{31} - 1 \approx 2$ billion. We can easily run out of numbers on today's computers.

5. The modulus operation with m can be conducted efficiently if m is an integer power of 2 by saving only the rightmost digits. For example if $m = 2^k$, then we save only the k rightmost digits.

6. Not too difficult to have situations when successive numbers are used in such a way that conflicts with the generating algorithm. For example, Statistically, it happens one time in a million that a number generated is less than 10^{-6}. If $a = 7^5 = 16807$ and $m = 2^{31} - 1$, then this number will always be followed by a number less than $10^{-6} * 16807 \approx 0.0168$.

A good choice of parameters due to Parker and Miller has $c = 0$, $a = 7^5 = 16807$ and $m = 2^{31} - 1$. There are other two choices: $a = 48271$ and 69621, with the same m. No other choices should be used!

3.2. Explore some toy models

To gain some insights into these linear congruential generators, let us consider the one having parameters: $c = 0$, $a = 13$, and $m = 2^6 = 64$.

Using $I_0 = 1$ as seed, the following sequence of 16 pseudorandom integers is generated: $1, 13, 41, 21, 17, 29, 57, 37, 33, 45, 9, 53, 49, 61, 25, 5$. After that the sequence repeats itself. The cycle length is therefore 16. Starting with 1, every $4th$ integer appears in the sequence. The gap size is $4/64 = 1/16 = 0.0625$. [Remember that these random integers have to be divided by m to produce the random variates.

If $I_0 = 2$ is used as seed, the following sequence of 8 random integers is obtained: $2, 26, 18, 42, 34, 58, 50, 10$. Starting with the seed, every $8th$ integer appears in the sequence. The cycle length is 8 and the gap size is $8/64 = 1/8 = 0.125$.

The following sequence is obtained for a seed of 3: $3, 39, 59, 63, 51, 23, 43$ The cycle length is 16 and the gap size is 4.

Finally with a seed of 4, we have the sequence: $4, 52, 36, 20$. Starting with the seed, every $16th$ integer appears in the sequence. The cycle length is only 4 and the gap size is $4/16 = 1/4 = 0.25$.

3.3. Marsaglia phenomenon

In 1968, Marsaglia points out in an article a major weakness of linear congruential generators: successive overlapping sequences of n numbers all fall on at most $(n!m)^{1/n}$ parallel hyperplanes. The title of the article is "Random Numbers Fall Mainly in the Planes". (Recall a song in the play or movie: "My Fair Lady", with one of the best-known lines in the show: "The rain in Spain stays mainly in the plain.")

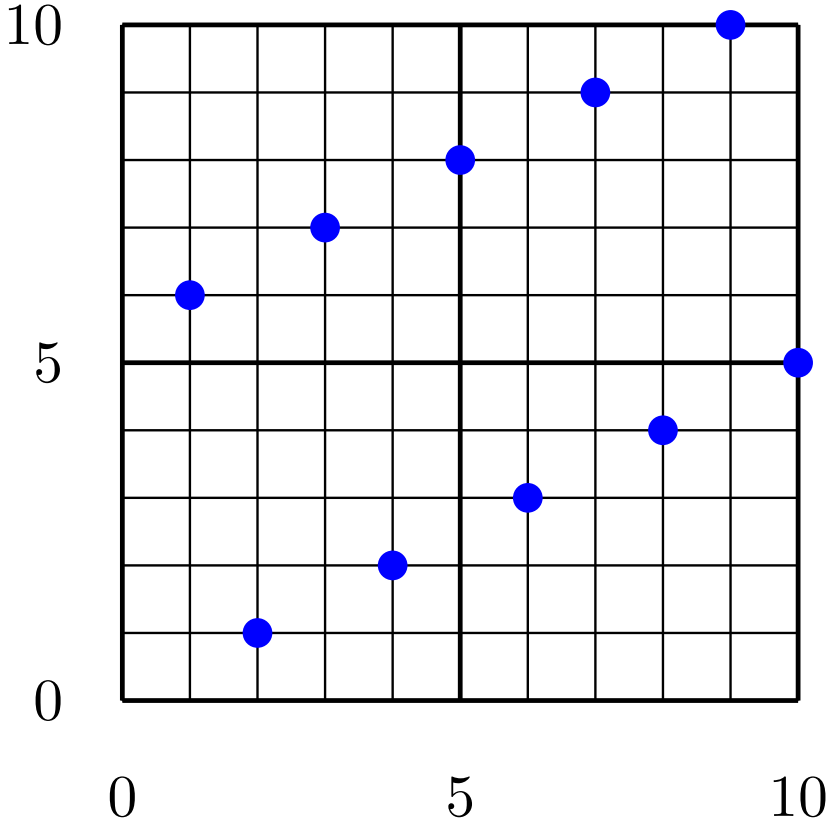
Assuming that $m = 2^{31}$, the maximum number of hyperplanes for a given n is shown in the following table.

n	$(n!m)^{1/n}$
1	2^{31}
2	2^{16}
3	2344
4	476
5	192
6	108
7	73
8	55
9	45
10	39

To illustrate the Marsaglia phenomenon, we use the linear congruential method with parameters $a = 6$, $c = 0$, and $m = 11$. We obtain the following sequence of integers starting with a seed of 1: 1, 6, 3, 7, 9, 10, 5, 8, 4, and 2. After integer 2, the sequence of integers repeats. We label this sequence of 10 $(=N)$ integers by $I_0, I_1, \ldots, I_9$. They appear to be rather random.

