Introducing Python Programming Using Monte Carlo

Walter Reisner

January 3, 2023

Professor: Walter Reisner[1]
Teaching assistants: Samin Majidi[2], Syed Nayyer Raza [3], Laurenz Kremeyer[4] and Syed Hassan[5]
Technician: Robert Turner[6]

1 Introduction

Monte Carlo, named after a famous Casino in Monaco, is a computational approach that uses random numbers to perform calculations and simulate physical systems/experimental outcomes. Monte Carlo approaches are widespread in physics, ranging from particle physics (e.g. simulation of particle scattering/interaction with materials), condensed matter and statistical physics (simulating systems in contact with heat baths, e.g. Ising models of magnets, polymers, fluids, Brownian motion), biophysics (noise in biochemical networks and gene expression) to recent fields like econo and socio-physics (physics of stock markets and politics). One would be hard pressed to find a field of physics that does not use Monte Carlo approaches in one way or another. From the point of view of experiments, Monte Carlo approaches are a natural way to understand how noise affects measurements. If I have some theoretical model, I can use Monte Carlo to simulate how experimental noise will alter my measurement, and make informed estimates in advance of how much sampling I have to perform to obtain a result with a certain degree of precision. Particle physics collaborations, for example, run sophisticated Monte Carlo to test whether a given detector and can measure a certain effect (e.g like a signal of neutrino oscillation). Monte Carlo methods are really useful when you are learning subtle statistical concepts, because they provide a way to make these notions concrete and accessible. In my own work (in area of single-molecule nanofluidics), I use Monte Carlo all the time, both to simulate experiments and as a theoretical approach in its own right (in collaboration with simulation experts).

The objective of this lab is to teach you the basics of Python script writing and Monte Carlo techniques. Specifically, you will learn to write Monte Carlo scripts in Python and to use this code to address some subtle statistical questions. The first part of the lab is a tutorial that will teach you how to write scripts in Python for generating random numbers. The second part (section 6) gives exercises that you will need to write up in your report. Note that, while I will occasionally ask questions in the tutorial section, only questions asked in the exercise section need to be explicitly addressed in your group's report.

I *strongly* suggest that you not just cut and paste the example programs from the tutorial directly into the Python editor (Spyder), but copy them line by line and test the commands in the Python console first so you can learn exactly what they do. While this will slow you down a little, this will help you learn the commands so you can more easily write your own programs.

2 How to Access Spyder

In this class we will be performing scientific programming using IPython in the Spyder environment (we use Python version 3.9 in this class). *It is **important** to know that there are currently two versions of Pythons in wide use: Python 2 and Python*

[1] reisner@physics.mcgill.ca
[2] samin.majidi@mail.mcgill.ca
[3] nayyer.raza@mail.mcgill.ca
[4] laurenz.kremeyer@mail.mcgill.ca
[5] syed.hassan3@mail.mcgill.ca
[6] robert.turner@mcgill.ca

