# Introduction to Monte Carlo Simulation using Python

Peter Crowther, Joshua Robinson and C. Patrick Royall
HH Wills Physics Laboratory, Bristol

November 22, 2016

## 1 Pre-session preparation

### 1.1 Installing Python

Complete the instructions in this first section before attending the first session.

We assume you are either using a Windows or Mac PC. If you are running Linux then you can probably work it out for yourself! A basic understanding of the terminal (mac) or command prompt (windows) is assumed. If not then Google a tutorial. We assume that you have not previously installed a Python 3 distribution to your system.

Download the Anaconda Python 3.5 distribution from `https://www.continuum.io/downloads`. This includes the Python language as well as useful libraries and tools. Install following the instructions and accepting default values.

Open a terminal window (mac) or command prompt (Windows). Navigate (using the "cd" command to navigate the file system) to an empty directory (I suggest somewhere in My Documents) where you want to store files generated in these lectures. Type:

```
ipython notebook
```

After a few moments this should open a window in your internet browser. In the top right hand corner click "new" and under notebooks select Python 3. This should open a Python notebook. This is an interactive Python session where your code is input into "cells" on the page. Once you have written some code in a cell, press the "Shift" and "Return" keys simultaneously to execute the code in the cell.

To test your distribution has installed correctly type the following into a cell and execute it:

```python
import numpy as np
print("Hello World")
```

```
> Hello World
```

In this tutorial, the numbered code on the light grey background is code that you should execute. Bold text is the expected output from that code.

**Ensure that this works correctly before attending the first session.** If in doubt, collaborate with one of your colleagues also taking this course to get it working.

### 1.2 Installing Ovito

Once we have created done some simulations it is nice to be able to visualise the trajectories of the particles. We will do this using some software called Ovito. Download from: `http://www.ovito.org/index.php/download`. The program does not need installing as such, just unzip it and run from the folder.

## 2 Basic Python structures

### 2.1 Basic Maths

You can just start typing maths into Python and it will do it.

```python
1+2
```

```
> 3
```

Division in Python 3 gives a decimal answer as you would expect.

```python
5/2
```

```
> 2.5
```

Beware the division operator in Python 2 (you should all be using Python 3!). The above would give an integer division and so return an answer of 2.

The symbol for power is a double star. A square root can be achieved by raising to the power of 1/2.

```python
2**3
```

```
> 8
```

## 2.2 Variables

We can store values in variables. You can think of these as "buckets" which hold values. You can call variables anything you like as long as it doesn't start with a number. In addition it is a very poor idea to use variable names that are reserved keywords elsewhere in Python such as "print", "sum" or "list".

```python
a = 6 + 3
print(a)
dead_parrot = "This parrot is no more! He has ceased to be!"
print(dead_parrot)
```

```
> 9
> This parrot is no more! He has ceased to be!
```

Variables have types. The main types you need to worry about are integers, floats and strings. Integers are whole numbers, floats are decimals and strings are text. Operations between floats and integers are permitted and the answer will be expressed as a float.

```python
spam = 7
eggs = 0.5
food = spam+eggs
print(food)
```

```
> 7.5
```

Operations between strings and number types are generally not permitted since they make no sense.

```python
song = "I'm a lumberjack and I'm OK" + 1
```

```
> TypeError                                 Traceback (most recent call last)
<ipython-input-47-84d795fdab55> in <module>()
----> 1 "I'm a lumberjack and I'm OK" + 1

TypeError: cannot concatenate 'str' and 'int' objects
```

Strings can be concatenated though using the plus symbol.

```python
line_one = "I'm a lumberjack and I'm OK, "
line_two = "I work all night and I work all day"
song = line_one + line_two
print(song)
```

```
> 9
> "I'm a lumberjack and I'm OK, I work all night and I work all day"
```

## 2.3 Lists

Lists are a simple data structure that allow the storage of multiple values. Lists are designated by square brackets. You can give lists any name you like as long as it isn't a reserved command (like print).

```
numberlist = [1, 2, 5]
print(numberlist)
```

```
> [1, 2, 5]
```

