Monte Carlo Methods

Sheila Kannappan, August 2014

"Monte Carlo methods" is a term covering pretty much any use of pseudo-randomness to help solve any kind of problem. – Niall O'Higgins

Copy all ".py" files in /afs/physics/users/s/sheila/public/reu/montecarlo/ to your own working space – these files include partial answers to all exercises below, left incomplete for you to finish. Select exercises have solutions provided with a ".solns" extension. You should perform this tutorial in Anaconda under a Windows/Mac OS or in ipython under linux (typing "ipython" starts python 2.7 in an interactive mode that is very user-friendly; use "run code" to execute code.py).

I. Random Number Generators

There are two ways to generate random numbers:

- physical measurements that are expected to be random (e.g., coin flips)
- computational algorithms that produce long sequences of apparently random results, in fact completely determined by an initial "seed" value

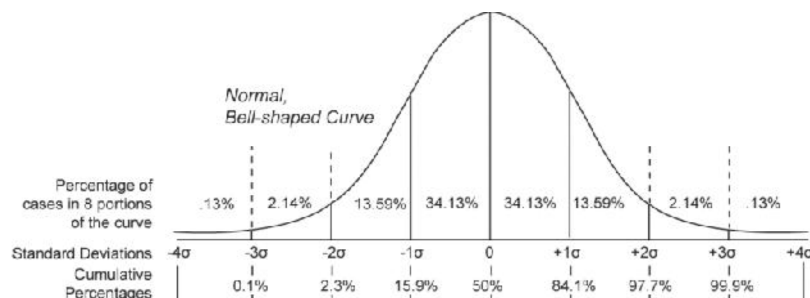
The latter are often called *pseudo-random* number generators.

Please look over the description of the "random" package for python here:
http://docs.python.org/2/library/random.html
(Check out the Mersenne Twister on the internet – it's not an amusement park ride!)

Exercise 1: Use *random.random* to generate a variable x consisting of 10 random numbers between 0 and 1. Repeat to create a second random variable y, and plot x and y against each other. Verify that there is no correlation.

Exercise 2: Use *random.seed* to control the random numbers in x and y such that they are identical. Plot x and y against each other and verify that there is a perfect correlation.

Exercise 3: Use *random.gauss* to generate a variable u (for "uncertainty") consisting of 1000 random numbers with mean zero and standard deviation σ = 1. Create a histogram of the values to verify that they look like a Gaussian distribution $1/(\sigma\sqrt{2\pi})\, e^{-\frac{u^2}{2\sigma^2}}$.



The Gaussian distribution is the most commonly used model for random uncertainties (non-systematic errors/noise) in data. In particular:

(i) the error bars on data values are typically set to equal the expected $\pm1\sigma$ variations due to random measurement errors/noise
(caveat: some research fields use Ë2† error bars)

(ii) the signal-to-noise (S/N) ratios for data values representing "detections" are typically given in terms of the background noise σ (i.e., S/N=3 means S=3σ)
(caveat: if the signal is extended in time/space/}/etc., it is really a sum of several data points and you must use error propagation rules)

From the diagram, we see a S/N>3 detection has only 0.1% probability of occurring by chance, so we say it is detected "at 99.9% confidence." For data values, the error bars are referred to as "confidence intervals." From the diagram, $\pm1\sigma$ corresponds to the "68% confidence interval."

The python package "numpy" enables array math, such as you may have used in matlab or IDL. The solutions to Exercises 3 & 4 illustrate some of what you can do with numpy.

Import numpy and use it to compute and overplot the expected Gaussian function shape on top of the histogram you made.

Exercise 4: Use numpy to convert u into an array and use numpy's *where* function to determine what percentage of the time the variable u lies inside $\pm1\sigma$. If you have an array of data with error bars equal to u, how often should the fit line go through the error bars?

II. Areas or Volumes of Enclosed Regions

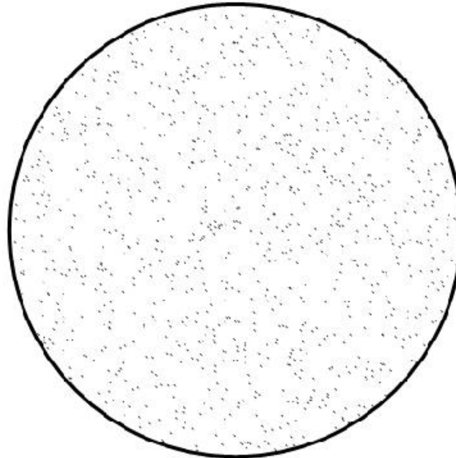
A basic application of random number generation is in measuring the areas or volumes of enclosed regions, especially non-rectangular regions for which a direct measurement would be difficult. The method is to choose points randomly in a rectangular region enclosing the region of interest, then find the fraction of the points that land inside the region of interest in order to assess its subarea/subvolume.

