P395: Guide 1: Random Numbers

Getting Started

Start up a new notebook and title it 'Guide1_your_name'.

This week for the first part of the activity guide, you will need to have interactive plots.

1. To get interactive plots, enter `%matplotlib widget` and execute the cell.
2. Import matplotlib and numpy

```
import matplotlib.pyplot as plt

import numpy as np
```

NOTE: with interactive plots turned on, if you are making a new plot, you need to turn off the previous plot by clicking the 'power button' on the top right of the plot.

Get in the habit of putting in headings and sub-headings (ideally one for every sub-task that produces output) to make your notebooks more readable.

Also get in the habit of liberally commenting your code (particularly nontrivial syntax, logic, or flow control), as this will help remind you what you were doing/trying to do, and help guide anybody else reading your code (notably, the marker).

Introduction to random numbers:

Random numbers are required many times in computation. We use them any time we want to simulate a system with stochastic processes where the fluctuations are important. We'll also see that we use them as a way of sampling states from a system, known as Monte Carlo sampling after the French city that is famous for its casinos. Random numbers will be used many times throughout this course, and likely in your future research.

A physical process like radioactive decay is random in that we can't predict exactly when the next atom will decay but we can nevertheless characterize to a high degree of precision the statistics of these events. In order to model such a random process we need to generate random numbers. Ideally, a set of random numbers should be uncorrelated, i.e., they never repeat themselves and what was generated before has no bearing on what will be generated in the future. Unfortunately, doing this on a computer is hard, and the best we can hope to achieve are what are known as pseudo-random numbers. These are sequences of numbers that appear 'random' but do eventually repeat after a certain period. Much research has developed algorithms with extremely long periods. Python's NumPy library has random number generators that are fast and use the top algorithms with long periods. In this lab, we will briefly look at a simple (but not great) algorithm for generating random numbers, with the rest of the time spent using Python's random number generators to explore some random processes such as radioactive decay and random walks.

Task 1: A simple random number generator

Pseudo-random number generators produce strings of numbers that will eventually repeat. If you are simulating a process where the number of calls exceeds this period, you will not be simulating a true random process. One of the simplest random number generators is the *linear congruential generator* that generates a new number x_{i+1} from the previous one x_i using the equation

$$x_{i+1} = (ax_i + c) \bmod m$$

where *a* and *c* are integers and *m* is typically a large prime (and mod means *modulo* arithmetic – done in python with the '`%`' sign). By dividing the resulting number by *m*, you get a real number between 0 and 1. Through a judicious choice of *a, c,* and *m*, this sequence can have a relatively long period and show few correlations. Correlations can be seen by plotting x_{i+1} versus x_i. Poor random number generators will show correlated bands and do not uniformly cover the plane of (x_i, x_{i+1}) (though sometimes you must look in 3D (i.e., (x_i, x_{i+1}, x_{i+2})) and plot three consecutive points to see banding). Let's look at how the linear congruential generator performs.

1. Keep things organized. Make a text cell with a new heading such as 'Task 1: Simple random number generator'.
2. Using the equation for the linear congruential generator above, start with a seed of $x_0 = 1$ and generate the next 12 numbers in the sequence, using *a* = 12, *c* = 0, and *m* = 143.
3. When does the sequence start to repeat? Explain why it has this period given this choice of parameters. Type your answer into a text cell.
4. Write a function that returns the next number in a sequence using the linear congruential equation above. It should take, *a*, *c* and *m* as parameters and the current value of the sequence x_i and return as output x_{i+1}.

In the 1970's and 80's, a popular choice for parameters was *a* = 65,539, *c* = 0, and $m = 2^{31}$. This generator was called "Randu" and was frequently used in research.

5. Using Randu's values for *a*, *c*, and *m*, make a list of 1,000 pseudo-random numbers (between 0 and 1) starting with a non-zero seed of your choosing.
6. Make a scatter plot of x_{i+1} versus x_i from your list.
7. Answer the following in a text cell: Do the points appear random and uniformly cover the plane? Do you see any evidence of banding?

Let's examine the correlation of the triplet (x_i, x_{i+1}, x_{i+2}) by plotting those points in 3D.

8. Set up a 3D plot by starting your cell with `plt.subplot(projection='3d')` and then make a 3D scatter plot with `plt.scatter(xpts, ypts, zpts)`, where you choose `xpts`, `ypts`, and `zpts` to correspond to the vector (x_i, x_{i+1}, x_{i+2}).

You should now have an interactive 3D plot that you can rotate using your mouse.

9. Slowly rotate the 3D plot. Keep rotating until you see something that looks rather striking.
10. What do you see? Do the points look randomly scattered in 3D? Put your answer in a text box.

Suffice to say 'Randu' was abandoned after this discovery, though numerous publications had been generated using it as a random number generator. Fortunately, the random number generators in numpy don't have such problems (at least not that have been found yet).