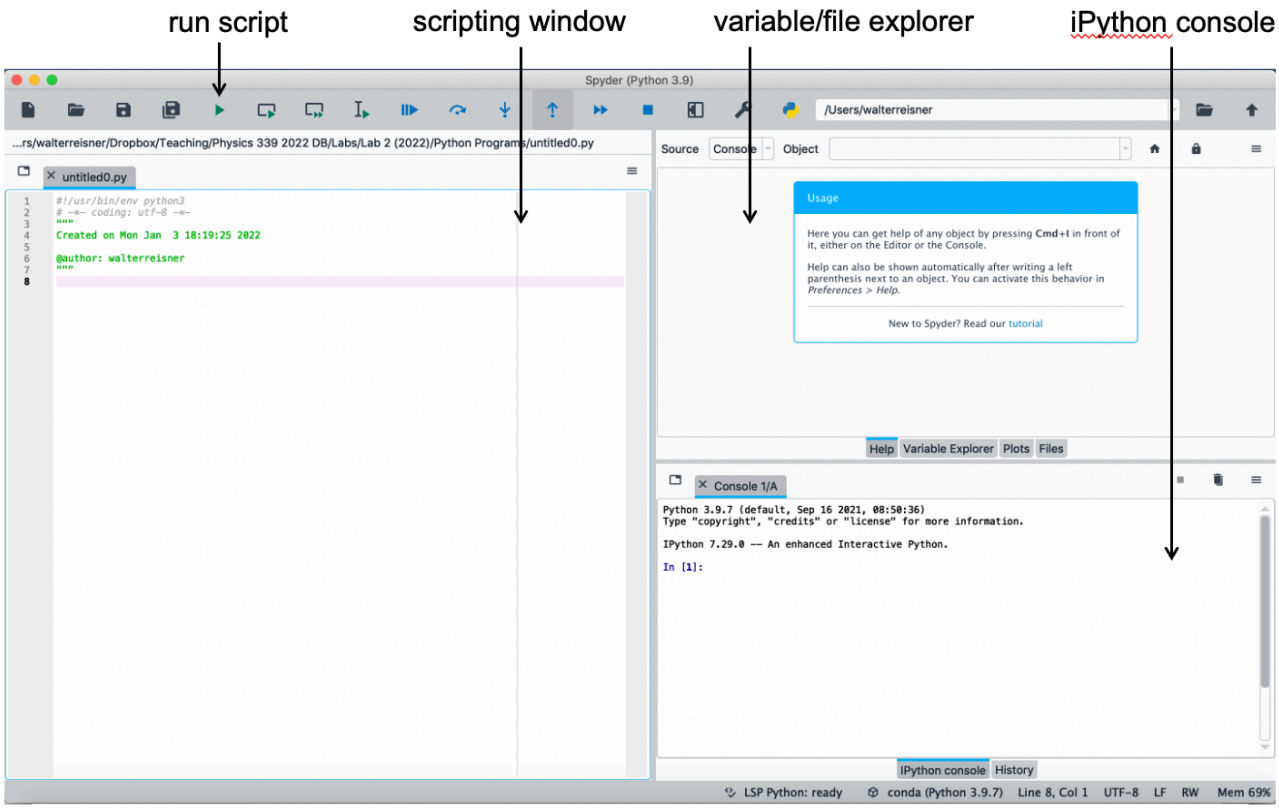

Figure 1: The Spyder default layout. The main window is for writing scripts. The variable/file explorer allows you to see your computer directory or lists of variables generated by your script. The iPython console allows you to run individual Python commands. Your scripts can also output results to the console. The green arrow button allows you to run your scripts. You can adjust this display using the options in the *View* menu bar. Note that *View-Window layouts -Spyder default layout* will return you automatically to the default layout.

3. For this lab we will be using Python 3. ***Be sure that you are compiling your code as Python 3 in Spyder, and that the documentation you consult is for Python 3.*** As discussed, you can use any programming language you desire (and in fact Matlab is provided). Note, however, that full support will only be provided to Python (limited support will be provided to Matlab upon request, on a one-on-one basis, excluding serial applications).

Spyder is shareware and can be downloaded as part of the Anaconda installation onto your personal computer running your operating system of choice. Anaconda is a kind of overview package that allows you access common Python environments (including Spyder and Jupyter). To download Anaconda, go to: *https://www.anaconda.com/products/individual* and install the most recent version of Anaconda with python 3.9 appropriate for your operating system. Open Spyder–the icon looks like a spider web!–and you should see a window that looks like Fig. 1.

Before you start, it is a good idea to enable the following preferences. First, go to the *python* menu and open *Preferences*. Go to the *Run* section, and ensure that the *Default working directory* is set to "The directory of the file being executed." This will ensure that all output files are placed in the directory where your script is saved and scripts can access files saved in this directory. In order to display figures in a separate pop-up window and not the console, in *Preferences* go to *IPython console*, *graphics* tab, and make sure *Backend* is set to "Automatic".[7] Also, you can change the console appearance (e.g. from light to dark or vice versa) in the Appearance menu. It is necessary to restart the kernel for this setting to take effect. Go to Consoles menu → Restart kernel. Your graph will open in a separate window, perhaps behind the main Spyder window.

[7]Note to mac users: when I run Spyder on my OSX 10.10.5 (Yosemite) laptop I set this to OS X.

3 Writing a Spyder Script to Generate and Display Uniformly Distributed Random Numbers

When you look at the Python editor window you will see a new script displayed. You can make a new script anytime by going to file menu and hitting "new file". At the top of your script you will see some text beginning with the symbol "#" which is greyed out and some green text between three quotation marks. The green text gives you date the file was created. This text is written in *comment* mode. Comments indicate text that is not to be executed when you run the program. The purpose of comments is to help you document your code, helping others understand your program and reminding you for later reference what your program as a whole or individual commands do. (I never sufficiently comment my code and always regret it). There are two types of comments: single line comments that begin with the # symbol (coloured grey) and multi-line comments that are inserted between three quotation marks (coloured green). Why not add the comment: "Example program to test random number generation in Python"? (or whatever). At this point, you might as well save the program (I suggest you call it "uniform monte carlo"). I also suggest you make a dedicated directory in the documents folder for each lab.

In Python many key functions are accessed through libraries, which must be separately imported. Two key libraries we will be using for many of our programs are `numpy` (numerical python, support for arrays and array operations) and `matplotlib.pyplot` (graphing). In this exercise, we will also be using the `random` library, providing support for pseudo-random number generation. You will learn later how to make your own function libraries and import them. For now let us import `numpy`, `matplotlib.pyplot` and `random`. You do this via the following commands that will begin your program:

```
import matplotlib.pyplot as plt
import random as r
import numpy as np
```

Note the command "as." This allows us to refer to the libraries in an abbreviated way (plt for pyplot, r for random, np for numpy). You will soon see that this is quite time-saving (you can use any abbreviation you like but it is best to pick specific abbreviations and be consistent).

To see what functions from the `random` library do, let us test the function `random()` in the IPython console. Type `import random as r` and hit enter in the console (not editor). Then type `r.random()` in the console and hit enter (repeat several times, you always need to type enter to execute a command in the console). You should see a different floating point number between 0 and 1 each time. The `random()` function gives you a uniformly distributed pseudo-random number in the open range $[0.0, 1.0)$. Note the syntax of this command. When I run a command from a given library in Python I always need to prepend the library name: a command takes the form `library name.function name`. If we did not choose to abbreviate the name of the random library name by "r" we would have to type `random.random()`, which is quite cumbersome. We will not get into the details here of how the pseudo random numbers are generated,[8] but the basic point is that the algorithm generates a sequence of random numbers given a certain seed value (the same seed will generate the same sequence of random numbers[9]). The values behave statistically very close to "true" random numbers, such as we would obtain from a random process in nature (like radioactive decay), but not exactly (hence the prefix, "pseudo").