Exercise 1: Use *random.uniform* to measure the area of a circle with radius 1 and thus to measure the value of π. How many darts do you need to get a good, consistent estimate?

Exercise 2: Use numpy's array version of random to measure the area under the Gaussian from -1σ to $+1\sigma$. Think about why this area is equal to the percentage of u values between $\pm1\sigma$ even though u was created with *random.gauss*.

III. Random Selection from a Non-Uniform Distribution

It may happen that you want to select random numbers from a distribution of your own. For example, suppose I want the distribution of radii for a set of points drawn randomly from within a circle as shown below.



The probability of a point having a given radius increases with the area of the annulus that radius lies in, so $p(r)dr = \frac{2\pi r dr}{\pi R^2}$ where R is the radius of the circle. (Note that the integral $\int_0^R p(r)dr = 1$ as is required for a probability distribution.) The trick to computing the (non-uniform) probability distribution for *r* is to map values *x* from a uniform distribution [0, 1] onto the values of *r* in such a way that the correct frequency of values is produced. A one-to-one mapping in which the *integrated* probability out to *r* in [0, R] is equal to the integrated probability out to *x* in [0, 1] does the trick.

Exercise 1: First, use numpy's version of *random* to select radii randomly in a circle by the above method. Second, compare the distribution of radii selected by the above method to the distribution of radii obtained by selecting "hits" in a circle as in Exercise 1 from Part II. (Note – the second task requires that you generate a new block of code, not just tweak the code provided. You should find bits of earlier code that you can copy/imitate/modify to make an array of radii, then plot the new radii in a histogram on top ofo a histogram of the original radii.) How do the histograms compare? Explain.

Exercise 2: Use numpy's version of *random* to select values from a Gaussian distribution using the above method. In this exercise you are essentially recreating *random.gauss*. Hint: the integral of a Gaussian function centered on 0 is

$\int_{-\infty}^{u} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{u^2}{2\sigma^2}} du = 0.5 + 0.5 \times erf(u/\sqrt{2})$. You can import "erf" from scipy.special.

IV. The Chi-Squared (χ^2) Distribution (Optional Advanced Topic)

The value of χ^2 quantifies the comparison of the *residuals from expected values* (numerator below, describing deviations of observed data points from a model) to the *expected residuals* (denominator below, the uncertainties for the data points).

$$\chi^2 = \sum_i \frac{(O_i - E_i)^2}{\sigma_i^2}$$

As an aside, in the special case of Poisson distributed data, that is, data for which the measurement is equal to the number of "counts," it is also true that $†_i^2 = E_i = N$ where N is the number of counts. However that fact is not important for the present investigation.

Exercise 1: Use the provided Monte Carlo experiment to build up an understanding of what a single χ^2 value is and how repeated experiments with random variations yield a distribution of χ^2 values. Answer the questions embedded in the code. Not that it is often assumed that the "reduced" χ^2 – χ^2/N – should come out to be near 1 for a good fit (to see why, study the equation above). However, the definition of "near 1" depends on N, as shown in the experiment. How much you should expect a reduced χ^2 to deviate from 1?

V. For Further Inquiry

Random number generation is useful in many contexts. For example, you may wish to generate mock data sets with realistic scatter to test algorithms. Simulated data play an important role in planning and testing experiments.
http://www.ligo.org/news/blind-injection.php

Another technique you may want to try is "bootstrapping," actually a family of techniques all of which use random resampling of a real data set to estimate the uncertainties on parameters or model fits characterizing that data set.
http://en.wikipedia.org/wiki/Bootstrapping_%28statistics%29

Yet another application is "simulated annealing" – a random search strategy for finding the global optimal solution in the presence of multiple local optimal points. It's not as foolproof as an exhaustive search, but it may be the right speed/reliability compromise.
http://en.wikipedia.org/wiki/Simulated_annealing