However if we plot points I_{n-1} versus I_n for $n = 1, 2, ..., 10$, where

I_{10} is taken to be the same integer as I_0, we generate the follow two dimensional plot.



It is clear that all these numbers lie within only two "planes" in this two dimensional space.

Professor Alain Bellerive has written a nice *Java applet* Java applet that demonstrates the Marsaglia phenomenon.

4. Mersenne Twister: A Much Better Generator

So far the best pseudorandom number generator seems to be the *Mersenne Twister* developed by Makoto Matsumoto and Takuji Nishimura during 1996-1997. Mersenne Twister has the following merits:

1. It is designed with consideration on the flaws of various existing generators.
2. Far longer period and far higher order of equidistribution than any other implemented generators. (It is proved that the period is $2^{19937-1}$, and 623-dimensional equidistribution property is assured.)
3. Fast generation. (Although it depends on the system, it is reported that Mersenne Twister is sometimes faster than the standard ANSI-C library in a system with pipeline and cache memory.)
4. Efficient use of the memory. (The implemented C-code mt19937.c consumes only 624 words of working area.)
5. The algorithm is coded into many modern languages and can

be freely downloaded.

5. Monte Carlo Simulation

We will be studying at some depth about Monte Carlo simulation methods. We begin here with some introductory remarks and follow by a concrete simple illustration how such methods can be used in computer simulations.

5.1. Introduction to Monte Carlo Simulations

Stochastic simulation is often a very power method for solving certain types of mathematical problems. Stochastic simulation methods attempt to mimic or replicate the behavior of a system by exploiting randomness to obtain a statistical sample of possible outcomes. Because of the randomness involved, these methods are also commonly known as Monte Carlo methods. They are useful for studying:

1. nondeterministic (stochastic) processes such as nuclear reactions (Monte Carlo methods were actually invented for the development of atomic weapons)

2. deterministic systems that are too complicated to model analytically
3. deterministic problems whose high dimensionality makes standard discretizations infeasible (for example Monte Carlo integration).

The two main requirements for using stochastic simulation methods are:

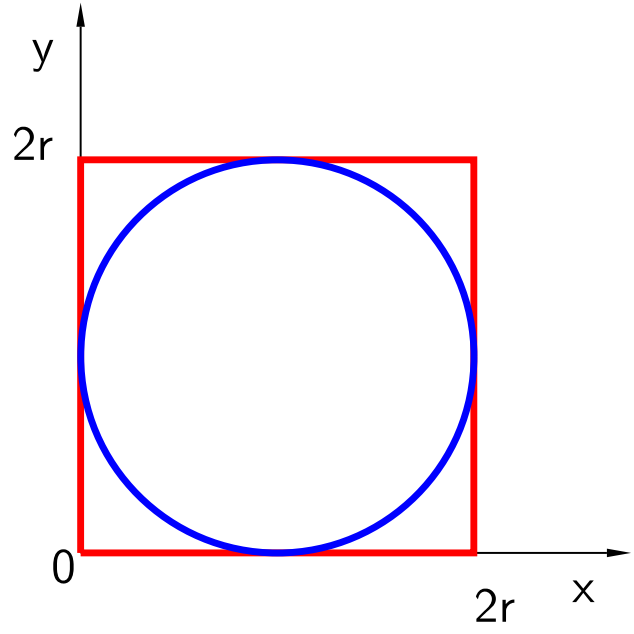
1. knowledge of relevant probability distributions
2. a good supply of random numbers for making random choices

Knowledge of the relevant probability distributions depends on theoretical or empirical information about the physical system being simulated. As a simple example, in simulating a baseball game the known batting average of a player might determine the probability that the player gets a hit in a given turn at bat.

5.2. A Simple Example of Monte Carlo Simulation

Imagine we have a circular rug of radius r that fits snugly inside a room of dimension $2r$ by $2r$ in the attic of a house. The room is

rather dusty and the floor as well as the rug is covered by a thin but uniform layer of dust particles. If we know that the total number of dust particles in the room is N, and out of those N particles we have N' particles covering the rug, thus we have a way of computing the value of π.



Assuming that each dust particle lands anywhere in the room with equal probability, the ratio of N' to N must be the same as the ratio

of the area of the rug to the area of the entire room. Thus we have

$$\frac{N'}{N} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}, \tag{4}$$

and so

$$\pi = 4\frac{N'}{N} \tag{5}$$

This suggests a way of computing the value of π by performing the following simulation. We take a speck of dust particle and throw it randomly inside the room, making sure that it has an unbias chance of landing anywhere inside. We have two counters, one for the number of particle we through, and the other for the number of particle that lands on the rug. We continue this process and stop after using up all N particles. The above formula then gives us an estimation of the value of π.