Now let us write a script to generate and display some random numbers. In particular, we will find N_{samp} random numbers, plot and then histogram them. In addition, our program will find and display the mean and standard deviation of the random values. First, write the following:

```
Nsamp=10

values=np.zeros(Nsamp)
trials=list(range(Nsamp))

for i in trials:
    values[i]=r.random()
```

[8] A good start is the scientific computing bible: W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes: The Art of Scientific Computing* , Cambridge University Press (1986)

[9] The random() function uses the current time as the seed if you do not provide one. You can set an explicit seed via seed(). The argument for setting an explicit seed is that your Monte Carlo will yield exactly the same result every time you run it (e.g. the same sequence of random numbers is used).

```
print('mean', np.mean(values))
print('std', np.std(values))
```

Pushing the green arrow at the top of the Spyder window will run this program (Fig. 1): you should see the computed mean and std for 10 samples displayed in the console window. Pushing the green arrow is equivalent to the command `runfile` which will appear in the console with appropriate arguments when the green arrow is pushed.

How does this program work? This code first assigns the variable N_{samp} the value 10 (note that in IPython you don't need to formally declare variables). The next command creates an array *values* containing N_{samp} zeros. We then create an array *trials* which contains integer values running from 0 to $N_{\text{samp}} - 1$. The `for` loop then iterates the variable *i* over the integer values in the array *trials*, running the indented program block below the `for` statement for each value of *i*. The command in the indented block computes a random value and assigns it to the *i*th position in the array *values*. When the `for` loop finishes the array *values* will contain N_{samp} random numbers. If you are new to Python, it is very important to note the critical role of indentation in the program syntax: all indented lines below a loop in Python belong in the loop (i.e. are run over each loop iteration). It is quite possible to completely change how your program runs by forgetting to indent or indenting by accident. The last two commands compute the mean and std of the array *values* (commands `mean(values)` and `std(values)`) and display the results in the console using the `print` command. Also, note that I do not need a semicolon after each command in Python (but I do not need to use semicolons if I wish to combine two commands on the same line).

If you are unsure about some of the specific details, try typing individual commands in the console. Type the command `zeros(10)` *(don't forget to* `import numpy` *first and prepend its alias!)*. Next, type `range(10)` and then `list(range(10))`. The `range` command is technically just instructions to create integer values, i.e. it does not store these elements in memory; the command `list` actually converts these instructions into a real array with each element stored in memory. Create a "test" array by typing `t=[3, 2, 4]`. Then type `t[0]=0`, followed by `t`, which will display the array and indicate what the command `t[0]=0` "does". What happens if you then type `t[2]`? Finally, test the `print` command. What does `print(10)` do? What about `print('test number', 10)`? These exercises are trivial, but note how they illustrate the critical role of the console in Spyder for playing around with individual commands and learning what they do. If you are unsure about what a command does, always implement it in the console on test values or arrays before incorporating it into your script. Mistakes in a script will be harder to diagnose than a mistake in the console. If you do get error messages after running a script, use the `print` command to output variable values before the code fails, this can help you learn what is causing the error.

Finally, let us plot the values in the array *values*. This can be done via the following lines of code:

```
binNumber = 20

f1 = plt.figure()
plt.subplot(1, 2, 1)
plt.plot(trials, values, 'o')
plt.xlabel('trials')
plt.ylabel('values')
plt.subplot(1,2,2)
plt.hist(values, bins=binNumber)
plt.xlabel('value')
plt.ylabel('counts')
plt.tight_layout()
```

Adding these commands at the end of the above script will instruct your program to plot the random values obtained as a function of the trial number and then histogram these values using *binNumber* bins. Play around with increasing N_{samp} and varying the bin number. Is the meaning of a "uniformly distributed random number" clear? Do you understand why the mean and std have the values they do? Note the use of `plot` and `hist` commands from the `matplotlib.pyplot` library. The commands `xlabel` and `ylabel` are useful for labelling the x and y axis. I also like the use of the `subplot` command to put multiple plots in one widow. Note that the first two integer arguments in `subplot` define the array size and the last integer defines the position of the given plot in the array. It is wise to label the figure window (here "f1") in case you wish to have multiple figure windows open during program execution. The command `plt.tight_layout()` separates the plots.

Congratulations, you have written your first script! Naturally, there are many perfectly equivalent ways to write the above. You could use a `while` loop instead of a `for`, or instead of assigning values to an existing array of fixed pre-defined length use `append` to append values one-by-one to an empty array. What if you want to learn more about some of the commands we have used? Try googling "range, python." You will find many sites giving you examples of this command and its different arguments (we have used only the simplest version of the command). You can access a range of standard documentation at the site *https://docs.python.org/3/*, but I find it faster to find information by Googling specific commands.