You can fetch a single value from a list by subscripting the list. You can fetch multiple values by asking for a range.

```
numberlist = [1, 2, 3, 4, 5, 6, 7]
print(numberlist[4])
print(numberlist[0:3])
```

```
> [5]
> [1, 2, 3]
```

Notice how the first command returns 5. This is because Python lists (and many things in programming) start counting at zero. This means asking for the zeroth element would return 1. The second command returns only three values. This is because the lower bound is inclusive but the upper bound is exclusive (for mostly the same reasons that things start counting at zero not one). The append command will add things onto the end of a list.

```
numberlist.append(8)
print(numberlist)
```

```
> [1, 2, 5, 8]
```

## 2.4 User input

Somtimes it is useful to accept some user input while the program is running and operate on this input. This can be achieved with the input command.

```
person_name = input("What is your name?")
print("Hello " + person_name)
```

```
> Hello Arthur Pewty
```

Note that the input function always interprets the users input as text. This means that if the input needs to be interpreted as a number then it should be converted explicitly to an integer or float using int() or float()

```
user_input_number = float(input("Enter a number of years: "))
print("Two more than your number is: ", userinoutnumber+2)
```

```
> Two more than your number is: 42
```

## 2.5 Flow control using if

Programs can only carry out very simple tasks if they do the same thing every time. If programs are able to make decisions based on values then this can make them much more powerful. The "if" statement is the most simple of these flow control tools.

```
a = 4

if a < 10:
  print(a, " is less than 10.")
else:
  print(a, " is not less than 10.")
print("This is printed regardless of the value of a")
```

```
> 4 is less than 10.
```

Note how the program flow is controlled by the indentation. Only the indented if statement that is true is printed. Since it is outside of the if statement, the last sentence is always printed.

If you recall the types in section 2.2, you will notice that that the "print" function seems happy to combine a numeric and string type. This is because "print" is a special function which converts arguments to strings before printing them. It will only do this if the arguments are separated by a comma as above.

More complex questions can be asked by asking multiple times using the elif statement. The elif statement only executes if the previous statements have been false

```
num_penguins = 17

if num_penguins < 10:
  print("You have too few penguins.")
```

```python
5  elif num_penguins > 10:
6    print("That is quite a lot of penguins")
7  elif num_penguins == 10:
8    print("Just the right number of penguins")
9  else:
10   print("This statement should never be printed")
```

```
> That is quite a lot of penguins
```

It is important to understand the distinction between comparions and assignment. Notice how on line 1 the assignment operator is one equals sign while on line 7 the comparison operator is two equals signs. The assignment assigns a value to a veriable while the comparison operator is asking whether two values are equal.

**Task 1. What happens if you change num_penguins?**

**Task 2. Write a program that takes two numbers from user input, stores them in two variables and then checks whether the numbers are equal. You should display a relvent message depending on whether or not they are equal.**

## 2.6 Loops

For loops are a flow control tool that can be used to repeat a command multiple times.

```python
list_of_numbers = []
for loop_number in range(0, 5):
  list_of_numbers.append(7)
print(list_of_numbers)
```

```
> [7, 7, 7, 7, 7]
```

**Task 3. What would happen if the print line was indented also?**

We can use the value stored in loop_pointer to count as well:

```python
list_of_numbers = []
for loop_number in range(3, 7):
  list_of_numbers.append(loop_number)
print(list_of_numbers)
```

```
> [3, 4, 5, 6]
```

Note how the bounds of the range command work. The first value (3) is printed but not the last value (7).

## 2.7 Flow control using while

Sometimes it is useful to have a conditional loop. A while loop, repeats a process untill a condition is met.

```python

secret_number = 4
user_number = 0

while secret_number != user_number:
  user_number = int(input("Guess my number by entering an integer
     between 1 and 10: "
  print("No, that's not it, guess again.")
print("Yes, that's it.")
```

```
 > Guess my number by entering an integer between 1 and 10:
>> 7
> No, that's not it, guess again.
> Guess my number by entering an integer between 1 and 10:
>> 4
> Yes, that's it.
```

The comparison operator != checks whether something is not equal. While the two values are not equal the loop will repeat. As soon as they are equal, the loop will exit and the next command will be printed.

