Random numbers

1. Why random numbers?

One of the most important things to be able to do with a computer is to draw random numbers. Technically, at least in most cases, these are really *pseudo-random*, not truly random. The reason we want to be able to do this is that often our calculation, or tests of our calculation, or simulation of experimental results, requires some random values. You may be surprised at how common this is. In fact, a famous and extremely useful book in early computation and cryptography was *a published book* of random numbers used for this purpose, if you can believe that.

Most computer random number generators are not truly random, but *pseudorandom.* In fact, they are usually initialized by a seed, and then there is a deterministic procedure which produces a sequence of numbers. This means that your "random" sequence is in fact reproducible if you start from the same seed (this can be a very good and important thing!). You should be careful about picking the seed, if you want it to be different each time.

There are ways of generating more truly random numbers but these typically involve utilizing some truly stochastic physical process.

2. Probability Distributions

When we draw a random number, we draw it from some *distribution* $P(x)$, which is the probability per unit x of drawing x. Distributions like $P(x)$ are constrained to have the property:

$$\int_{-\infty}^{\infty} \mathrm{d}x \, P(x) = 1 \tag{1}$$

That is, when you draw a random number, the probability that you have drawn a number between negative and positive infinity should be unity.

For example, standard random number routines often draw randomly from a uniform distribution with $P(x) = 1$ for $0 < x < 1$. This means that if you draw a large number of random numbers, and plot their histogram, it will eventually be flat. Other distributions can be arranged, as we will see.

3. Linear Congruent Method

This is was once a common simple random number generator which gives integers between 0 and $M - 1$. If you start with some integer x_i, you generate the next as:

$$x_{i+1} = (ax_i + c) \bmod M \tag{2}$$

where a, c, and M are parameters.

In the notebook, I show a couple of examples of this, with both bad and good choices for a and c.

Of course, if you use more than M random numbers, your random number sequence is going to start repeating itself — under this method, any time you repeat a number, you are restarting the loop because you will get back the same next number, and the same after that, etc.

The choice of a and M are restricted such that their multiplication should not exceed the highest integer available.

To get a uniform distribution between 0 and 1, you then can divide the x values by M.

The linear congruent method or its variants is not anymore a good standard. The NumPy random routines use something different, which is called the Mersenne Twister algorithm. This is basically a more complex, but still linear transformation. It still has the property that the random generator has a particular *state* at each step, that is updated over time. In the case of linear congruent, the state is fully expressed by the last random number. The *improvement* over linear congruent is that the transformation is based on a longer sequence than just the last random number. This means that there are more internal states of the random generator, so a longer period — i.e. a given random number generated won't *always* be followed by the same successor each time.

4. Assessment of Randomness and Uniformity

For any new random number generator, you should test it has the properties you want. I have found it fairly rare for random number generators to be bad.

The first thing worth doing is checking that the distribution is actually uniform. In the notebook, I show how to do this with a histogram. You can also check the randomness in the following way, also demonstrated in the notebook. Make an *ensemble* of random sets, and bin in a set of small-ish bins, but expecting a few values to land in each bin.

The *expectation value* of the number of values falling in each bin is:

$$\bar{N}_{\rm bin} = E(N_{\rm bin}) = \frac{N}{n_{\rm bins}} \tag{3}$$

for N points over $n_{\rm bins}$ bins.

For a reasonable number of bins and a large enough N the *distribution* of values in each bin will be approximately uncorrelated with each other and will be a *Poisson distribution.* This distribution can be written:

$$P(N_{\rm bin}) = \frac{\bar{N}_{\rm bin}^{N_{\rm bin}}}{N_{\rm bin}!} \exp\left(-\bar{N}_{\rm bin}\right) \tag{4}$$

The variance of a Poisson distribution is:

$$\mathrm{Var}\left(N_{\mathrm{bin}}\right) = \bar{N}_{\mathrm{bin}} \tag{5}$$

so the standard deviation is $\sqrt{\bar{N}_{\mathrm{bin}}}$.

Note that for large $\bar{N}$, the Poisson distribution becomes Gaussian. This is a specific example of the very general Central Limit Theorem, which states that the distribution of the sum of random, uncorrelated variables tends to Gaussian as the number of variables gets high, no matter what the distribution of the variables (as long as they have means and variances, I think).

Explain based on the central limit theorem why a Poisson distribution becomes Gaussian in the limit of large $\bar{N}$.

We can estimate this standard deviation from the ensemble to make sure it has this scaling.

This method depends on the binning. In particular, it can miss correlations between numbers on smaller scales than the bin size. Also, it doesn't look for direct correlations.

A method described in the book which is very good to understand is to look at the near neighbor correlations:

$$C(k) = \frac{1}{N}\sum_{i=1}^{N} x_i x_{i+k} \tag{6}$$

for various choices k. As the book describes, the expectation value for this quantity is 1/4, independent of k. If you take an ensemble of cases with different N, the uncertainty should go down as $1/\sqrt{N}$. Testing these facts tests the uniformity and randomness of the distribution.