4 Generating Discrete Random Numbers: If-Statement Programming

While `rand()` generates uniformly distributed numbers, what if we want to generate random numbers that are non-uniformly distributed? There is a simple way to do this for an arbitrary 1-D probability distribution. However, before we tackle this problem, we will describe how to generate *discrete* random variables, a useful trick in its own right and an important intermediate step in understanding the continuous case.

Let us first simulate a "loaded" or "unfair" coin that returns heads with a probability $p_c > 0.5$. To do this, we will generate a uniformly distributed random number r and then say "heads" (0) is returned if $r < p_c$ and "tails" (1) if $r \geq p_c$. To do this we will modify our code with an `if` statement, representing p_c by the variable *probCoin*. We will also add an array (*coinFlips*) to record whether the coin flip yields heads or tails on each iteration. I give the full code below for clarity (note much of it can be copied and pasted from your first code):

```
"""
Monte Carlo to simulate an unfair coin
"""

import matplotlib.pyplot as plt
import random as r
import numpy as np

nsamp = 100
probCoin = 0.8

values = np.zeros(nsamp)
trials = range(nsamp)
coinFlips = np.zeros(nsamp)

for i in trials:
    values[i] = r.random()
    if (values[i] < probCoin):
        coinFlips[i] = 0
    else:
        coinFlips[i] = 1

numTails = sum(coinFlips)

print('coin flips mean', np.mean(coinFlips))
print('coin flips std', np.std(coinFlips))

binNumber = 20
binNumberFlips = 10

f1 = plt.figure
plt.subplot(2,2,1)
plt.plot(trials, values, 'o')
plt.xlabel('trials')
plt.ylabel('values')
```

```
plt.subplot(2,2,2)
plt.hist(values, bins=binNumber)
plt.xlabel('value')
plt.ylabel('counts')

plt.subplot(2,2,3)
plt.plot(trials, coinFlips, 'o')
plt.xlabel('trials')
plt.ylabel('heads (0) or tails (1)')

plt.subplot(2,2,4)
tailsPercent = 100 * (numTails/nsamp)
headsPercent = 100 - tailsPercent
plt.bar([0,1], [headsPercent, tailsPercent], width = 0.8)
plt.xlabel('heads (0) or tails (1)')
plt.ylabel('probability')
```

To understand the code in detail, note the structure of the `if` statement. The comparison `(values[i]<probCoin)` is a logic statement yielding a "true" or "false" Boolean value if the comparison is respectively true or false (what does this Boolean business mean?). Play with the comparison operators $<$ and $>$ on the command line by typing $2 < 4$ and $2 < 1$. While you are at it you might also try playing with $==$ (Boolean equals), $!=$ (Boolean not equal in Python), $<=$ (less than or equal to) and $>=$ (greater than or equal to). The expression in parentheses, `(values[i] < probCoin)`, is evaluated, and if it is found to be "true", the indented command `coinFlips[i] = 0` is executed. If the expression evaluates to "false", the indented code after the `else` clause (`coinFlips[i] = 1`) executes. We then find the total number of tails returned using the `sum()` command, which just sums all the 1's in the array. Finally I plot the percentage of heads/tails returned as a barchart using `bar`. It is instructive to play with the N_{samp} variable and see how this alters the outcome.

One step up in complexity is to simulate a scenario with more than two-outcomes. For example, we can write a Monte-Carlo program to simulate the roll of an "unfair" six-sided die (weighted to bias our results for 6). Let the probabilities of the die returning $1-6$ be denoted p_i with i ranging from 1 to 6. Now we generate one uniformly distributed random number r. If $r < p_1$, then the die-roll returns 1. If $p_1 <= r < (p_1 + p_2)$, then the die-roll returns 2. If $(p_1 + p_2) <= r < (p_1 + p_2 + p_3)$, then the die-roll returns 3. If $(p_1 + p_2 + p_3) <= r < (p_1 + p_2 + p_3 + p_4)$, then the die-roll returns 4. If $(p_1 + p_2 + p_3 + p_4) <= r < (p_1 + p_2 + p_3 + p_4 + p_5)$, then the die-roll returns 5. If $(p_1 + p_2 + p_3 + p_4 + p_5) <= r < 1$, then the die-roll returns 6. This is like having too much to drink, closing your eyes and "throwing a dart" at a unit interval broken up into six sections with the length of each section corresponding to the probability of the dart hitting that section. Naturally, if you are *very* drunk, the chance the dart will land in a given section of the interval should be proportional to the section's length.

