Lecture 8

- Random numbers
- Non-uniform probability distributions
- The Metropolis algorithm

Random numbers:

It might appear strange to ask a computer, which carries out deterministic procedures, to produce a random result. In fact, it simply cannot. Some people prefer to therefore talk of pseudo-random numbers when referring to the ones generated by computers. We will not make that distinction there.

So what is a random number generator? The definition is operational. Any two random number generators should, on the same machine, achieve the same physical result for a code that assumes they are random number generators.

The definition may seem (and is) circular, but it is just a fact that there are out there many generators that behave in such a way for a wide variety of physical applications. Therefore if you create a new one, there is a good test-bed against which to calibrate.

Knuth, Seminumerical Algorithms, 2nd Edition, Chapter 3

Most compilers come with a system-supplied random number generator, typically named "ran" or similar names, and it is usually invoked with a "seed" (an integer number), like `ran(iseed)` in Fortran.

These generators ("linear congruential generators") work by considering a sequence:

$$I_{j+1} = aI_j + c \pmod{m}$$

So the "iseed" one supplies provides the first term in the sequence, it runs for a while and returns the "random number" and updates the value of iseed so one is ready to run again. The random number is typically I_{j+1}/m and will therefore be between 0 and 1.

Such a sequence will eventually repeat, but the largest the m the less frequently this will happen.

There doesn't exist an "universally good" random number generator.

Typically m will be close to the largest integer the machine can handle, like 2^{32}.

The numbers are not really random. If one uses k random numbers to give the coordinates of a point in k-dimensional space, one will see the points grouping in k-1 dimensional hyperplanes. There will be at most $m^{1/4}$ such planes. So for instance the number of planes in a three dimensional random plot will be around 1600. If you are sampling a small fraction of space in your physics problem, the planes will be quite noticeable.

Worse, certain combinations of a and m conspire to produce particularly bad random number generators. In particular IBM in the 60's produced one such choice that was copied in a widespread way through many other machines.

Most modern compilers have better random number generators, that are adequate for most general purpose applications.

What if you want to write your own generator? Tricks are needed. One is interested in using the largest possible integer numbers that the computer can handle. Yet operations between such large numbers have to be done carefully not to exceed machine precision. One possibility is to code things in assembler-level language. If one wants to write things in a high level language (Fortran, C), one needs to recur to tricks involving integer arithmetic. These are not terribly insightful so we will not discuss them in detail here.

For instance, Schrage's algorithm uses c=0 and the result,

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{if it is} \geq 0, \\ a(z \bmod q) - r[z/q] + m & \text{otherwise} \end{cases}$$

Where, $m = aq + r, \quad$ i.e., $\quad q = [m/a],\ r = m \bmod a$

And Numerical Recipes implements this with

$q = 127773$ and $r = 2836$.

```
FUNCTION ran0(idum)
INTEGER idum,IA,IM,IQ,IR,MASK
REAL ran0,AM
PARAMETER (IA=16807,IM=2147483647,AM=1./IM,
     IQ=127773,IR=2836,MASK=123459876)
    "Minimal" random number generator of Park and Miller. Returns a uniform random deviate
    between 0.0 and 1.0. Set or reset idum to any integer value (except the unlikely value MASK)
    to initialize the sequence; idum must not be altered between calls for successive deviates
    in a sequence.
INTEGER k
idum=ieor(idum,MASK)          XORing with MASK allows use of zero and other simple
                                  bit patterns for idum.
k=idum/IQ
idum=IA*(idum-k*IQ)-IR*k      Compute idum=mod(IA*idum,IM) without overflows by
                                  Schrage's method.
if (idum.lt.0) idum=idum+IM
ran0=AM*idum                  Convert idum to a floating result.
idum=ieor(idum,MASK)          Unmask before return.
return
END
```

One has to guard against idum=0 since we chose c=0.

Proof: $aI_i \bmod m = I_{i+1}$, then

$$I_{i+1} = aI_i - [\frac{aI_i}{m}] \cdot m = aI_i - [\frac{aI_i}{(aq+r)}] \cdot (aq+r)$$

Let's see what is happening in the square brackets

$$\frac{aI_i}{aq+r} = \frac{I_i}{q(1+\frac{r}{aq})} = \frac{I_i}{q} \cdot \frac{1}{(1+\frac{r}{aq})}$$

