Random numbers and random number generators

It was a very popular deterministic philosophy some 300 years ago that if we know initial conditions and solve Eqs. of motion then the future is predictable. First, there is no way to get initial conditions for all particles even in a relatively small "practical" systems of 10^{23} particles. Second, on "human" time scales interactions with the outside world will enter the problem and we will be forced to include the outside world into the solution (along with its initial conditions), etc. The popular philosophy finally died with the rise of quantum mechanics.

With our ability to monitor only a few parameters for the system, like energy, pressure, magnetization, ... we often have to resort to the statistical, or probabalistic, description of all the other degrees of freedom we do not have under our control. Throwing dice is formally not a random experiment if it is done by the precisely engineered mashine with very precise control over the relatively short trajectory; this control is far more difficult to achieve for flipping coins. In real life, if coins are flipped by humans, the coin rotation is fast, the trajectory is long, the substrate is rough, ... it becomes impossible to predict the outcome and we start using probabilitites.

The Nature makes things random and probabalistic by employing macroscopically large number of variables and quantum mechanics. In MC methods, where simulations are based on the notion of probability, the "source of randomness" is actually a very short, completely deterministic piece of code generating "random numbers"! It is not practical to ask Nature for random numbers since the corresponding interface would be too slow for most Monte Carlo applications (Nature generated random numbers are also sold on CDs and DVDs but even this resource is too slow and short of supply). It comes as a surprise that several lines of code are capable of producing extremely long sequencies of numbers which appear to be absolutely random **For All Practical Purposes** (FPPA).

The output of the random number generator produces a variable $rndm$ which is uniformly distributed in the range $[0, 1)$. Simple techniques exist then for transforming rndm into random numbers with any other probability density distribution. By simple rescaling $c \cdot rndm$ and shift $s + c \cdot rndm$ we produce a random number uniformly distributed on the $[s, s+c)$ interval. To convert real or double precision rndm to random integer, I, taking on any value from 1 to N with equal probability $1/N$ we just take the integer part of $[rndm \cdot N]$ (here [#] stands for the closest integer smaller than #) and add unity, $I = [rndm \cdot N] + 1$.

The most important application of random numbers in MC simulations is the possibility to perform some action with the probability p. Indeed, the propability of *rndm* to be smaller than p is p, so if the action is performed only if $rndm < p$ then it is performed with the probability p.

Simple realizations of the random number generator

The *rndm*-generator should necessarily depend on the history of its own work, otherwise it will return the same number all the time. The simplest realization would be

$$i_n = f(i_{n-1}) \ ,$$

i.e. the next number depends on the previous one. To get the process started we need to provide the first number which is called a **seed**. Most *rndm*-codes work with integers in the interval $[0, m)$ which are then converted to real numbers by dividing them by m. Clearly, at best such generators may produce only m different random numbers, so they will start reproducing the same sequence again after a while. The length of the sequence before it starts repeating itself is called the generator **period**.

A widely used generator of the above mentioned form is called a **linear congruential generator** (Lehmer, 1951)

$$i_n = MOD_m(ai_{n-1} + c) \ , \tag{1}$$

where $MOD_m(k)$ is the modulo operation which returns the remainder after k is divided by m. It can generate up to m random numbers with the right choice of a and c constants. The larger m the better but unfortunately there is a limit on the maximum one-word integer—32 bit computers typicaly allow integers up to $w = 2^{31}$ (one bit for the sign in Fortran) or $w = 2^{32}$ (in C and Pascal). In addition, some languages are sensitive to the overflow problem and complain if $ai_{n-1} + c > w$. A good choice of constants requires that a and m are coprime (i.e. do not have common factors other than 1), and one of the recommended/tested sets is $a = 9301$, $c = 49297$, $m = 233280$. It generates m numbers which pass most tests on 'randomness" (see below).

Recent AMD-processors (opterons) work already with 64 bit numbers and it seems that the overflow problem will soon become history. In any case, there are tricks to handle the problem with some extra CPU time (Schrage, 1979). Consider the linear congruential generator with $m = 2^{31} - 1$ and $a = 16807$, $c = 0$ (it has the largest possible m in FORTRAN) The trick is to subtract the right number of periods while multiplying a by i_{n-1}:

- Define (as parameters) $q = [m/a] = 127773$ and $p = m - qa = MOD_a(m) = 2836$ (notice that $p < q$ in this example). The idea is to write $m = qa + p$.