We won't need interactive plots anymore, so let's turn them off.

11. Turn off interactive mode by running `%matplotlib inline` in a cell.

Task 2: Generating uniform random numbers and timing calculations in Python

Python's default random number generator is called the Mersenne twister, whose period is 2^{19937}-1.

You can generate uniform random numbers between [0,1) using `np.random.random()`. Let's time how long it takes to run a loop that generates 10,000 random numbers between [0,1) and using `np.random.random()`. One way to time things in python is to use the timeit module. Here's some code for how to time a code block:

```
from timeit import default_timer as timer

start = timer()
# ... DO SOME STUFF
end = timer()
print(end - start)
```

1. Using the python timing functions, time how long it takes to run a loop that creates a list of 10,000 random numbers by calling `np.random.random()` 10,000 times.
2. Look up the help on `np.random.random()` using `?np.random.random` and learn how to call it once to make an array of 10,000 random numbers. Use the timing functions to determine how long this takes.
3. By what factor is this faster than what you did in (1) above? Put your answer in a text cell.

This little exercise shows how Python can be really slow. Because it is an interpreted language, calculations slow down when loops are involved. Functions like `random` (and all the other scientific modules that we will use) are compiled and therefore execute extremely fast. Where possible, try to use the built-in functions to generate lists/arrays and avoid creating these with loops.

4. Learn how to histogram data and create a histogram of the values you generated in (2) (or (3)). Try using different numbers of bins.
5. What happens to the histogram if you start using too many bins? How does the histogram of values compare to what you would expect for this distribution?
6. If you wanted to keep your relative bin-to-bin fluctuations <= 5%, what's the maximum number of bins you should use? (Think about counting error from one of your lab courses.) Put your answer in a text cell, and make a histogram using this number of bins.

Task 3: Probability Distributions and Histogramming

One use of random number generators in simulations is to generate sequences of stochastic processes. In some cases, the stochastic process may lead to an analytic result for the statistical distribution for the random variable. In the following exercise, you are going to simulate a stochastic process and show that a histogram of a sample of randomly generated data matches the known statistical distribution for the process. You will then learn how to use Python to generate random numbers that are sampled from a variety of known distributions.

(NOTE: To see all the various methods for generating random numbers from both discrete and continuous distributions, see https://docs.python.org/3.13/library/random.html or the NumPy versions, https://numpy.org/doc/2.3/reference/random/index.html.)

Radioactive decay

In radioactive decay, an unstable nucleus has a probability *r* per unit time for turning into a different isotope. If one watches such a first-order decay process (a death process) and records how long it took for each decay to happen, one has a list of decay times.

1. If $P_{\text{not yet}}(t)$ is the *probability* that a decay has not happened up to time *t*, and $r\,\Delta t$ is the probability that a decay happens in Δt, show in a few steps that the *probability density* for the decay occurring at *t* is an exponential distribution, $p_{\text{now}}(t) = r\exp(-rt)$. Write up your steps and proof in a text cell and format your equations. (Hint: Write down an equation for $P_{\text{not yet}}(t+\Delta t)$ and work out the differential equation that $P_{\text{not yet}}(t)$ satisfies. Then relate $p_{\text{now}}(t)$ to $P_{\text{not yet}}(t)$ and solve for $p_{\text{now}}(t)$.)
2. Generate 10,000 decay times by simulating a first-order decay process. Choose a value of r and Δt such that $r\,\Delta t < 1$. To generate a single decay time, start at $t = 0$ and keep generating a uniform random number and adding Δt to t until you get a random number that is less than $r\,\Delta t$. Your value of t is then your random decay time. Use `timer` to time your calculation.
3. Histogram your decay times. Figure out how to use the histogram method to create a normalized probability density of decay times. Plot the exponential distribution $P_{\text{now}}(t)$ on top of your histogram using your chosen value of r. Do they agree? Label your axes and put in a legend.
4. Try redoing (2) and (3) with the same value of *r*, but a different value of Δt; and still with the condition of $r\,\Delta t < 1$.
5. Now use the NumPy function `np.random.exponential` to generate an array of 10,000 decay times using the same value for r, and histogram the times. Use `timer` to time your calculation. How does the time compare to what you did in (2) ?
6. Histogram these decay times. As before, plot the exponential distribution on top of your histogram. Does it match up with the histogram of your stochastic process?

If instead we watch the radioactive decay process for a fixed amount of time *T*, then the number of decays follows a Poisson distribution with an expected number $\mu = rT$. The python package `scipy.stats` has a number of statistical distribution functions, including the Poisson distribution, that you can generate random samples from.