5. Application: random walk

Pretty much the simplest application you can imagine is a random walk. Let's imagine you start at $x = 0$. Then take a random step between -0.5 and 0.5, then another, etc., up to N total.

What sort of physical process does this correspond to?

Qualitatively, if you had an ensemble of random walks, how does the distribution of x evolve as the number of steps increases? Quantitatively, how do you think the width of the distribution increases with N?

Can you derive quantitatively this scaling?

What is r for the example?

I implement this example in the notebook. You can see in the plots there a very general result, which is that the distribution, though it starts out (for $N = 1$) uniform, rather quickly becomes Gaussian.

Why does the distribution become Gaussian?

6. Nonuniform random distributions

Nonuniform random distributions are also useful. Luckily they can in many cases be generated by starting with a uniform distribution.

Generically, this can be done as follows. The distribution of uniform points is:

$$f_r(r)\mathrm{d}r = \mathrm{d}r \quad \text{for } 0 < r < 1 \tag{7}$$

and we want some other distribution of, say, x: $f_x(x)\mathrm{d}x$.

For each little differential, the equality must hold:

$$|f_r(r)\mathrm{d}r| = |f_x(x)\mathrm{d}x| \tag{8}$$

where the absolute values are necessary because the x and r may be negatively correlated.

Then for some desired $f_x(x)$:

$$\left|\frac{\mathrm{d}x}{\mathrm{d}r}\right| = \left|\frac{1}{f_x(x)}\right| \tag{9}$$

from which we can derive a transformation $x(r)$.

Then if we choose r uniformly, we can apply the transformation $x(r)$ and the resulting variable is distributed as $f_x(x)$.

For example, we can derive that to achieve:

$$f_x(x) = \exp(-x) \tag{10}$$

for $x > 0$, we need the transformation $x(r) = -\ln r$.

Now, in practice this isn't always so simple. A very common case is generating Gaussian random variables. NumPy actually has a routine to do this, but it is useful to understand what sort of things underlie these routines. In the case of NumPy, what underlies it is something called the Box-Muller transformation.

If you try to implement the method described above in order to get a normal distribution:

$$f_x(x) \propto \exp\left(-x^2/2\right) \tag{11}$$

you will find:

$$r(x) = \int_{-\infty}^{x} \mathrm{d}x' \exp(-x'^2/2) \tag{12}$$

and this integral doesn't have any simple analytic form, which means you can't invert it to get $x(r)$.

Instead, you can generate two random uniform variables together, and transform in the 2D plane to two independent random normal variables:

$$\begin{aligned} y_1 &= \sqrt{-2\ln x_1}\cos 2\pi x_2 \\ y_2 &= \sqrt{-2\ln x_1}\sin 2\pi x_2 \end{aligned} \tag{13}$$

The basic idea is that you want a separable 2D distribution:

$$f(y_1, y_2)\mathrm{d}y_1\mathrm{d}y_2 = \exp(-y_1^2/2)\exp(-y_2^2/2) \tag{14}$$

which can be written in terms of the radial coordinate $y = \sqrt{y_1^2 + y_2^2}$ and angular coordinate θ:

$$f(y, \theta)\mathrm{d}y\mathrm{d}\theta = \exp(-y^2/2)y\mathrm{d}y\mathrm{d}\theta \tag{15}$$

In θ, the distribution should be uniform. This justifies the trigonometric parts of the definition of y_1 and y_2.

To justify the prefactors we can write:

$$\begin{aligned} \frac{\mathrm{d}y}{\mathrm{d}r} &= \frac{1}{f(y)} \\ \mathrm{d}r &= \mathrm{d}y f(y) \\ \mathrm{d}r &= \mathrm{d}y y\exp(-y^2/2) \\ r(y) &= \left|\int_y^\infty \mathrm{d}y' y'\exp(-y'^2/2)\right| \end{aligned} \tag{16}$$

which *is* integrable analytically! So:

$$r(y) = \exp(-y^2/2) \tag{17}$$

or

$$y(r) = -2\ln(r) \tag{18}$$

This justifies the prefactors in the definition of y_1 and y_2.

Because the 2D distribution is separable, we are guaranteed that y_1 and y_2 are independent, so we have gotten two independent Gaussian-distributed variables from two independent uniform-distributed variables. The actual implementation avoids the use of trig functions in a clever way, but I won't describe that here.

In fact, if you look at the latest versions of NumPy, it uses a so-called "Zigurrat" implementation. This is a form of rejection sampling that creates a distribution that follows a piece-wise constant function, which requires two uniform deviates. The piece-wise distribution is designed to just slightly exceed the Gaussian (with $\sigma = 1$). It then uses a rejection algorithm. This is a lot more complicated code than Box-Muller, but it turns out to be a lot faster.

This is a pattern you'll see in most modern technology—often the simple, elegant solutions give way to almost bizarrely complex solutions that end up being more robust, faster, and accurate.