Here is a program that implements the above. For sake of concreteness, we will let $p_o = 1/6$ and then parameterize the p_i in the following way: $p_6 = p_o + b$ (b is the bias indicating how unfair our die is) and for $i < 6$ the $p_i = (1 - p_6)/5$ (say outcomes $1-5$ have equal probability).

```
"""
Monte Carlo to simulate an unfair die
"""
import matplotlib.pyplot as plt
import numpy as np
import random as r

Nsamp=10
po=1/6.0
bias=0.05

p6=po+bias
p=(1-p6)/5.0
```

```
probVect=[p, p, p, p, p, p6]
cumVect=np.cumsum(probVect)

values=np.zeros(Nsamp)
trials=list(range(Nsamp))
dieToss=np.zeros(Nsamp)

for i in trials:
    values[i]=r.random()
    if (values[i]<cumVect[0]):
        dieToss[i]=1
    elif (cumVect[0]<=values[i]<cumVect[1]):
        dieToss[i]=2
    elif (cumVect[1]<=values[i]<cumVect[2]):
        dieToss[i]=3
    elif (cumVect[2]<=values[i]<cumVect[3]):
        dieToss[i]=4
    elif (cumVect[3]<=values[i]<cumVect[4]):
        dieToss[i]=5
    elif (cumVect[4]<=values[i]<cumVect[5]):
        dieToss[i]=6

numToss=np.zeros(6)
for i in range(1,7):
    numToss[i-1]=sum(dieToss==i)

numTossPercent=100*(numToss/Nsamp)

f1=plt.figure()

plt.subplot(1, 2, 1)
plt.plot(trials, dieToss, 'o')
plt.xlabel('toss trials')
plt.ylabel('toss value')

plt.subplot(1, 2, 2)
plt.bar(range(1,7), numTossPercent, width=0.8)
plt.xlabel('toss value')
plt.ylabel('probability')

plt.tight_layout()
```

It is fun to run this code for different N_{samp} to see how many tosses is required to reliably detect that the die has been given to you by a con-man.

Here is a detailed explanation of the programming. First, we introduce a vector *probVect* with a length of 6 that holds the probabilities for each of the six outcomes. We use an extension of the `if` syntax (`elif`, abbreviation of "else if") that enables us to add additional comparisons into the `if` statement to take into account the six possible outcomes. Note that the command `cumsum()` provides a very handy way to take into account the added probabilities defining the interval bounds (as usual, to see what `cumsum()` does try it on a test array in the console, say [1, 2, 3, 4]). In order to count the number of times $1-6$ are returned on each throw, we use the slightly tricky (but extremely powerful!) syntax `sum(dieToss==i)`. The `numpy` library has a lot of functionality for processing arrays. Arrays created with `numpy` functions have special properties to enable this functionality, and there is even a command `array()` that converts a regular Python array into

a `numpy` array. In particular, you can thread function operations over the elements of a `numpy` array (that is apply the function operation in parallel to each element of a `numpy` array without the need for a loop). In fact, you can also thread logic (i.e. Boolean) operations. The command `dieToss==1` for example will produce a `numpy` array containing the Boolean value "True" whenever an element in `dieToss` equals 1 and a "False" whenever the element equals any other value (test this in the console!). When we `sum` that array with Boolean values, we just count the number of times "True" is returned.

5 Generating Non-Uniformly Distributed Random Numbers

Now we are ready to simulate a non-uniform continuous random distribution. Think what would happen if instead of six outcomes we wanted some huge (tending to infinite) number of outcomes p_i. We will parameterize each outcome p_i by the function $p(x)$, so that $p_i \equiv p(x_i)\Delta x$ (i.e. each outcome p_i corresponds to some value x lying in a range Δx about x_i). It is evident that in the limit that the number of outcomes goes to infinity $\Delta x \to dx$ and our process is determined by the non-uniform continuous probability density $p(x)$. We will let $p(x)$ be defined on the interval $[x_{\text{min}}, x_{\text{max}}]$ (the limits can go to positive and negative infinity, or they can be finite).

It seems at first that this process would be impossible to Monte-Carlo: how can we write an infinite number of else-if statements? Yet in reality it is not so bad. To see why, let us generate a uniformly distributed random number r. As before we assign r to the ith bin corresponding to the ith outcome along the unit interval:

$$\sum_{j=1}^{i} p_j \leq r < \sum_{j=1}^{i+1} p_j \tag{1}$$

If we take the continuous limit:

$$\sum_{j=1}^{i} p_j \to \int_{x=x_{\text{min}}}^{x} p(x')dx' \quad \text{and} \quad \sum_{j=1}^{i+1} p_j \to \int_{x=x_{\text{min}}}^{x+dx} p(x')dx' \to \int_{x=x_{\text{min}}}^{x} p(x')dx' \tag{2}$$

The point is that in the continuous limit both the upper and lower bounds become *equal*, as the differential dx is infinitesimal, and both bounds tend towards the function

$$c(x) \equiv \int_{x=x_{\text{min}}}^{x} p(x')dx'. \tag{3}$$

We call $c(x)$ the "cumulative probability density." The cumulative probability density gives the probability of getting an outcome $x' \leq x$. Evidently, in the continuous limit, as r is bounded by two quantities that become equal and are themselves equal to $c(x)$, we can say $r = c(x)$. Our Monte-Carlo recipe is thus surprisingly simple: (1) generate r and (2) find the inverse c^{-1} of the cumulative probability to obtain $x = c^{-1}(r)$.