7. Import the `scipy.stats` package by entering `import scipy.stats as stats`
8. Learn how to use the Poisson distribution from this package and use it to generate a single random sample using an expected number $\mu = rT$ of counts, where *r* and *T* are of your choosing.
9. Generate 1,000 random samples from the Poisson distribution and histogram the result. What are the permissible values of a Poisson random variable? Make sure to set up your histogram bins to be appropriate.
10. Calculate the mean and variance of your samples using `np.mean` and `np.var`. How do these compare to their expected values given your choice for μ?
11. Use `poisson.pmf` to plot the exact distribution on top of your histogram. How well do they match?
12. Make a separate figure plotting several different Poisson distributions using `poisson.pmf`, from small expected number μ of counts to much larger values. As you increase μ, what does the shape of the Poisson distribution begin to look like?

Thus python provides a large library of statistical distributions that can be used when you need to sample from a known stochastic process, either discrete or continuous.

Task 4: Random Walks

A stochastic process known as a random walk captures the behaviour of diverse systems, from the motion of a diffusing particle, to the transfer of heat, to the shapes adopted by a disordered polymer. Consider a 1D walk of *N* steps, each of step size *a*. At each step, for a symmetric random walk, there is a 50/50 chance of going left or right. If the random walk starts at $x(0) = 0$, at the *n*th step it is at position $x(n)$. After *N* steps have been taken, the random walk's end-to-end displacement is $r(N) = x(N) - x(0)$. In the following exercise, we are going to simulate random walks and look at the statistics of the end-to-end displacement.

1. Simulate a 1D random walk (i.e., steps are only to the left and right) of length *N* steps, each of step size $a = 1$. Generate 10,000 such walks for several *N* (say *N* = 10, 20, 30, 40, and 50). For each walk, record the end-to-end displacement $r(N)$. As you simulate, think about how to store your results as a single array of data, where you have as many rows as values of *N* and as many columns as the number of simulated random walks.
2. For each of your *N,* calculate the mean end-to-end displacement $\langle r(N) \rangle$. Print the resulting values.
3. What do your simulations suggest is the mean of the end-to-end displacement of a 1D symmetric random walk? Write your answer in a text cell.
4. Calculate the mean squared end-to-end distance (MSD), $< r(N)^2 >$. Make a scatter plot of the MSD vs. *N*. What sort of curve does the MSD appear to be, as a function of *N*? For comparison, if you had taken *N* steps all in the same direction (i.e., not a random walk, but a directed walk), what sort of curve would the MSD be as a function of *N*?
5. Do the math: If each step has a displacement $\mathrm{d}\vec{x_i} = \pm a$, then $r(N) = \sum_{i=1}^{N} \mathrm{d}\vec{x_i}$. What is $< r(N) > = \sum_{i=1}^{N} < \mathrm{d}\vec{x_i} >$? What is $< \mathrm{r}^2(N) > = \sum_i \sum_j < \mathrm{d}\vec{x_i} \cdot \mathrm{d}\vec{x_j} >$? (Hint: if the steps are completely uncorrelated what is the average value of $< \mathrm{d}\vec{x_i} \cdot \mathrm{d}\vec{x_j} >$ if $i \neq j$? What about for $i = j$?) Write your answers in a text cell.

6. From your data above, choose 3 different values of increasing *N* and make histograms of $r(N)$ as 3 subplots. Use the same range for each and figure out how many bins you need to use so that the binning is the same for each. As above, normalize your histograms so that they are densities. Do these distributions look like any distribution that you know? How do the distributions change as you increase the number of steps? Can you think about how the quantities you calculated in (2) and (3) relate to the parameters of these distributions? Write your answer in a text cell.
7. A random walk is equivalent to flipping a coin (potentially an unfair one) *N* times and recording the result. Read up (i.e., on Wikipedia) about random walks and their relation to the binomial and Gaussian distributions. For each of your distributions in (5) above, plot the appropriate Gaussian distribution with the appropriate choices for mean μ and variance σ^2. Do they match up with your histograms?
8. Simulate an isotropic, two-dimensional random walk on a square lattice. At each step, the walker makes a step of size *a* in each of the *x* and *y* directions (i.e., there is movement in both the *x* and *y* directions), with equal 50-50 probabilities of left-right and up-down. Use the same step size `a=1` as you did in 1D for both the *x* and *y* directions. Plot the mean squared end-to-end distance versus the number of steps *N*. How does the 2D MSD compare to the 1D case? Has anything changed?

Task 5 – Wrap up

I would appreciate any feedback you might have on the Activity guide. Using text cells, please provide answers to the following questions:

1. Were there any questions that were confusing or worded in a way that made them difficult to understand? If so, please reference the Task and question number, so that I can improve it.
2. Please provide any other feedback that you think might have helped to improve your learning experience with this Activity Guide.
3. Download your notebook to the local computer and email it to yourself for safe keeping (i.e., make a backup). Don't assume that your files will be there the next time you log in to syzygy (but let's hope so).