Chapter 9

Lab 9: Monte Carlo Simulation

9.1 Introduction

In this lab, we will introduce the Monte Carlo method, an approach to solving a wide range of problems by repeatedly drawing random numbers. You will use histograms to compare randomly thrown values to their corresponding probability distributions functions. You will measure π and compute one integral using a Monte Carlo method.

9.2 Preparation

In this exercise, we will make use of slicing and boolean masks, which are covered in sections 1.4.1.5 and 1.4.1.7 of the Scientific Python lecture notes. Make sure you are comfortable enough with these concepts to understand the following exercises:

△ **Jupyter Notebook Exercise 9.1:** Run the following snippet, and make sure you understand the output:

```
x = np.arange(10)
print("x       = ", x)
# all but the first element:
print("x[1:]   = ", x[1:])
# all but the last element:
print("x[:-1]  = ", x[:-1])
# the first 4 elements:
print("x[:4]   = ", x[:4])
# elements from index 2 to 6:
print("x[2:7]  = ", x[2:7])
# elements in reverse order:
print("x[::-1] = ", x[::-1])
```

△ **Jupyter Notebook Exercise 9.2:** Run the following snippet,

```
x = np.arange(5)
print("x =          ", x)
print("midpoints:   ", (x[1:]+x[:-1])/2)
```

which demonstrates a trick we will be using later, for calculating the midpoints between each value in x. Note that the first array has five entries, the second has four.

△ **Jupyter Notebook Exercise 9.3:** Run the following snippet, and make sure you understand the output:

```
x = np.array([1,2,5,1,5,1,8,2])
print("x        = ", x)
mask = x > 2
print("mask     = ", mask)
print("x[mask] = ", x[mask])
```

△ **Jupyter Notebook Exercise 9.4:** Run the following snippet,

```
x = np.array([7,2,10,4,2,9,1,3])
mask = x>3
np.sum(mask)
```

which demonstrates a trick we will using later, for counting the number of entries in an array which satisfy a particular condition.

9.3 Generating random numbers

The Monte Carlo method relies on the generation of random numbers, so we will start there. The numbers we generate using computers are actually "pseudorandom" numbers, because they are deterministically obtained from an algorithm. However, the algorithm is choosen so that the numbers appear random for practical purposes. This is no small concern. Much of the computational work in the early 1970's had to be redone because of the widespread use of a deeply flawed pseudorandom number generator called RANDU.

In this section, you will generate a pseudorandom number sequence using the linear congruential method. This sequence is determined iteratively from the simple relationship:

$$I_{n+1} = (a\, I_n + c) \bmod M$$

Recall that $x \bmod y$ (coded as `x % y` in Python) is the remainder after integer division (`x//y` in Python). Each I_n is called a seed, and the initial seed I_0 must be provided to start the sequence. Notice that the seeds are all integers in the range from 0 to $(M-1)$. If we wish to convert these seeds into a random variable x in the range from 0 to L, we simply use $x_n = L * I_n/M$. As long as M is much larger than L, x is approximately continuous.

The algorithm works because the product $a * I_n$ is generally many times larger than M, so the remainder is effectively a uniform random number. The effectiveness of this algorithm is highly dependend on the choice of a,c, and M. Choose poorly and you get RANDU. Choose wisely and you get the highly regarded algorithm of Park and Miller. We will do the latter and use $a = 7^5$, $c = 0$, and $M = 2^{31} - 1$.

△ **Jupyter Notebook Exercise 9.5:** Define a function:

```
def parkmiller(i):
    # your code here
    return i # updated value
```

which, given a seed i, returns the next seed in the Park and Miller algorithm. Make sure a, c, and M are integers or the algorithm will not work properly! Check your code by testing that for a initial seed of one, the generator returns a **seed** of 1043618065 after 10000 calls.

△ **Jupyter Notebook Exercise 9.6:** Use your `parkmiller(i)` function to fill a numpy array `xarr` with 5 randomly thrown x values in the range $[0, 1]$. Check you code with:

```
print(np.around(xarr,2))
```

Now that we have seen how randomly number are generated, we will use the standard numpy tool to produce array of randomly drawn values as needed for us:

△ **Jupyter Notebook Exercise 9.7:** Use the `np.random.uniform` to create an array `xarr` of five randomly thrown x values in the range $[0, 1]$. Check you code with:

```
print(np.around(xarr,2))
```

9.4 Probability Density Functions and Histograms

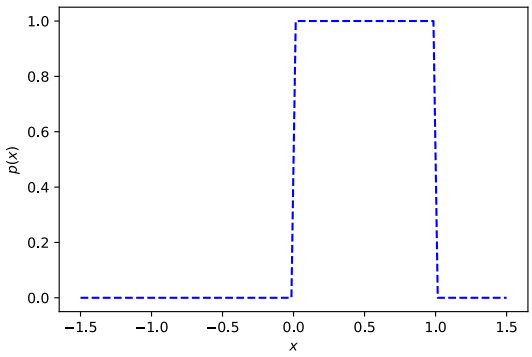


Figure 9.1: The PDF for the process of throwing a random variable uniformly in the region $[0, 1]$.