Let us test this procedure with an exponential probability distribution: $p(x) = a\exp(-ax)$ defined on the range $[0, \infty)$ (note $p(x)$ is normalized!). The corresponding cumulative probability density is: $c(x) = 1 - \exp(-ax)$. Inverting we find $x = -(1/a)\log(1-r)$. We can implement this trick with a small modification of our original uniform random number generation program. I also add some additional lines demonstrating how to obtain the cumulative histogram and how to compare our results against the expected probability density function (PDF) and cumulative probability density function (CDF):

```
import matplotlib.pyplot as plt
import numpy as np
import random as r

# good to group parameters at beginning
# of code so you can find/modify them easily

Nsamp = 100
binNumber = 20
cumBinNumber = 100
```

```
a = 1

#These commands define the exponential probability density (PDF)
#and its cumulative distribution function (CDF)

def expModelPDF(a, x):
    x = np.array(x)
    return a*np.exp(-a * x)

def expModelCDF(a, x):
    x = np.array(x)
    return 1 - np.exp(-a * x)

values = np.zeros(Nsamp)
trials = range(Nsamp)

for i in trials:
    ro = r.random()
    values[i] = -(1/a) * np.log(1 - ro)

print('mean', np.mean(values))
print('std', np.std(values))

#Obtain histogram with number of bins equal to binNumber--this is to compare to the PDF
histValues1, binEdges1 = np.histogram(values, binNumber)

#Obtain histogram with number of bins equal to cumbinNumber--this is to compare to the CDF
histValues2, binEdges2 = np.histogram(values, cumBinNumber)
cumHistValues = np.cumsum(histValues2)/Nsamp

#Need to initialize new arrays binCenterHist and binCenterCumHist
binCenterHist = np.zeros(len(binEdges1)-1)
binCenterCumHist = np.zeros(len(binEdges2)-1)

# convert bin edges to bin center position
# note len() gives number of elements in array

for i in range(len(binEdges1) - 1):
    binCenterHist[i] = 0.5 * (binEdges1[i] + binEdges1[i + 1])

for i in range(len(binEdges2) - 1):
    binCenterCumHist[i] = 0.5 * (binEdges2[i] + binEdges2[i + 1])

#find bin widths
#diff commmand simply takes difference between (i+1)th and ith array elements

binWidthHist=np.diff(binEdges1)

#The quantity norm is a normalization constant needed to compare the theoretical PDF
#to the histogram of simulated values

norm = Nsamp*binWidthHist[0]

f1 = plt.figure()
```

```
plt.subplot(2, 2, 1)
plt.plot(trials, values, 'o')
plt.xlabel('trials')
plt.ylabel('values')

plt.subplot(2, 2, 2)
plt.plot(binCenterHist, histValues1, 'o')
plt.plot(binCenterHist, norm*expModelPDF(a, binCenterHist), 'k')
plt.xlabel('values')
plt.ylabel('counts')

plt.subplot(2, 2, 3)
plt.plot(binCenterCumHist, cumHistValues, 'o')
plt.plot(binCenterCumHist, expModelCDF(a, binCenterCumHist), 'k')
plt.xlabel('value')
plt.ylabel('cumulative counts')

plt.tight_layout()

# These commands output the histogram and cumulative histogram results to csv files
# for plotting

np.savetxt('nonuniformHist.csv',
           np.transpose([binCenterHist, histValues1]), delimiter = ",")

np.savetxt('nonuniformCumHist.csv',
           np.transpose([binCenterCumHist, cumHistValues]), delimiter = ",")
```

Increase N_{samp} from 10 to 1000 and see what happens. Note the function syntax at the beginning of the program defining the *expModelPDF* and the *expModelCDF* that we can compare to the histogram and cumulative histogram (obtained by applying `cumsum()` to a histogram of our Monte Carlo results with very fine binning). Note that, to compare the exponential probability distribution to the simple histogram, I need to be more careful with the normalization (the normalization depends on bin-size and the number of samples, see definition of variable *norm*).

The last line of code (`savetxt()`) saves the cumulative histogram data to a text file with title *nonuniform.csv*. The `transpose` command ensures that the first column is the array *binCenter* and the second column is *cumHistValues*. You can open this file in Excel or a text editor. The purpose of creating this file is so we can import it and plot it with another program. Why would we want to perform plotting in another program? Python of course can produce perfectly nice plots via matplotlib.pyplot. However, when we create final plots for papers and reports, we want them to look very nice and I believe it takes too much effort to do this via command line (this is naturally a matter of opinion). Another option is *Veusz*, a shareware Python-based plotting program with a graphical interface that enables easy creation of plot-containing graphics that can be quickly made and formatted (see *https://veusz.github.io/*).

6 Optional: Plotting using Veusz

Here I will provide very brief instructions for how to plot the file "nonuniform.csv" using Veusz. Open Veusz. In Veusz go to *data* menu and select *import*. Use *browse* button to open the file (the preview should look reasonable–you should see the first couple values in columns 1 and 2). Then click *import*. Back in main-window you should see the data loaded in the data window. Now click on the plot in the main window and delete it (*edit* menu-*delete*). There are two rows of icons on the top of the screen. On the bottom row, click on the 3rd icon from the left (looks like xy axis, see Fig. 2): this will create a new base graph and an axis should appear in the main window. Click on this axis. Then click on 5th icon from left (looks like axis with data). This will allow you to add data to the axis. In the properties window, let x-data be *col1* and the y data be *col2* (or whatever you have decided to name your imported data). Now you should see the data appear! In the