- Calculate first $x = [i_{n-1}/q]$ and $y = i_{n-1} - qx$; so that i_{n-1} can be written as $i_{n-1} = qx+y$ (by definition, $y < q$ and $x \leq a$ since $i_{n-1} < m$).
- The identity $MOD_m(ai_{n-1}) = MOD_m(a(qx + y) - xm)$ may be used then to transform the first argument as $a(qx+y)-x(aq+p) = ay-xp$. Both numbers in the last expression are smaller than m and the calculation does not result in the overflow. So, in practice we perform

  $$x = [i_{n-1}/q] \ ; \ \ y = MOD_q(i_{n-1}) \ ; \ \ i_n = ay - xp \ , \ \ IF(i_n < 0) i_n = i_n + m \ ,$$

Let's call this set of operations $i_n = rndm_1()$ for brevity.

This an example of the code.

Keep learning some programming language, please. Later in the course you will have to write things yourself. I will use Fortran in my examples.
Below $i0$ is an integer defined as a common variable outside $rndm_1()$.

```
double precision function rndm₁()          ! just the name

integer, parameter :: m=2147483647
integer, parameter :: a=16807
integer, parameter :: q=127773
integer, parameter :: p=2836
integer:: x,y

x=i0/q
y=mod(i0,q)
i0=a*y-x*p
IF(i0<0) i0=i0+m

rndm₁=i0/m

return
end function rndm₁                         ! this is all
```

The linear congruential generator is too simple and in fact has some correlations between the generated numbers. Also, the sequence length is not as long as necessary. A simple solution with little CPU overhead is to make the generation of i_n dependent on many previously generated numbers $i_n = f(i_{n-1}, i_{n-2}, \ldots, i_{n-k})$. The sequence length is then the "FAPP infinity", i.e. not possible to reach in practice. The so-called **shuffled generator** (Bays and Durham, 1976), which has very good random properties, is of this "long-memory" type. It works as foolows.

- At the start of the program it has to be initialized by generating a list of L+1 random numbers:

  $$u_1 = rndm_1(\text{from seed})\ ,\ \ u_2 = rndm_1\ ,\ \ldots\ u_L = rndm_1\ ,$$

  $$y = rndm_1\ .$$

  Typically L is rather short, say smaller than 100.

- The new random number is generated by using existing y to determine the index $k = [y * L] + 1$. This index determines the new random number $rndm = u_k$, and the new value of $y = u_k$.

- Finally, the u_k value is replaced with $u_k = rndm_1$.

In essence, $rndm_1$ is used to fill in the storage boxes, and random numbers extracted from the boxes determine also which box to use next, i.e. we constantly shuffle the sequance of L numbers.

The last rndm-code I would like to discuss is the **lagged Fibbonacci generator** by Mitchell and Moore, 1958. It is also based on a long history of previously generated numbers, but uses only two of them in a simple MOD-function:

$$i_n = MOD_m(i_{n-r} \bigcirc i_{n-s})\ ,\ \ \text{where}\ \bigcirc = \text{``}+\text{''}\ \text{or}\ \text{``}\times\text{''}\ \text{operation}\ ,$$

The most common choice proved to be very good is $r = 24$, and $s = 55$. It has to be initialized as well, using, e.g. the $rndm_1$ generator. Its period is extremely long FAPP.

How random?

There are many other codes for rndm-functions, some nice some bad. It is always a good idea to have at least two "good" generators and verify that results do not change when rndm is replaced. Also, there are many standard tests for rndm and good generators should pass all of them. Tests are looking for deviations from the predictions of what "true" generators should give. For example, if rndm is perfect then

$$\langle\, (rndm)^n \,\rangle = \int_0^1 (rndm)^n d(rndm) = 1/(n+1)$$

$$\langle\, (rndm_i - 1/2)(rndm_{i+k} - 1/2) \,\rangle = 0$$

testing that the distribution is flat and that numbers in the sequence separated by k rndm-calls apart are not correlated. By average here is meant the result of a very long run

$$\langle \text{Quantity} \rangle \;=\; \lim_{M\to\infty} \frac{\sum_{i=1,M} \text{Quantity}_{run\ i}}{M} \;.$$

Problem. Perform these tests for the shuffled linear congruential and lagged Fiibonacci generators discussed above, i.e. compare your results for $\langle (rndm)^n \rangle$ with (n=1,2,3) and $\langle (rndm_i - 1/2)(rndm_{i+k} - 1/2)\rangle$ with (k=1,2,3) averaged over 10^7 runs with the ideal behavior. Please, write codes for rndm-functions yourself.

There are many other tests which check for correlations between more than two numbers, but we will not review them (if interested, → Knuth, *The Art of Computer Programming , Vol. 2, 1981*).