When I generate ten random numbers from the Park and Miller algorithm, I get the following ten values:

```
[0.6448101  0.32334154 0.40125851 0.95175061 0.07252577 0.94062374
 0.06316977 0.69433175 0.63370098 0.61235022]
```

We say that these ten numbers are thrown (like dice) or drawn (like cards?) uniformly in the interval from $[0, 1]$. How can we describe this process of thowing random numbers in terms of probability? The probability of drawing a particular number, like 0.06316977 is extremely small. If the computer

had unlimited precision, the probability of drawing any particular number would drop all the way to zero. Probability by itself doesn't seem to be very useful here. The solution is to recognize that it is the probability of throwing a number in some region $[a, b]$ which is non-zero. Instead of a probality, we describe this process by a probability density function (PDF), in this case:

$$p(x) = \begin{cases} 1 & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{9.1}$$

This PDF is illustrated in Fig. 9.1. To find the probability P that we would throw x in some interval $[a, b]$, we integrate the probability density function:

$$P = \int_a^b p(x)\, dx$$

For example, the probability that we throw a number less than one half is:

$$P = \int_{-\infty}^{\frac{1}{2}} p(x)\, dx = \int_0^{\frac{1}{2}} 1\, dx = \frac{1}{2}$$

As are all PDFs, this one is normalized to a total probability of one:

$$\int_{-\infty}^{+\infty} p(x)\, dx = \int_0^1 1\, dx = 1$$

△ **Jupyter Notebook Exercise 9.8:** Define a python function:

```
def pdf(x):
    # your code here
    return p
```

which implements the PDF from Equation 9.1. Use it to create x and y values for plotting as:

```
xf = np.linspace(-1.5,1.5,100)
yf = pdf(xf)
```

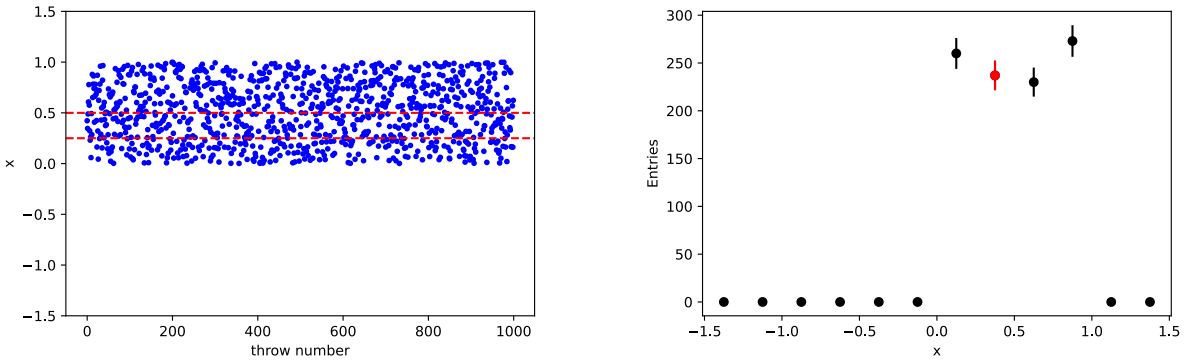
Reproduce the plot in Fig. 9.1. Hint: for this to work, you have to make sure the function `pdf(x)` can handle properly the case that x is a numpy array, and return a numpy array of the same size.

But how can we be verify that our numbers drawn from the Park and Miller algorithm:

```
[0.6448101  0.32334154 0.40125851 0.95175061 0.07252577 0.94062374
 0.06316977 0.69433175 0.63370098 0.61235022]
```

are actually being drawn from the PDF in Fig. 9.1? The tool of choice for seeing the "shape" of a list of values is the histogram, and the process of building one is illustruated in Fig. 9.2. First, let's draw 1000 random values. One way to visualize these values is shown in Fig. 9.2a, which simply plots each value above the throw number (from 0 to 1000). To build a histogram, we devide the x range into small regions, called *bins*. For example, we have a bin from 0.25 to 0.5, as indicated by the dashed red lines. The number of blue points contained within that range is 237. In Fig. 9.2b, we plot the count as the red point. The y-value of the point is the count 237. The x-value is the middle of the bin:

$$(0.50 + 0.25)/2 = 0.375$$



(a) (b)

Figure 9.2: The 1000 measurements of variable x in (a) are used to produce the histogram in (b). The red data point in (b) is the count of the number of entries in range indicated by the red dashed lines in (a).

