12
Generation of uniform $\hat{\mathbf{U}}(0,1)$ random numbers

12.1
Pseudorandom numbers

In this appendix we explain how it is possible to generate $\hat{\mathbf{U}}(0,1)$ independent random numbers, i.e. random numbers uniformly distributed in the $(0,1)$ interval that can be efficiently used in any stochastic algorithm, Monte-Carlo or Langevin. We can not, and will not, cover all the vast bibliography in this topic. Our aim is to introduce the reader to the major problems one encounters for the generation of uniform random numbers and the basic families of algorithms that have been proposed, so that the routines that will be used do not appear to us as "magic black boxes" which provide random numbers in an unknown and uncontrollable way.

The first thing we need to realize is that the algorithms for the generation of random numbers are implemented in a computer where real numbers are not stored with an infinite precision, but the number of decimal places is finite and set from the beginning. Therefore, most random number generators provide them in the form of a fraction $u = m/M$, where m is an integer number uniformly distributed in the interval $[0, M-1]$[1)]. The greater M, the better the approximation of the generated numbers to the true distribution $\hat{\mathbf{U}}(0,1)$.

Once a large value of M has been chosen, the question to answer is: how can we generate integer numbers uniformly distributed in the interval $[0, M-1]$? A legitimate way could be the following: make M identical balls numbered 0 to $M-1$ and place them in a bag. Shake the bag, pick one of the balls and read the number, say m_0, on the ball. Replace the ball in the bag, shake the bag again, pick one of the balls and read the number, say m_1, on the ball. Replace the ball, shake the bag again ... and repeat as many times as random numbers you need.

First of all, it is not clear that this procedure would produce a set of truly uniform random numbers. Some of the balls might be slightly heavier that the others and have a different probability of being chosen, once replaced. If the shaking is not good enough, the ball might stay near the top of the bag and have a larger chance

1) In some cases, and in order to avoid some singularities that can appear when computing functions such as the logarithm, it is better to write the uniform random number as $(m+0.5)/M$. However, this is not very important in the majority of applications.

to be chosen again, etc. We could certainly improve the procedure. For instance, chose $M = 2^b$, a power of 2 and work in base 2 (a base loved by computers). Then we could manage simply with two balls labeled 0 and 1. For the generation of each number m_i, we would extract b balls and construct that number bit by bit. In any event, even if the procedure were absolutely fair, it is clear that it is not efficient. Recall that we might need a large number (of the order of millions or billions) of random numbers for our numerical work and it would be completely unpractical to obtain them this way. It is best to design some numerical algorithm to generate the values $m_0, m_1, m_2, \ldots$. Now, a numerical algorithm is, by definition, deterministic. It produces the same result every time we run it. How can then be random the obtained sequence of numbers? The answer is subtle: the sequence will not *be* random, but it will *look* random. Now there is a thin line between "to look" and "to be" and we ask whether "to look" is enough for the applications. Since the numbers we will generate "are" not random, but "look" random, they are sometimes called, for the sake of rigor, "pseudorandom" numbers.

Let us first explain the first historical algorithm designed to generate pseudorandom numbers. It is due to von Neumann. It takes $M = 10000$ and sets an initial value $m_0 \in (0, 9999)$ at will. This first choice could be deterministic (for instance, my birth year) or truly stochastic (the last four digits of the time in seconds since the computer has been up). Once the first value, the so-called "seed", has been set, it is squared, m_0^2, the last two digits are discarded and the new last four digits constitute the next value m_1. The process is repeated to obtain $m_2, m_3, \ldots$. Let us see an example taking $m_0 = 5232$ as seed:

$$\begin{aligned} m_0 = 5232 &\to m_0^2 = 27373824 \to m_1 = 3738 \\ m_1 = 3738 &\to m_1^2 = 13972644 \to m_2 = 9726 \\ m_2 = 9726 &\to m_2^2 = 94595076 \to m_3 = 5950 \end{aligned}$$

The sequence follows the algorithm

$$m_{i+1} = [m_i^2/100] \bmod 10000, \tag{12.1}$$

being $[x]$ the integer part of x. In the previous example, this would yield the sequence of random numbers: $u_0 = 0.5232$, $u_1 = 0.3738$, $u_2 = 0.9726$, $u_3 = 0.5950$, etc. The question is whether this sequence "looks" random. i.e. If we give this sequence to a good statistician, will he be able to tell us that the sequence is not really random? Given a long list of numbers $(u_0, u_1, u_2, \ldots)$, what would the statistician do to determine whether it is random or not? He would apply a series of tests to check for randomness. Very sophisticated tests do exist and it is not our intention to describe them in any detail. We just state two basic tests:

1 Moments. Check that $\langle u_i^n \rangle = (n+1)^{-1}$. In particular, $\langle u_i \rangle = 1/2$, $\langle u_i^2 \rangle = 1/3$.
2 Correlations. Check that $\langle u_i u_j \rangle = 0.25$ if $i \neq j$.