Let's now remember that $\frac{r}{aq}$ is a small number as $r < a$ by definition and let's also demand that $r < q$. Thus, $\frac{r}{aq} << 1$.
For a small ϵ the following Taylor series expansion can be written

$$(1+\epsilon)^{-1} = 1 - \epsilon$$

Then,

$$\frac{aI_i}{aq+r} = \frac{I_i}{q(1+\frac{r}{aq})} = \frac{I_i}{q} \cdot \frac{1}{(1+\frac{r}{aq})} = \frac{I_i}{q}(1-\frac{r}{aq})$$

or

$$\frac{I_i}{q}(1-\frac{r}{aq}) = \frac{I_i}{q} - \frac{I_i}{aq} \cdot \frac{r}{q}$$

$$\frac{I_i}{q}(1 - \frac{r}{aq}) = \frac{I_i}{q} - \frac{I_i}{aq} \cdot \frac{r}{q}$$

$\frac{I_i}{aq} \approx \frac{I_i}{m} < 1$ and $\frac{r}{q} << 1$ it is garanteed that it can be different from $[\frac{I_i}{q}]$ only by 1. Thus, the integer taken of $\frac{I_i}{q}(1 - \frac{r}{aq})$ will be either the same as integer of $\frac{I_i}{q}$ or less than this integer by 1.

Now it is clear that

$[\frac{aI_i}{m}] = [\frac{I_i}{q}]$ when neglecting of ϵ does not effect an integer part and
$[\frac{aI_i}{m}] = [\frac{I_i}{q}] - 1$ when omitting ϵ increased the result by 1.

Eventually we can state that

$$aI_i \mathrm{mod}\ m = aI_i \mathrm{mod}\ q - r[\frac{I_i}{q}], \qquad \text{if} \qquad aI_i \mathrm{mod}\ q - r[\frac{I_i}{q}] \geq 0$$

$$aI_i \mathrm{mod}\ m = aI_i \mathrm{mod}\ q - r[\frac{I_i}{q}] + m, \qquad \text{if} \qquad aI_i \mathrm{mod}\ q - r[\frac{I_i}{q}] < 0$$

Not even this algorithm is bulletproof. Numerical recipes discusses six other generators. An issue to keep in mind is how fast they execute. The more sophisticated the algorithm, the slower it will be.

Non-uniform probability distributions:

The random number generators we just discussed are based on a (theoretical) uniform probability,

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

If we now consider a function *y=f(x)* its values will be distributed with a probability given by the fundamental transformation law of probabilities, i.e.,

$$|p(y)dy| = |p(x)dx|$$

$$p(y) = p(x) \left| \frac{dx}{dy} \right|$$

Suppose you are given *p(y)=f(y)* and you wish to generate random numbers with such probability. Using the previous equation with *p(x)=1* we get,

$$\frac{dx}{dy} = f(y) \qquad \Rightarrow x = F(y)$$

With F(y) the indefinite integral of f(y), or, inverting, $\boxed{y = F^{-1}(x)}$

These manipulations look trivial, but in practice can be cumbersome: one needs to analytically integrate the probability density (usually feasible) but then *invert* the resulting function (easily very hard).

All this can be generalized to many dimensions straightforwardly by recalling that,

$$p(y_1, y_2, \ldots) dy_1 dy_2 \ldots = p(x_1, x_2, \ldots) \left| \frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)} \right| dy_1 dy_2 \ldots$$

However, it is obvious that in practice we will need a different strategy.

One such strategy is the **Von Neumann rejection**. Suppose you wish to generate points with a probability $w(x)$ with x in [0,1]. The method consists in considering a function $w'(x)>w(x)$ in the same interval and sprinkling points in two dimensions uniformly filling the area below $w'(x)$ (a simple choice is to consider $w'=constant$).

One then tests if the sprinkled point lies below $w(x)$.
If it does one accepts it, if it doesn't one rejects it. It is clear that the resulting points will be uniformly distributed below $w(x)$. By retaining just the "x-coordinate" value of each point we end up with a table of numbers with the desired probability density.

There exist many tricks for handling probability distributions popular in physics, Gaussian, Binomial, Poisson, etc. Numerical Recipes gives several examples.