## 2.8 Imports

Sometimes we want functions from another package that are not included in the basic Python set of functions. We can import these packages. Here we import the function "random" from the package "random" and then use it.

```python
from random import random

print ("Your random number is: ", random())
```

```
> 0.29497908252
```

Packages only need to be imported once per session. This means that imports are usually done at the top of the Python script as the first thing that is done.

The random function prints a random decimal number between 0 and 1.

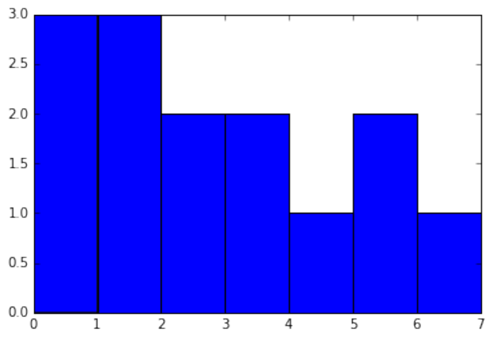
**Task 4. Write a for loop that will print out a random number between 0 and 5, 5 times.**

### 2.9 Histograms

There is a very powerful package called matplotlib that will allow us to draw graphs.

```python
import matplotlib.pyplot as plt
%matplotlib inline

listofnumbers=[0.2, 0.3, 0.4, 1.2, 1.7, 1.9, 2.2, 2.5, 3.1, 3.4,
    4.6, 5.1, 5.6, 6.0, 7.4, 7.4, 7.6]
plt.hist(listofnumbers, bins=[0, 1, 2, 3, 4, 5, 6, 7])
plt.show()
```



First we import the matplotlib library, then the command "%matplotlib inline" tells ipython to plot the graphs in the workbook. Again you only need this command once per session so it is common to put it at the top with the import commands. Then we generate a list of numbers and plot them as a histogram.

**Task 5. What does the bins argument do? What happens if you leave it out entirely?**

**Task 6. Generate 1000 random numbers between 0 and 10 using a for loop and append them to a list. Plot a histogram of this list.**

## 3 Stochastic modelling

### 3.1 Calculation of pi

Some processes can be modelled using random numbers. One of the simplest examples of this is finding the value of pi. Pi is the ratio of a circle's radius to its circumference. Equivalently we can say that Pi is the ratio of area of an inscribed circle of a square to the area of the square as shown in Fig. 3.

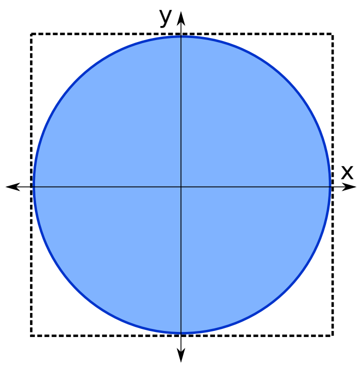


Figure 1: The blue circle is inscribed in the black dashed square.

If we pick a series of random points inside the square then some will fall inside the circle and some will fall outside. The ratio of these two quantities will tell us the relative areas of the two shapes. From this it follows that we can calculate pi using this method.

How can we tell if a random point is inside the circle or not? Pythagorus' theorem tells us the hypotenuse of a right angled triangle. For a circle radius 1 we know that a point is inside the circle if:

$$\sqrt{a^2 + b^2} < -1, \tag{1}$$

where $a$ and $b$ are the side lengths of the triangle. Since the circle is symmetric, we don't need the whole thing, just a quarter will do. This makes the generation of random numbers easier since we can generate numbers between 0 and 1 to put random coordinates inside a box of side length 1. This scheme is shown in Fig. 2

**Task 7. Using all of the above techniques, calculate pi using random numbers. Try using 100, 1000, 10,000 or more random samples, how does this affect the error? If you calculate pi 1000 times and plot a histogram of these values, what sort of distribution would you expect the values to follow and why? What is the nature of the relationship between number of samples and distribution width?**

If you find this task too hard, there are some hints in Appendix A. Try not to look at them unless you are really stuck. Learning (of programming in

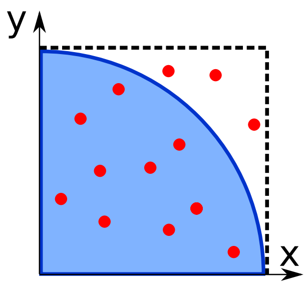


Figure 2: If we pick random points on the figure some will fall outside the circle and some inside. The ratio of the two allows us to find pi.

particular) works much better if you got through the process of problem solving rather than just reading the answer. The hints in Appendix A are graded in terms of difficulty. The first hint gives you a little help and the subsequent hints more help.