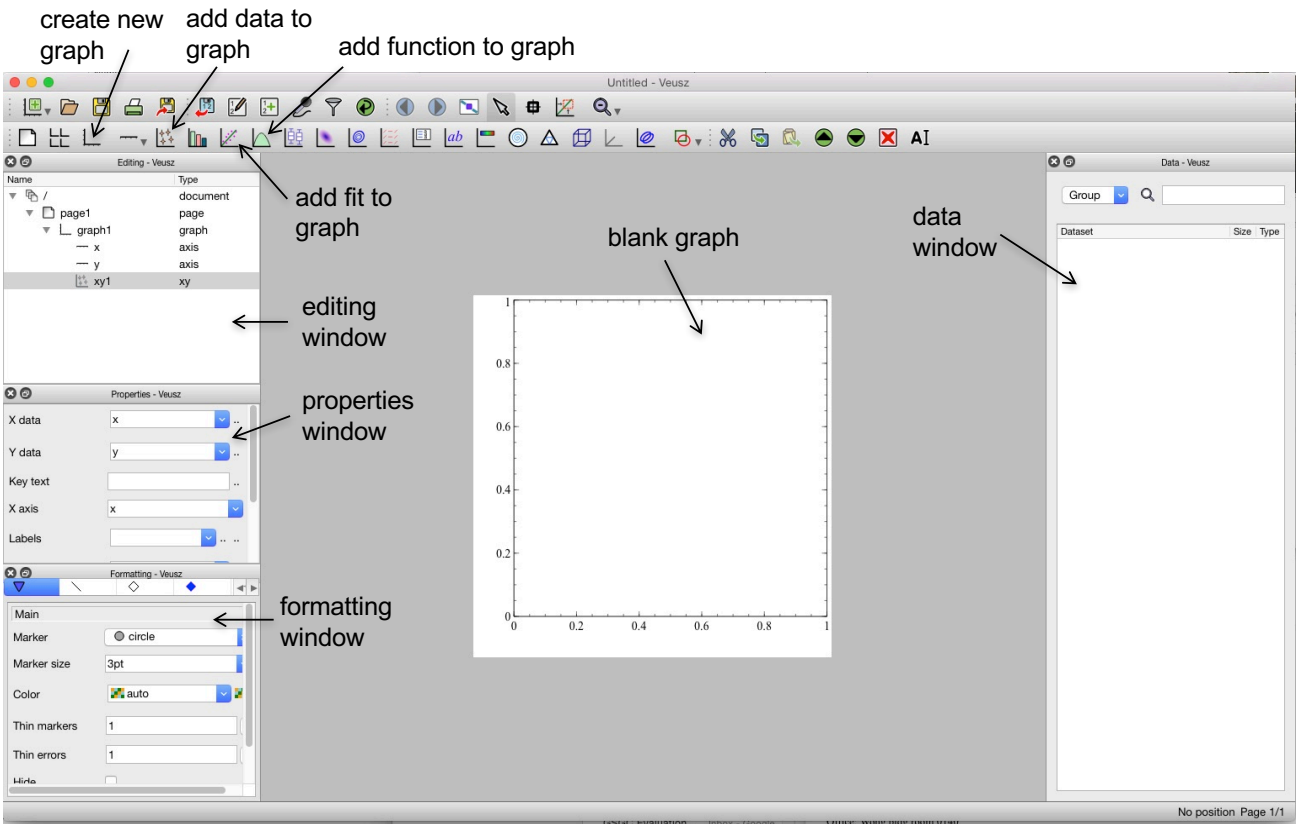

Figure 2: The Veusz graphical interface. The interface is now configured to have one graph with data (but no data has yet been uploaded). Note the icons for adding a new graph and adding data/functions to an existing graph.

formatting window, you can play with the way the data is displayed. Selecting the value 1 for '*Thin markers* will ensure all points are displayed. If you don't want the data points to be connected by a line, go to the line options and click *hide*. I won't go into more detail here, but it is worth spending some time playing with different display options (you can change point color, size, outline, etc.).[10] Finally, to add a function, hit the 8th icon over (looks like a filled inverted parabola). In the properties window, type the function you want to appear (Veusz uses Python syntax, so ** for power). Typing 1-exp(-x) will immediately display the cumulative probability density against the data! (maybe you want to increase the line thickness). To add axis labels, click on x-axis and y-axis in the editing window and type the appropriate labels in the properties window.[11] Note that the axis and every new element added to a given graph (data, function, fit) will appear in the editing window and can be easily accessed for modification by simply selecting it.

OK, I think Veusz is pretty nice, especially for what it costs: nothing. You have to get used to the graphical window, but once you do I feel it is easier than doing everything via command line. You can easily make multiple plots in one figure (try it), format them at will. You can also do curve-fitting and there is an array of advanced features like contour and polar plots. There is only one thing I genuinely do NOT like about Veusz: this is that adding error-bars is harder than it should be. However, once you figure it out (I will explain), it is not that bad.

[10] I suspect students wonder why we fuss so much over plot formatting. The issue is that the exact way a plot looks will have tremendous impact over how the results are interpreted in a publication. For this reason you are graded directly on the aesthetics of data presentation.

[11] If you want to change label font and/or size, go to *edit* menu-*Default-styles-font*. The default label size is a little small.