Gaussian distribution:

$$w(x) = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$$

Doing the inversion requires evaluating the "error function". A smarter way is to use the *central limit theorem*, which states that the sum of a large number of uniformly distributed random numbers will approach the Gaussian distribution.

```
      SUBROUTINE DIST(GAUSS)
      INTEGER SEED
      DATA SEED/39249187/

      GAUSS=0.
      DO 10 I=1,12
        GAUSS=GAUSS+RAN(SEED)
10    CONTINUE
      GAUSS=GAUSS-6.
      RETURN
      END
```

In this example, from Koonin & Meredith they add up 12 normal distributions (since the variance of the uniform distribution in [0,1] is 1/12 and the mean is 6 this produces a Gaussian centered at 6 with variance 1).

http://einstein.unh.edu/~meredith/compphys/fortran/chap8e.for

Another Gaussian trick, is the Box-Mueller method, is based on the observation that the coordinate transformation,

$$y_1 = \sqrt{-2\ln x_1}\cos 2\pi x_2 \qquad x_1 = \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right]$$

$$y_2 = \sqrt{-2\ln x_1}\sin 2\pi x_2 \qquad x_2 = \frac{1}{2\pi}\arctan\frac{y_2}{y_1}$$

Has Jacobian,

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}}e^{-y_1^2/2}\right]\left[\frac{1}{\sqrt{2\pi}}e^{-y_2^2/2}\right]$$

So we see that each y is individually distributed with a Gaussian probability.

The Metropolis algorithm (Metropolis, Rosembluth[2], Teller[2] (1953))

If the dimensionality involved is high, all previous methods become quite impractical.

Supose you wish to generate points **X** with probability density $w(\mathbf{X})$.

This algorithm proposes to generate such points by taking a random walk through the space of **X**'s in such a way that when the walk is a long one the points visited are sprinkled with the desired probability.

The random walk proceeds in the following way. Suppose you have taken n steps and are sitting at X_n. To go to $X_{n+1}$the walker takes a random trial step (for instance by choosing points randomly in a cube of small size). The trial step is then accepted or rejected according to the ratio $r=w(X_t)/w(X_n)$.

If the ratio is larger than one the step is accepted. Otherwise it is accepted with probability r.

This latter step is easily accomplished by generating a random number η in [0,1] and accepting the step if η<r.

To prove that the algorithm does indeed generate a sequence of points distributed according to w, let us consider a number of walkers starting from different initial points and moving independently.

If $N_n(\boldsymbol{X})$ is the density of walkers at **X** after n steps then the net number of walkers moving from **X** to **Y** in the next step will be,

$$\Delta N(X) = N_n(X)P(X \to Y) - N_n(Y)P(Y \to X)$$

The equilibrium densities are therefore related by,

$$\frac{N_e(X)}{N_e(Y)} = \frac{N_n(X)}{N_n(Y)} = \frac{P(Y \to X)}{P(X \to Y)}$$

And the system is **stable** (meaning it returns to equilibrium if perturbed).

It remains to be proved that the distribution one achieves in equilibrium is the desired one.

The probability of taking a step from X to Y was,

$$P(X \to Y) = T(X \to Y)A(X \to Y)$$

Where T is the probability of taking a trial step from X to Y and A is the probability of accepting it.

If Y can be reached from X in a single step, and the walker is moving isotropically then,

$$T(X \to Y) = T(Y \to X)$$

So the equilibrium distribution satisfies,

$$\frac{N_e(X)}{N_e(Y)} = \frac{A(Y \to X)}{A(X \to Y)}$$

Now, if $w(X) > w(Y)$ then $\mathrm{A}(\mathrm{Y} \to X) = 1$ and $A(X \to Y) = \dfrac{w(Y)}{w(X)}$

And, if $w(Y) > w(X)$ then $\mathrm{A}(\mathrm{X} \to Y) = 1$ and $A(Y \to X) = \dfrac{w(X)}{w(Y)}$

Therefore in either case,

$$\frac{N_e(X)}{N_e(Y)} = \frac{A(Y \to X)}{A(X \to Y)} = \frac{w(X)}{w(Y)}$$

So indeed the equilibrium densities are distributed with the desired probability weights.

-Any probability rule for the stepsize will do. For instance T(X->Y)=w(Y) would be optimal (but of course, hard to do).

-The art of choosing the rule consists in minimizing the number of lost steps.

A problem of this algorithm is that although the points generated are sprinkled with the desired probability if the walk is a long one, each point is statistically correlated with the next one. So for instance, if one wished to use this method to compute an integral using the Monte Carlo method, the variance estimate of the error we introduced would be incorrect, since it assumes points are not correlated with each other.

A way around this problem is to sample the integral using points generated in the random walk sampled at a fixed interval of steps large enough that the resulting points have very little correlation. Of course, this is where things become more an art than science. A similar remark corresponds to the choice of starting point of the random walk. A lousy choice will yield a distribution that is only vaguely related to the desired one.

Summary

- Random numbers: if problems suspected, try a different generator.
- Von Neumann rejection: simple and quick.
- Metropolis algorithm: especially useful in higher dimensions (statistical mechanics, particle physics on the lattice).