## 3.2 Subroutines

Now you have a slightly larger program it can be convenient to split it up into smaller subsections. This makes the logic of the program easier to follow and makes the code easier to maintain. A subroutine is a collection of code that is united under one title that can be “called" from elsewhere. While not exactly synonymous, this document uses the words *function* and *subroutine* interchangeably.

```
def hungarian_phrasebook():
  print("My hovercraft is full of eels.")

for i in range(0, 2):
  hungarian_phrasebook()
```

```
> My hovercraft is full of eels
> My hovercraft is full of eels
```

While this does break up the code somewhat, the function is not very useful since it does exactly the same thing each time. It would be much more useful if it could do different things depending on how we called it. This can be achieved by passing a value to the function called a parameter.

```
def adding_six(input_value):
  print(input_value + 6)

adding_six(3)
```

```python
adding_six(14)
```

```
> 9
> 20
```

It is a common convention to use a function called main as the entry point of the code. This means that the code is run starting in main and then splits off into other subroutines.

```python
def adding_six(input_value):
  print(input_value + 6)

def main():
  for i in range(0, 2):
    adding_six(i)

main()
```

```
> 6
> 7
```

Functions don't just have to print values. They can do some processing and then return the value to the main loop using the "return" command.

```python
def adding_six(input_value):
  return input_value + 6

def main():
  number_list = [1, 2, 3, 4]

  for i in range(0, len(number_list):
    number_list[i] = adding_six(number_list[i])
  print(number_list)

main()
```

```
> [7, 8, 9, 10]
```

We pass each value of "number_list" into the subroutine "adding_six" which adds six and then returns the value to be put back into "number_list". Note the use of the "len()" function inside the range function. "len()" returns the number of elements in a list, in this case in combination with "range" it is an easy way to iterate through the whole list.

**Task 8. Add subroutines to your code to split it up into sections.**

With larger programs it can be convenient to split subroutines up into entirely different files. For example, if we have some simulation code to run a Monte

Carlo simulation it would be sensible to split any analysis code off into a separate file. You will see this structure in the Monte Carlo code you will work with later. If we have a file called *main.py* and another file called *analysis.py* (in the same directory) containing the subroutine *do_analysis* then main.py might look like:

```python
from analysis import do_analysis

def get_random_number()
  # Chosen by fair dice roll, guaranteed to be random
  return 4

def main()
  for counter in range(0, 100):
    test_number = get_random_number()
    do_analysis(test_number)

main()
```

By importing the file *analysis* in the first line, we can call any functions it contains as if they were in the same file.

While this may seem a little complex at first, it is vital to keep code organised as it grows. This will save a lot of time later down the line. Note the comment in one of the functions. By proceeding a line with a hash (#) we can insert text comments that are ignored by the Python processor. This aids in understanding the function of a particular piece of code. Commenting code is another important coding habit.

## 4 Speed, Efficiency and Numpy

Computers are fast but not infinitely fast. This means that while trivial calculations can be done using any method, more complex calculations such as simulations of hundreds or thousands of particles often need more care. Python is beautiful and easy to use but it is not the fastest programming language. This means that to do things like Monte-Carlo simulations we must take some consideration of the efficiency of the programming constructs that we use.