From uniform to arbitrary probability density distribution.

Suppose, that you need random numbers generated not according to the uniform probability density distribution $g_{rndm}(z) = 1$, but according to some other function $g(z)$. By definition, $g(z)dz$ is the probability of producing random number within the interval $(z - dz/2, z + dz/2)$. What we need is the relation $z = F(rndm)$ such that if rndm is uniformly distributed between 0 and 1 then z is distributed acording to $g(z)$. Just write down the "wish" using inverse function $rndm = F^{-1}(z)$ and differentiate:

$$d(rndm) = \left[F^{-1}(z)\right]' dz = g(z)dz \;. \qquad (2)$$

Integrating this Eq.

$$F^{-1}(z) = \int_{z_{min}}^{z} g(z)dz = rndm \; . \tag{3}$$

we obtain the functional relation between rndm and z. $F^{-1}(z)$ is nothing but the probability of producing any random number smaller than z. Unfortunately, this relation is not of the form we would like to have it. Even when $F^{-1}(z)$ can be found in a closed analytic form we still have to invert the functional relation to get $F(rndm)$.

One easy case we already did before (scale and shift): for $g(z) = 1/(b-a)$ and $z \in [a, b)$ we have

$$F^{-1}(z) = (z-a)/(b-a) \implies z = a + (b-a)\cdot(rndm)$$

The other example of interest is the exponential distribution $g(z) = ae^{-az}$ for $z \in [0, \infty)$. Following the rule

$$F^{-1}(z) = 1 - e^{-az} \implies z = -\frac{1}{a}\ln(1 - rndm) \; .$$

Similarly, for the Lorentzian distribution $g(z) = (\gamma/\pi)/(z^2+\gamma^2)$, $z \in (-\infty, \infty)$ we find

$$F^{-1}(z) = \frac{1}{2} + \frac{1}{\pi}\tan^{-1}(z/\gamma) \implies z = \gamma\tan[\pi(rndm - 1/2)] \; .$$

and for the spherical angle distribution $g(z) = \sin(z)/2$, $z \in [0, \pi)$ we find

$$F^{-1}(z) = (1 - \cos(z))/2 \implies z = \cos^{-1}(1 - 2\cdot rndm) \; .$$

Problem. Find the $z = F(rndm)$ relation such that z is distributed according to $\sim e^{a/z}/z^2$, $z \in [0, \infty)$.

The method described above is called the **transformation method**. When it works, i.e. when a closed form solution for $F(rndm)$ exists, it is the best we can do. For the Gaussian distribution

$$G(z) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-z^2/2\sigma^2} \; ,$$

the transformation method is of little help unless you wish to tabulate yourself the inverse of the error-function. However, there is a trick to get around this problem by generating two random variables with the Gaussian distribution using two standard random numbers. The solution originates from the properties of two-dimensional integrals in cylindrical coordinates. If we have two independent random variables x and y then the probability to have them in the differential interval is a product

$$G(x)G(y)\,dxdy \equiv \frac{1}{2\pi\sigma^2}e^{-(x^2+y^2)/2\sigma^2}\,dxdy = \frac{1}{2\pi\sigma^2}e^{-(r^2)/2\sigma^2}\,rdrd\phi$$

where $x = r\cos(\phi)$, and $y = r\sin(\phi)$. The problem is reduced to generating r with the probability distribution $re^{-(r^2)/2\sigma^2}/\sigma^2$, $r \in [0,\infty)$, and ϕ with the uniform distribution $1/2\pi$, $\phi \in [0,2\pi)$. Both can be easily done by the transformation method ($r^2/2$ distribution is exponential):

$$r = \sigma\sqrt{-2\ln(1-rndm)}\ , \quad \phi = 2\pi \cdot rndm\ .$$

and immediately converted back to x and y.

If the inverse relation cannot be found one may use the **rejection method**. It is absolutely general, works in all cases, and with tricks can be made quite efficient. In its simplest form it is as follows. Let the maximum value of the distribution is $g_{max} = max\{g(z)\}$ and $z \in [a,b)$.