But there are many other tests that should be satisfied by true random numbers. It is not difficult to find a test that the von Neumann generator does not satisfy. For a simple reason: von Neumann generator is necessarily cyclic. i.e. if there exists an

integer number L such that $m_L = m_0$, then the series repeats itself $m_{i+L} = m_i$. At most, the period L is equal to 10000 (the maximum possible number of values for m_i), but it can be much less than that. Obviously, no truly random sequence is cyclic. Another problem is that if the number 0 is hit at some time, $m_K = 0$, then $m_i = 0, \forall i \geq K$. It is not necessary to be a good statistician to realize that a series of zeros does not look very much random. If a modern simulation that uses millions or billions of random numbers, these problems are certainly to appear sooner than later and von Neumann algorithm, although very important conceptually, is useless from the practical point of view.

But even if von Neumann algorithm does not satisfy the minimal needs for modern simulations, it certainly sets the basic settings of what are called *congruential generators*. All we need to do is to replace M by a much larger number and devise an algorithm replacing Eq.(12.1). Before tackling this question, let us enumerate the properties we would like a good pseudorandom number generator to have.

1.- Good statistical properties This has been said before. The generator must pass a series of statistical tests. However, we must realize that there are always some tests that the generator will fail at. The question is if these tests are important to the particular application we are using the numbers for. For example, most generators have a period L after which they repeat themselves. This period L has to be much larger that the length of the sequence of random numbers required. A rule of thumb says that we can not use more than $L^{1/2}$ numbers in our simulation before the good statistical properties are lost.

2.- Efficiency. A modern simulation requires billions of numbers. Besides the generation of the random numbers, many other operations have to be made. The generation of random numbers can not be a bottle-neck for the simulation.

3.- Reproducibility. This property might look paradoxical at first sight. How can we demand that a random number sequence should be reproducible? Note, however, that we are dealing with complicated numerical algorithms that have to be fully controllable from beginning to end. Imagine that you rush to your thesis supervisor and tell him that your simulation has yielded a beautiful result (a critical exponent that verifies some hyperscaling theory) but that you can not reproduce that result!! You will be in trouble. Furthermore, when we are debugging a program we do not want to have any uncontrolled source of variability in the results of the program.

12.2 Congruential generators

They are based on a recurrence relation of the form

$$m_{i+1} = F(m_0, m_1, \ldots, m_i) \mod M, \tag{12.1}$$

and a suitable large number M. Due to the modulus[2)] operation, all numbers satisfy $m_i \in [0, M-1]$. One might think that complicated nonlinear relations would be necessary to produce a "chaotic" sequence that looks random. However, it comes to a small surprise that the simplest one-step memory linear relation

$$m_{i+1} = a m_i + c \mod M \tag{12.2}$$

suffices to yield random numbers with good statistical properties if the numbers a, c and M are chosen conveniently. Notice, though, that any one-step recurrence relation of the form $m_{i+1} = F(m_i) \mod M$ will have a period L at most equal to M. Furthermore, the probability that two consecutive numbers are identical is equal to zero (otherwise the sequence repeats itself infinitely). This would not be the case if the numbers m_i were really random since then the probability that $m_{i+1} = m_i$ is equal to $1/M$. Of course, if M is large, this is a small number and it might not be important that it is instead equal to zero for our generator.

In order to stress the deterministic character of (12.2), let us write down the explicit solution of this recurrence relation

$$m_i = \left(m_0 - \frac{c}{1-a}\right) a^i + \frac{c}{1-a} \mod (M). \tag{12.3}$$

A first criterion to determine whether a congruential generator defined by the set of numbers (a, c, M) has good properties is to make sure that its period L is very large. How large can it be? If $c \neq 0$ then L can be as large as M. However, if $c = 0$, we want to avoid the value $m_i = 0$ as it would lead to $m_j = 0$ if $j \geq i$. This implies that L could be at most equal to $M-1$. In a somehow old-fashioned terminology, a generator with $c \neq 0$ is called a "mixed" congruential generator, whereas one with $c = 0$ is called a "multiplicative" congruential generator. We only discuss the conditions under which a mixed congruential generator has the maximum period M and refer the reader to more specialized bibliography if he is really interested.