We continue this process for as many bins as we wish to plot. The result is the entire set of points in Fig. 9.2b which we call a histogram. A histogram is a set of bins, with a count corresponding to each bin.

Notice that each data point in our histogram has an error bar: the verticle line passing through each point in the histogram. A fundamental result from statistics is that the best estimate for the statistical uncertainty on a count N of independent occurances is simply $\sqrt{N}$. The error bars in Fig. 9.2b have been taken as the square root of the count in each bin.

Fortunately, the process of calculating a histogram from an array of values is handled by the python function `np.histogram`, which you will use in the following exercise:

△ **Jupyter Notebook Exercise 9.9:** Run the following code snippet:

```
NTOT = 1000  # total number of events thrown
NBIN = 12    # number of bins in histogram
XMIN = -1.5  # maximum X value
XMAX = 1.5   # minimum X value
# throw NTOT random values uniformly in [0,1]:
xarr = np.random.uniform(size=NTOT)
#create a histogram from the random values in xarr:
# hx:   the histogram counts (length NBIN)
# edges: the bin edges (length NBIN+1)
hx,edges = np.histogram(xarr,bins=NBIN,range=(XMIN,XMAX))
# calculate the center of each bin, for plotting:
cbins = (edges[1:]+edges[:-1])/2
# calculate the error in each bin as the square root of the count
err    = hx**0.5
# plot the histogram, including errorbars, using the errorbar function:
plt.errorbar(cbins,hx,yerr=err,fmt="ko",label="Histogram")
# add the labels
plt.xlabel("x")
plt.ylabel("Entries")
```

to construct a histogram like that of Fig. 9.2b.

There are some lines of particular importance in the code snippet, which you will need to understand to succeed in this lab. This line:

```
hx,edges = np.histogram(xarr,bins=NBIN,range=(XMIN,XMAX))
```

actually creates the histogram for the array `xarr`. It creates `NBIN=12` bins in range from `XMIN=-1.5` to `XMAX=1.5`. The count for each bin is contained in the array `hx` which has length `NBIN`. The edges of the bins are contained in the array `bins` which has length `NBIN+1`. Because they are different lengths, you cannot simply plot `hx` versus `bins`. Instead, we calculate the bin centers with the line:

```
cbins = (edges[1:]+edges[:-1])/2
```

which uses slicing to produce an array of length `NBIN` containing the bin centers. We calculate the statistical error for each point in the histogram as the square root of the count:

```
err   = hx**0.5
```

The line:

```
plt.errorbar(cbins,hx,yerr=err,fmt="ko",label="generated")
```

plots the histogram with errorbars. Sometimes students are confused by the name of the `plt.errorbar` function: it plots both the histogram and the errorbars!

△ **Jupyter Notebook Exercise 9.10:** Modify the code snippet from the previous example to create and draw a histogram from 1000 random values thrown in the range $[0, 1]$ using your Park and Miller algorithm.

Your histogram in the previous exercise should look much like that of Fig. 9.2b. We can also see that it has the same shape as the PDF in Fig. 9.1. But the histogram has a maximum value of around 270, whereas the PDF has a maximum value of 1. This is because the histogram presents a count and the PDF presents a probability density. For $N_{\rm tot}$ total events thrown, we can use the PDF $p(x)$ to predict the number of events N_{ab} in a bin with edges a and b as:

$$N_{ab} = N_{\rm tot} \int_a^b p(x)\, dx$$

where $p(x)$ is the PDF, $N_{\rm tot}$ are the number of throws. From the mean value theorem we can find a particular x^* in the range $[a, b]$ such that

$$\int_a^b p(x)\, dx = (b-a)\, p(x^*)$$

and so we have:

$$N_{ab} = N_{\rm tot}\, \Delta x \cdot p(x^*)$$

where $\Delta x = b - a$ is the bin size. As a practical matter, instead of calculating N_{ab} for each bin, we simply plot the PDF scaled as:

$$N(x) = N_{\rm tot}\, \Delta x \cdot p(x)$$

which allows us to directly compare a PDF to the histogram. In our case, the scale factor is:

$$N_{\rm tot}\, \Delta x = 1000 * (0.50 - 0.25) = 250.$$

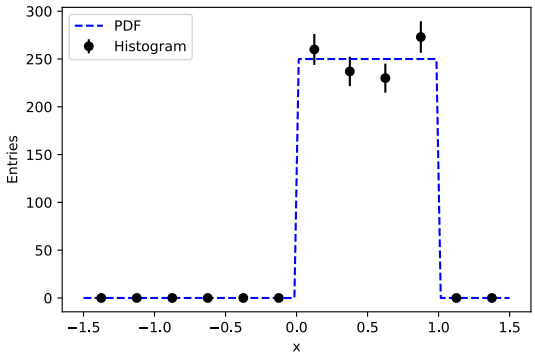


Figure 9.3: Direct comparison of a histogram containing 1000 events thrown uniformly in the range $[0, 1]$ with the corresponding PDF, scaled appropriately.

A comparison of the histogram with the PDF scaled by this factor is shown in Fig. 9.3.

△ **Jupyter Notebook Exercise 9.11:** Add a plot of the scaled PDF to your histogram, to produce a figure like that of Fig. 9.3

9.5 Calculating the value of π

You can be the life of your next party by showing off how to determine the constant π by throwing toothpicks! The procedure is simple: you cut a piece of paper to a width of four toothpicks, then draw two vertical lines separated by the width of two tooth picks. Take turns tossing toothpicks, as in Fig. 9.4.

From the geometry of the setup, it can be shown that the probability that a toothpick which is entirely on the paper also crosses a line is given by $1/\pi$. Therefore, one can measure π by counting the total number of toothpicks that landed entirely on the page and dividing by the number of those toothpicks that crossed a line. This is, in essence, the Monte Carlo method.

An easier Monte Carlo method to implement computationally is shown in Fig. 9.5 which was generated with following code:

```
N = 1000 # number of random values to throw
# throw N x and y random variables uniform in [0,1]
x = np.random.uniform(size = N)
y = np.random.uniform(size = N)
# determine which (x,y) position or inside/outside the unit circle:
rsq = x**2 + y**2
inside  = rsq<=1
outside = np.logical_not(inside)
# set aspect ratio to 1 so unit circle looks like a circle.
plt.axes().set_aspect('equal')
# plot inside as blue dots and outside as red dots
plt.plot(x[inside],y[inside],"b.",label="inside")
plt.plot(x[outside],y[outside],"r.",label="outside")
# plot the unit circle:
xfin = np.linspace(0,1,100)
```

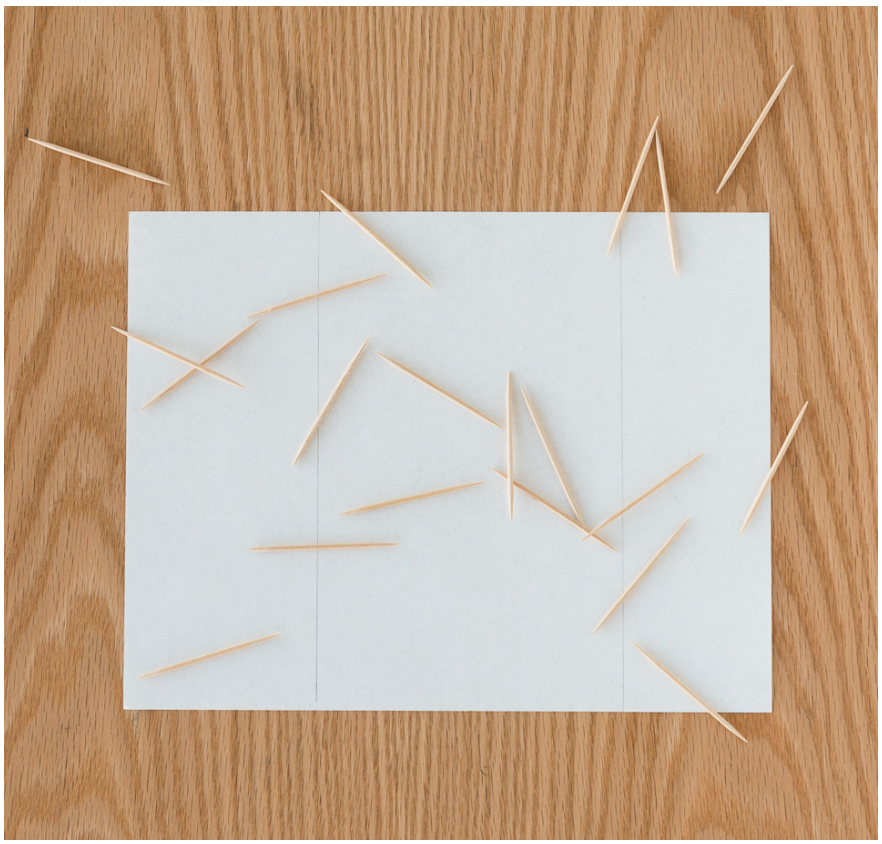


Figure 9.4: Determining π by throwing toothpicks.