7 The Exercise

7.1 Generating Random Numbers with Linear Probability Distribution

You are now ready to start programming and applying your own non-uniform random number generators. First, to directly test what you have learned, you will generate random numbers distributed according to the below linear probability density:

$$p(x) = Ax \tag{4}$$

defined on the interval 0 to 1. The constant A, of course, is determined from the normalization requirement. You will need to find the cumulative probability distribution $c(x)$ for the linear distribution and then apply the rule $x = c^{-1}(r)$ Verify quantitatively that your random number generator works. How might you attach an error bar to the histogram/cumulative histogram points? (e.g. the error associated with a certain fixed sample number). Also show that you understand the values of the mean and standard deviation produced by your generator.

7.2 Generating Random Numbers with Gaussian Probability Distribution

Second, you will implement a random number generator to create values with a Gaussian distribution. There is a trick to doing this efficiently, known as the *Box-Muller transformation*. The idea of Box-Muller is to consider a 2D Gaussian probability distribution:

$$p(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2}{2\sigma^2}\right) \exp\left(-\frac{y^2}{2\sigma^2}\right) \, dxdy \tag{5}$$

and then transform the distribution to polar coordinates $(x, y) \to (r, \theta)$:

$$p(r, \theta) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{r^2}{2\sigma^2}\right) rdrd\theta \tag{6}$$

We can then introduce the variable $\mu \equiv r^2/(2\sigma^2)$ and transform $r \to \mu$:

$$p(\mu, \theta) = \frac{1}{2\pi} \exp(-\mu) d\mu d\theta \tag{7}$$

Note that we can write $p(\mu, \theta)$ as the product of the distribution $M(\mu) \equiv \exp(-\mu)$ and $\Theta(\theta) \equiv 1/2\pi$. The former is the exponential distribution we already know how to simulate. The latter is just a uniform distribution over the angle θ (i.e. the angle is a uniform random variable in the interval 0 to 2π. The fact that $p(\mu, \theta)$ can be written as the product of $M(\mu)$ and $\Theta(\theta)$ means that the two distributions are statistically independent and you can Monte Carlo them separately using the methods you have already learned. You can then use the relationship between (x, y) and (r, θ) to obtain the corresponding one-dimensional Gaussian distributions for x and y (you only need one of these). Verify quantitatively that your random number generator works. Note that Python does provide built-in random number generators for common distributions,[12] which you can use in the future, but now you know how they work!

7.3 Probability Distribution of Mean

You will now use your random generators to explore some interesting questions. It is best to write a single self-contained script to address each question. First, using the uniform, linear and Gaussian random number generators, find the probability distribution associated with the mean of n values generated according to these distributions. You should produce plots that clearly show the behaviour the mean n-value distribution, for each distribution type, as n gets large. Also compute the standard-deviation of the mean n-value distribution as a function of n for each distribution type and discuss the behaviour, in particular the limiting behaviour for large n.

[12] To generate Gaussian distributed random numbers with mean mu and standard deviation sigma, use the command gauss(mu, sigma). For a complete listing of available built-in random number generators see *https://docs.python.org/2/library/random.html*.

7.4 Simulating Measurement of Digitized Values (Optional Extra Credit)

Computers hold data in memory as a collection of bits (binary digit, 0 or 1). A collection of eight bits (called a "byte") is the smallest number of bits computers use to represent numbers. For example, an "8-bit unsigned integer" can represent an integer from 0 to 255 (to understand why, note that 2 bits can represent just $2^2 = 4$ numbers, say the integers 0 (00), 1 (01), 2 (10), and 3 (11). Eight bits can represent $2^8 = 256$ numbers). If we have some analog (continuous) signal and we want to represent it using bytes, we have to break the analog value into 256 discrete values (including 0). In this exercise we will pretend that we are measuring some physical quantity x that lies in the range 0 to 1 and our measuring system can only store the values in 8-bit units, so the data is necessarily output to the computer in 8-bit units (the computer, naturally is not limited to 8-bit formats). The question is: can we measure x more precisely (e.g. have an error-bar smaller than) that allowed by the digitization? What do you think? Discuss with your team members!

We will address this question by using our Gaussian random number generator to simulate repeated measurements of x. The key idea is that we will "pretend" to have the capability of adjusting the measurement uncertainty of x. In particular, the uncertainty on x can be tuned in our simulation by simply varying the standard-deviation (σ) of the Gaussian generator. The first step is to pick some value for x (maybe 0.5?), which is unimportant. Then, for a given choice of σ, generate N_s "measurements." We then enforce the 8-bit digitization, placing our continuous samples into the 256 discrete bins on the 0 to 1 scale necessitated by our 8-bit parameterization. We then repeat this procedure, generating multiple digitized x-distributions, and find the mean for each generated x-distribution. Once we have the list of mean-values for each generated digitized distribution, we can find the uncertainty on the mean (standard-deviation of list of mean x-values) for a given N_s. We can then ask: "in the presence of digitization how does the uncertainty on the mean-value of x depend on our measurement uncertainty (σ)?" Compute the uncertainty on the mean-value of x as a function of σ (for several N_s values) and discuss the behaviour. What is a smart range over which to vary σ? How do you deal with the case when only one bin-is filled? (i.e. your instrument returns only one digitized-value?)