One can prove a theorem that says that a mixed congruential generator reaches the maximal period $L = M$ if and only if:

(i) c and M are relatively prime numbers (their greatest common divisor is 1).
(ii) $a = 1$ mod g, for each prime divisor g of M.
(iii) $a = 1$ mod 4, if M is multiple of 4.

It is very useful in a computer to use a number M of the form $M = 2^b$. This is so because computers work with binary numbers, and the congruence modulus a power of 2 is then very easy to obtain. Just think in human (base 10) terms: $12891243712340234 \mod 10^6$ is 340234, the last six digits. In the same way for a computer it is easy to determine that $1101010111111000111 \mod 2^6 = 000111$ or, in base 10 notation, $876487 \mod 64 = 7$. In this case of $M = 2^b$, the conditions for a maximum period are simply that c must be odd and that $a = 1 \mod 4$.

Once these general properties have been established, people have searched for (in many cases using trial and error techniques) triplets of numbers (a, c, M) that yield

2) A modulus is also called a congruence, hence the name

good statistical properties. There are no theorems now and mostly we believe what other people tell us about the quality of a generator. With this in mind, three sets of allegedly "good" numbers are:

$$\begin{array}{lll} M = 2^{35}, & a = 129, & c = 1, \\ M = 2^{32}, & a = 69069, & c = 1, \\ M = 2^{32}, & a = 1812433253, & c = 1. \end{array}$$

Although it seems that the third row of values is superior in the sense that the produced numbers have better statistical properties, the second row has been extensively used. The "deep" reason seems to be that $a = 69069$ is an easy number to remember! Most "free" random number generators (those coming built in with the machine operating system or main compilers) are congruential with $(a, c, M) = (69069, 1, 2^{32})$.

The use of $M = 2^{32}$ has another advantage since most integer arithmetics is performed with 32 bits accuracy. The maximum integer number that can be written with 32 bits is $2^{32} - 1$ and any integer multiplication whose result is larger that $2^{32} - 1$ results in an overflow. Indeed we do not care about the overflow, since we are performing arithmetics modulus 2^{32} and simply neglecting the overflowed bits gives the correct answer. However, compilers can get annoying if they find an overflow and it necessary to tell the computer not to worry about the overflow whenever it is produced (this is usually achieved with some option during compilation, for example `-check=nooverflow` or something similar). In this way, the modulus operation is performed automatically. The only thing to worry about is that in a 32 bit representation, a number which has the first bit, the most significant bit, equal to 1 is considered to be a negative number. In fact, integer numbers in 32 bit arithmetics run from -2^{31} to $2^{31} - 1$ while we consider them to run between 0 and $2^{32} - 1$. All we need to take into account, then, is that if the computer tells us that the random number $u_i = m_i/M$ is negative, indeed it should be $(m_i + M)/M = 1 + u_i$.

With all this in mind, let us give a simple computer program for the function `ran_u(m)` that generates pseudorandom numbers uniformly distributed in the $(0, 1)$ interval.

```
function ran_u()
integer, save :: m=1234567
parameter (rm=2.0**(-32),ia=1812433253,ic=1)
m=m*ia+ic
ran_u=rm*m
if (ran_u < 0.0) ran_u=1.0+ran_u
end function ran_u
```

It uses the particular value `m=1234567` as the seed. If we want a different sequence of numbers, we can change the value for the seed. For this program to work, we have to be sure that the integer arithmetics is performed with 32 bits precision and that the program does not detect overflows. We can avoid these conditions if we use double precision arithmetics, such as:

```
double precision function ran_u()
```

```
double precision ia,ic,ma,rm
double precision, save :: m=1234567.0d0
parameter (ma=2.0d0**32,rm=2.0d0**(-32))
parameter (ia=1812433253.0d0, ic=1.0d0)

 m=mod(m*ia+ic,ma)
 ran_u=rm*m

end function ran_u
```

but this is usually slower that the previous implementation. Do not forget to define in this case `ran_u` as double precision in the calling program.

12.3 A theorem by Marsaglia

We have already mentioned that numerical deterministic algorithms will not produce true random numbers and that there will always be some test that our generator will not pass. Here comes an interesting test that a simple congruential random generator does not pass. Let us take the recurrence relation (12.2) with $M = 2^8$, $a = 25$, $c = 1$. We generate random numbers $(u_0, u_1, u_2, \dots)$ and organize them in pairs (u_i, u_{i+1}). Each pair is then plotted in a 2-dimensional plane. The result is in figure 12.1, left panel. If the numbers u_i were truly independent of each other, the points would look uniformly distributed in the plane, while it is clear that there is an underlying pattern. The first reaction is that we should blame our choice of the numbers (a, c, M) which was not good enough. Indeed, if we repeat the plot using the numbers $(69069, 1, 2^{32})$ the result looks much better, see the right panel of figure 12.1. Is there *a priori* way of choosing of choosing the triplet (a, c, M) such that these correlations between consecutive numbers do not exist?