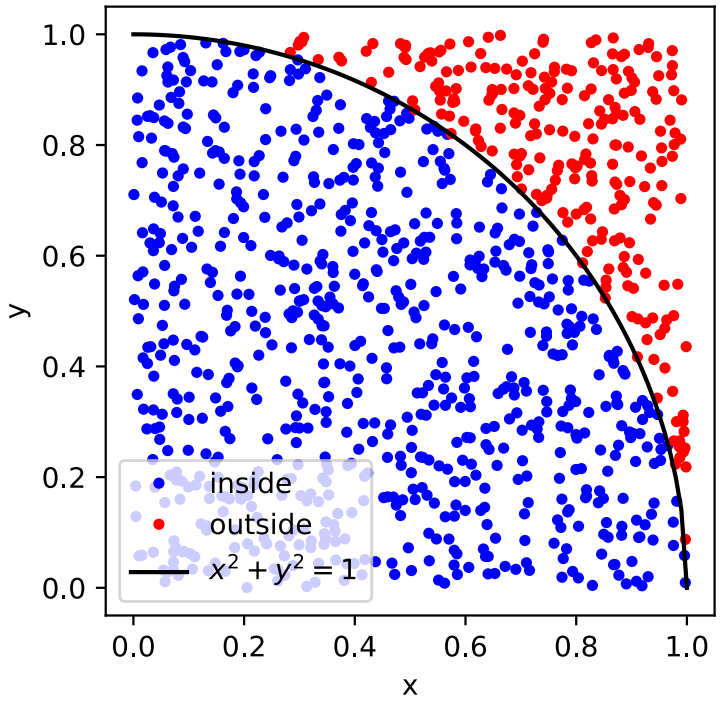


Figure 9.5: Monte Carlo Determination π .

```
yfin = sqrt(1-xfin**2)
plt.plot(xfin,yfin,"k-",label="$x^2+y^2=1$")
# add labels and legends:
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc=3)
```

The idea is to throw points uniformly in the unit square of area 1. Much like in the toothpick example, the value of π can be determined by counting the number of generated points which also landed within the unit circle. Make sure you understand the example code, particular how masks are used to draw the blue and red dots.

△ **Jupyter Notebook Exercise 9.12:** Run the example code. Then, count the number of points inside the unit circle using:

```
n_inside = np.sum(inside)
```

Estimate π using the Monte Carlo method. Hint: based on area, what fraction of total events do you expect to find within the unit circle?

△ **Jupyter Notebook Exercise 9.13:** This is an example of a binomial process, and the statistical uncertainty on your measured value of π works out to be:

$$\sigma_\pi = \sqrt{\frac{\pi\,(4-\pi)}{n}}$$

where n is the number of generated events. Does your measured value of π agree with the known value within your statistical uncertainty?

9.6 Monte Carlo integration

The Monte Carlo method can also be used to numerically integrate a function. Monte Carlo integration methods generally only outperform deterministic methods when the number of dimensions is large, but we can illustrate the method most easily in one dimension. In this section, you'll use the Monte Carlo method to perform the integral:

$$\int_0^\pi \sin^2\theta\, d\theta$$

To do so, you should make a copy of your solution from the previous section and modify it in the following manner:

- Instead of thowing x in $[0, 1]$, throw θ in $[0, \pi]$. This means the area of the rectangle A is now π instead of 1.
- Count the number of throws that land below the integral $y < \sin^2\theta$.
- Determine the area under the curve as the fraction of the throws under the curve times the total area of the rectangle A.
- The statistical uncertainty in this case is $\pi/(2\sqrt{n})$ where n is the number of generated events.

△ **Jupyter Notebook Exercise 9.14:** Use the Monte Carlo method to calculate the integral:

$$\int_0^\pi \sin^2\theta \, d\theta$$

Make a plot similar to that of Fig. 9.5 showing the thows above the curve in red and below the curve in blue. Calculate the integral and statistical uncertainty and compare it to the value you obtain analytically.