Thus we need to perform the following computation. Use the pseudorandom number generator to obtain a uniform variate, u. We scale it up by multiplying it by 2 so that the value lie randomly between 0 and 2 instead of between 0 and 1. We assign that value to be the x-

coordinate of a dust particle. We obtain another random variate from the generator, and again multiply it by 2. This time we assign the resulting value to be the y-coordinate of the particle. That particle lands on the rug if $(x-1)^2+(y-1)^2<1$. In that case we increment the counter N' by 1, otherwise we repeat the process with another particle. [Note that you can also compare $\sqrt{(x-1)^2+(y-1)^2}$ with 1, but the square root is really not necessary and it slows down the program by roughly a factor of 2.]

The algorithm for computing π using MC simulation is:

1. Initialize counters N and N' to 0.
2. Go through the following loop N times:
   (a) Generate a random variate u.
   (b) Let $x = 2u$.
   (c) Generate another random variate u.
   (d) Let $y = 2u$.
   (e) Increment N' by 1 only if $(x-1)^2 + (y-1)^2 < 1$.
   (f) Increment N by 1 and repeat.
3. Value of π is given by $4N'/N$.

5.3. Errors in Stochastic Simulations

In addition to obtaining the quantity of interest, it is especially important in a computer simulation to obtain a good measure of the amount of error we may have for that quantity. We will be spending several weeks analyzing errors associated with Monte Carlo simulation methods and ways to reduce these errors.

First of all it is clear that any quantity obtained from a simulation almost always suffer from errors due not to roundoff errors (which are usually comparatively very small) but due to the stochastic nature of the simulation. One can see that in any one of the following ways.

1. Perform a simulation using a given number of particles N. Then repeat the simulation using the same number of particles but change the seed at the start of the simulation. The positions of these N particle will therefore be different compared with the first simulation. You will find that the quantity you obtain for N' and therefore for the value of π will be different.

2. Perform a simulation using a given number of particles N_1. Repeat the simulation using a higher number of particles, $N_2 > N_1$

but using the same seed as the first simulation. You will find that the results obtained for π are different for the two simulations.

3. Consider the following thought or hypothetical ("gedanken") experiment. Imagine performing a simulation with N number of particles, obtain N' number of particles inside the rug, and therefore $\pi \approx 4N'/N$. Imagine repeating the simulation using the same seed as the first experiment but with $N+1$ particles. The first N particle have exactly the same positions as the first experiment and so the same value for N'. If the last extra particle in the second experiment is inside the rug, then N' is one higher than for the first experiment and therefore the result for π is $4(N'+1)/(N+1)$. But this cannot be the same as the previous value of $4N'/N$, because

$$4\frac{N'+1}{N+1} = 4\frac{N'}{N} \Rightarrow N(N'+1) = N'(N+1) \Rightarrow N = N', \quad (6)$$

which cannot be true. On the other hand, if the last extra particle is outside the rug, then N' is the same as in the first

experiment, and therefore the value of π must be slightly less than the value we obtained from the first experiment.

We perform the above simulation using 5000 particles and a certain starting seed for the pseudorandom number generator we are using, and obtain a value of 3.1096 for π. The relative error is -0.0102. A second calculation using the same number of particles but with a different starting seed gives 3.1536 and a relative error of 0.0038. The randomness in the computed values is evident.

Next we repeat the simulation using $500,000$ particles but with 3 different seeds each time. The results for π are $3.1408(-0.00025)$, $3.1396(-0.00063)$, and $3.14284(0.00040)$. The numbers in parentheses are the corresponding relative errors. This result strongly suggests that the error generally decreases with increasing number of particles. We will re-examine the errors to be expected in this type of Monte Carlo simulation in more detail in about two weeks.

5.4. Main Advantage of Computer Simulation

It is clear from the above example of using Monte Carlo simulation to compute the area of a circle that by a slight modification of the program we can easily compute that area (or volume in three dimension) of any object, no matter how irregular its shape may be. All we need is to be able to find out if a given point is inside the object or not. Thus we see that Monte Carlo simulations (and in fact computer simulations in general) are very versatile and can readily be adapter to treat extremely complex problems. This is where computer simulations become so invaluable.

References

[1] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, 3rd edition (McGraw Hill, 2000).

[2] Ilya M. Sobol, *A Primer for the Monte Carlo Method*, (CRC Press, 1994).

[3] J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*, (Methuen, London, 1964).

[4] M. H. Kalos and P. A. Whitlock, *Monte Carlo Methods*, (Wiley, New York, 1986).

[5] P. Bratley, B. L. Fox and E. L. Schrage, *A Guide to Simulation*, (Springer-Verlag, New York, 1983).

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery *Numerical Recipes*, Second Edition, Sec. 7.6, p. 295 (Cambridge University Press, 1992).