A surprising and negative answer was given by Marsaglia in a very interesting article[60] with the suggestive title *Random Numbers Fall Mainly in the Planes*. In short, Marsaglia has shown that **all** congruential generators will have subtle correlations. These correlations show up when we organize the sequence of random numbers $(u_0, u_1, u_2, \dots)$ in groups of d to generate points in a d-dimensional space $z_1 = (u_0, u_1, \dots, u_{d-1})$, $z_2 = (u_d, u_{d+1}, \dots, u_{2d-1})$, etc. The theorem proves that there exists a dimension d for which all points $z_1, z_2, \dots$ fall on a hyperplane of dimension $d-1$. This is rather annoying and shows that congruential generators fail to pass a particular test for independence. The accepted compromise is to use a generator which has a large dimension d for the Marsaglia planes. Only then can we maybe accept that the presence of these correlations between the numbers will not hopefully be of any significance for our calculation.

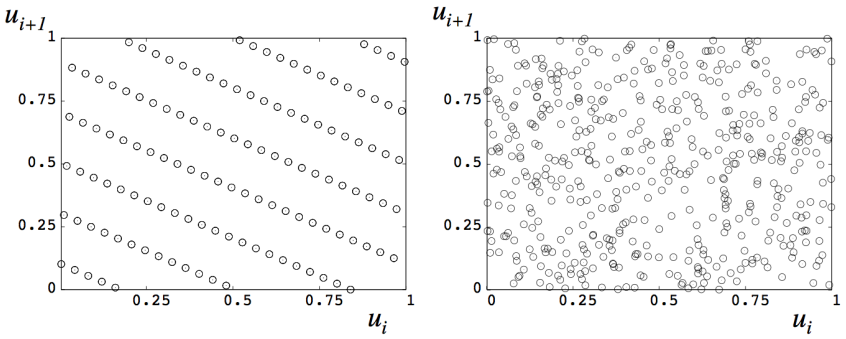

Figure 12.1 Plots of pairs of random numbers (u_i, u_{i+1}) obtained by using the congruential algorithm (12.2) with $(a, c, M) = (25, 1, 2^8)$ (left panel) and $(a, c, M) = (69069, 1, 2^{32})$, right panel. Note the clear presence of the Marsaglia planes in this $d = 2$ plot in the left panel

12.4 Feedback Shift Register generators

The "Feedback Shift Register" (FSR) random number generators use similar ideas to the congruential generators but organize the resulting numbers in a different way. The idea is to use a congruential generator modulus 2 (so it only produces 0 and 1) but to increase the memory of the recurrence relation. If we denote by z_i to the different bits, the recurrence relation is:

$$z_i = c_1 z_{i-1} + c_2 z_{i-2} + ... + c_p z_{i-p} \mod 2, \tag{12.1}$$

where $c_k = 0, 1$ for $k = 1, \ldots, p$, is a given set of binary constants and we need to set the initial values $(z_0, z_1, \ldots, z_{p-1})$. The actual random numbers are obtained first by constructing b-bits integers by joining the bits $m_0 = z_0 z_1 \cdots z_{b-1}$, $m_1 = z_b z_{b+1} \cdots z_{2b-1}$, etc. The real numbers are, as before, $u_i = m_i/2^b$.

It is clear from the recurrence relation that the numbers z_i necessarily repeat after a maximum period of $L = 2^p$ and hence the maximum available number of distinct random numbers is 2^{p-b}. How can we obtain a maximum period? A theorem tells us that an almost maximal period of $2^p - 1$ is obtained in the recurrence relation (12.1) if and only if the polynomial

$$f(z) = 1 + c_1 z + c_2 z^2 + ... + c_p z^p \tag{12.2}$$

can not be factorized as $f(z) = f_1(z) f_2(z)$ (all operations are modulus 2). Technically, it is said that $f(z)$ is primitive in GF(2), the Gaulois field. It turns out that good statistical properties can be obtained by using the simplest primitive polynomials, those of the form

$$f(z) = 1 + z^q + z^p \tag{12.3}$$

with suitable values for p and q. The recurrence relation becomes

$$z_i = z_{i-p} + z_{i-q} \mod (2). \tag{12.4}$$

If, instead of binary arithmetics we use binary logical operations, this relation is equivalent to:

$$z_i = z_{i-p} \otimes z_{i-q}, \tag{12.5}$$

where $\otimes$ is the *exclusive or* logical operation (let us recall its table: $0 \otimes 0 = 0, 0 \otimes 1 = 1, 0 \otimes 1 = 1, 1 \otimes 1 = 0$). The exclusive or operation is included in many compilers and it has the advantage that it is really fast.