(1) Suggest the value of z uniformly distributed on the $[a,b)$-interval

$$z = a + (b-a)\cdot rndm\ .$$

(2) Accept this value with the probability $g(z)/g_{max}$, i.e. accept it only if another random number satisfies

$$rndm < g(z)/g_{max}\ .$$

Otherwise, start over again from point (1) and continue untill the number is accepted

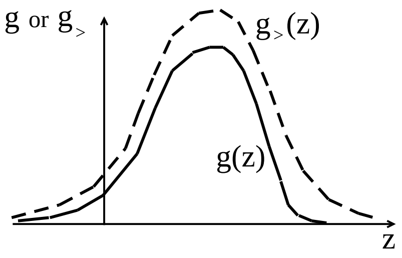
To see that the goal is achieved we compute the probability distribution of producing different z numbers which is the product of probabilities to suggest it, $dz/(b-a)$ and to accept it

$$\frac{dz}{b-a}\,\frac{g(z)}{g_{max}} = g(z)dz\,\frac{1}{g_{max}(b-a)}\ .$$

This distribution is proportional to $g(z)dz$ as required, and we do not have to worry about normalization since the answer depends on the ratio $g(z)/g_{max}$.

The rejection method can be immediately generalized to deal with infinite-range distributions and to improve its efficiency by making the initial suggestion as close to $g(z)$ as possible (this generalization is also known as a **hybrid method**). Instead of g_{max} we may introduce some function $g_>(z)$ which has two properties,

(i) $g_>(z) \geq g(z)$ for all z, and (ii) $g_>(z)$ is from the class of functions which may be dealt with by the transformation method, i.e. easy to integrate and solve for the inverse of the integral. Then



(1) Suggest the value of z distributed according to the $g_>(z)$ function.
(2) Accept this value with the probability $g(z)/g_>(z)$. Otherwise, go back to point (1).

The proof is exactly as before: the probability of z to be accepted is proportional to $[g_>(z)dz] \cdot [g(z)/g_>(z)] = g(z)dz$. It is also easy to estimate the efficiency of the method. The probability to accept the suggested number is, on average, $\int g(z)dz$, and each time we use two random numbers to get it. Thus on average we need $N = 2/\int g(z)dz$ rndm-numbers to produce z. Since $g_>(z)$ is normalized to unity (we need this to be able to use the transformation method), and $g_>(z) \geq g(z)$, we have $\int g(z)dz < 1$ and $N > 2$ ($N = 2$ would mean that $g_>(z) = g(z)$ which we assume is not the case).

The hybrid method also illustrates an idea of "reweighting" which we will discuss later in the course. In it's most general form the idea is to generate a distribution similar but somewhat different from the desired one and then "reweight" the corresponding contribution to the statistics by a factor $g_{desired}/g_{actual}$. Why doing it? Well, simplicity and efficiency are among the reasons.

As an example, consider the problem of evaluating the integral

$$I = \int_0^\infty g(z)dz \ .$$

For one of the functions which can be simulated by the transformation method, $g_0(z)$, we know how to take the integral

$$I' = \int_0^\infty g_0(z)dz \ ,$$

If we pretend that both distributions were generated by the hybrid method, then their efficiencies would relate as I/I'. On another hand, the first distribution is $g(z)/g_0(z)$ times more likely to be rejected then the other, thus $I/I' = \langle g(z)/g_0(z)\rangle_z$ where the average is taken over the process of suggesting different z values (we do not need to perform actual rejections in this simulation). Finally, we may choose $g_0(z)$ itself to be the generating function normalized to unity and get

$$I = \langle \ g(z)/g_0(z) \ \rangle_z \ , \tag{4}$$

Now, the average is over the process of randomly generating z with the distribution $g_0(z)$.

One may complain that we are restricted here to study only positive definite $g(z)$. In fact, Eq. (4) is general, and works equally well for non-positive $g(z)$ too. Simply imagine that that I is evaluated piece by piece on intervals where $g(z)$ does not change sign, i.e. if there are k intervals and $g(z)$-sign on interval i is s_i then

$$I = \sum_{i=1}^{k} s_i I_i$$

where each contribution

$$I_k = \int_{\in i} |g(z)| dz$$

can be calculated using Eq. (4) with $g(z) \to |g(z)|$, thus

$$I = \sum_{i=1}^{k} s_i \langle \ |g(z)|/g_0(z) \ \rangle_{z\in i} = \sum_{i=1}^{k} \langle \ g(z)/g_0(z) \ \rangle_{z\in i} = \langle g(z)/g_0(z) \ \rangle_z \ .$$

Problem. Implement this calculation in a code and determine

$$I_1 = \int_0^\infty \frac{sin^2(z)}{z \sinh z} \, dz \ ; \qquad I_2 = \int_0^\infty \frac{sin(z)}{z \cosh z} \, dz$$

using $g_0(z) = e^{-z}$. Not the most eficient way of doing low-dimensional integrals, but still ...

Accidentally, if you finished this problem you finished the first realistic MC simulation in this course, the rest is just a little more elaborate but similar in spirit.