A possible way to implement the algorithm is to work directly at the level of the m_i integers. First, set p initial values $m_0, m_1, \cdots, m_{p-1}$, being m_i integers with the desired accuracy (for instance, $b = 31$ bits, so we avoid the presence of negative numbers). This is implemented using another random generator. Then use the recurrence relation $m_i = m_{i-p} \otimes m_{i-q}$ which is understood by the compiler as the operation $\otimes$ is performed at the level of each bit of the integer numbers. Finally set $u_i = m_i/2^b$.

Once the maximum period has been ensured, which pairs (p, q) will give good statistical properties? A first suggested choice was $p = 250, q = 103$. The resulting generator is called R250. Its period is $2^{250} \approx 1.8 \times 10^{75}$ sufficiently large for any application. It was very popular for a time until it was shown in [61] that some spurious correlations showed up when computing the equilibrium properties of the Ising model in regular lattices, probably due to the existence of Marsaglia planes of not large enough dimension. Other values that have been suggested as producing good statistical properties are $p = 1279$, $q = 418$.

12.5 RCARRY and lagged Fibonacci generators

The family of general lagged Fibonacci generators use the relation:

$$m_i = (m_{i-p} \odot m_{i-q}) \mod M, \tag{12.1}$$

where $\odot$ denotes either a sum, a subtraction, a multiplication or an exclusive or (performed bit by bit, in this case we recover the FSR generators). When using a sum or a subtraction there is a mixing of bits and, allegedly, the statistical properties are better.

James[62], based on the ideas of Marsaglia et al. [63], has proposed the so–called RCARRY generator. It starts from a lagged Fibonacci sequence with a subtraction but adding an additional subtraction if m_i is a negative number. The algorithm is:

$$m_i = m_{i-r} - m_{i-s} - c_i \mod (M),$$

where $r > s$ and the remainder is $c_i = 1$ if $m_i \leq 0$ (before the modulus operation) and $c_i = 0$ otherwise. The subtraction mixes the different bits of the integer number m_i and the use of the remainder c_i is meant to destroy most of the correlations in

the sequence of random numbers. For the initialization of the algorithm one needs to give a sequence of r integer number $m_i,\ i = 1, ..., r$. A convenient choice is $M = 2^{24},\ r = 24, s = 10$. The period of the generator is 48 times less than the number of different states that can be represented using 24 numbers with 24 bits, or $(2^{24})^{24} \approx 10^{173}$.

12.6 Final advice

The reader might be optimistic by nature and conclude that there are some good random numbers that are better than others. Or he might be pessimistic and conclude that all random numbers generators are bad, some of them are worse than others. The truth is that a pseudorandom number generator that uses a deterministic algorithm will never pass all tests that check the goodness of the generator. The most important point, in our opinion, is not to rely blindly on the random number generator provided by "we don't know who" and use only the generators considered to be "good" in the literature. But be aware that even those good generators might have problems. If a weird result is found in your simulations it might be worthwhile to check that it does not have its origin in the random number generator, for instance, by repeating the simulation with a different generator.

Exercises

1) Program von Neumann's algorithm. Generate 1000 numbers using this algorithm and compute the average values $\langle x\rangle$, $\langle x^2\rangle$ and the correlation $\langle x_i x_{i+1}\rangle$. Do results depend on the seed m_0?
2) Use the random number provided by your favorite compiler or library and check whether the period is larger than $2^{32}\approx 4.3\times 10^9$. Generate 10^6 numbers using this algorithm and compute the average values $\langle x\rangle$, $\langle x^2\rangle$ and the correlation $\langle x_i x_{i+1}\rangle$. If the generator requires a seed, do results depend on the seed m_0?
3) Repeat the previous problem using l'Ecuyer's algorithm RANECU which uses $a = 40692$, $m = 2147483399$, $c = 0$ (purely multiplicative).
4) Same with algorithm R250.
5) Compute numerically the dimension of Marsaglia's planes for R250 and the congruential algorithm with $M = 2^{32}$, $a = 69069$, $c = 1$.
6) A famous problem states that the probability that the second degree equation $ax^2 + bx + c = 0$ has real roots is $\frac{5}{36} + \frac{\ln(2)}{6} \approx 0.254$ if a, b, c have been chosen randomly and uniformly in the interval $(0, 1)$. Use your favorite random number generator to check this result. Do not forget to include the errors of the estimator of this probability.