A very simple way to add 1 to each value in a list would be to step through the list and add 1 to each value.

```python
from time import time

```

```python
start_time = time()
long_list = range(0, 10000000)

for i in range(0, 10000000):
  long_list[i] = long_list[i] + 1
end_time = time()

print(end_time-start_time)
```

```
> 2.05770301819
```

Note the use of the time function in this example. This is a very simple way to assess the speed of code using the system clock. It is not the canonical way to measure performance but like many things in this short tutorial, it is simple and effective if not the best way to do it. If we think about it then that is fairly impressive, it has been through that loop 10,000,000 times in 2 seconds. But can we make it faster? One key concept in writing speedy code is vectorisation, instead of a loop we want to apply the transformation to every element in the list simultaneously. Lets try that:

```python
short_list = range(0, 5)
print(short_list)
short_list = short_list + 1
print(short_list)
```

```
> [0, 1, 2, 3, 4]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-5d83b7f97176> in <module>()
      3 print short_list
      4
----> 5 short_list = short_list + 1

TypeError: can only concatenate list (not "int") to list
```

Basic Python doesn't understand vectorisation. Instead of adding one to each element in the list it is trying to add the number 1 onto the end of the list. This is where numpy (short for numerical Python) comes in. It is a library that adds data structures and functions suited to mathematical operations. A numpy array looks like a list but has some subtle differences.

```python
import numpy as np
short_np_array = np.arange(0, 5)
print(short_np_array)
short_np_array = short_np_array + 1
print(short_np_array)
```

```
> [0 1 2 3 4]
> [1 2 3 4 5]
```

The numpy function "arange" generates a numpy array. This is a type of object like a Python list which holds a series of values. The only difference we see in the output that is printied is that while Python lists have commas seperating the values, a numpy array doesn't. One of the big differences between lists and numpy arrays underneath is that arrays allow vector operations. This means mathematical operations can be applied to the whole array simultaneously and quickly.

```python
from time import time
import numpy as np

start_time = time()
long_np_array = np.arange(0, 10000000)

long_np_array = long_np_array + 1
end_time = time()

print(end_time-start_time)
```

```
> 0.0694818496704
```

Using numpy provides an impressive speedup of nearly 30 times for this simple operation. You will see use of the numpy library in the code you will use for the Monte Carlo simulations.

## 5 Running code outside of iPython

iPython is good for experimenting as you learn to build code but the normal distribution method for Python code is a simple text file. A Python script is stored as a plain text file with the ".py" file extension. To run such a Python script open a terminal or command prompt, navigate to the directory where the file is installed and type:

```
python3 filename.py
```

replacing *filename.py* with the name of the script. This will directly run the file and the output will be printed to the screen.

# 6 Monte Carlo Methods

## 6.1 Introduction

Monte Carlo methods are stochastic simulation methods. The method itself is very general and can be applied to many problems in science. Here we study in more detail the method as applied to evaluating time integrals for a many particle system. To simulate the time evolution of a system, particles are random displaced, guided by the changing energy of the system as the particles move.

The outline of a Monte Carlo simulation is quite simple. As we will see however, how we implement each section depends on our system and is not trivial. First, consider the outline of the algorithm. For this outline and when describing general concepts throughout this document, I use psedodcode with a Pythonic syntax. Names with brackets after them indicate functions which we will explore in more detail later.

```
initialise_system()

loop for number_of_sweeps:
  loop for_num_samples_per sweep:
    randomly_chosen_particle = get_random_particle()
    new_position = get_random_displacement()
    energy = get_energy_of_random_particle_at_new_position()
    if new_particle_energy is good:
      move_particle to new_position
    Else:
      don't move particle
  measure_system_properties()
```

## 6.2 Initialising the system

We want to start our simulation with a randomly ordered configuration. For a very low volume fraction system (a small number of particles in a large box), it is realtively easy to initialise the system in a a random fashion.

```
num_added_particles = 0
num_particles_to_add = 1000

while num_added_particles < num_particles_to_add:
  x_coord = random_number()
  y_coord = random_number()
  z_coord = random_number()
  particle_fits = does_particle_fit()
  if particle_fits == True:
    add_particle_to_box()
    num_added_particles = num_added_particles + 1
```

The function does_particle_fit, checks every particle already in the system to see if it will overlap with the position of the new particle since hard spheres are not allowed to overlap. This works fine for a very low volume fraction but as the volume fraction increases, the time taken to find spaces for all of the particles increases exponentially. Eventually it is impossible to randomly pack the particles into the box in this fashion.

For high volume fractions it is best to start the simulation with all of the particles arranged in a crystalline lattice, this ensures that all of the particles fit in the box and takes no time at all. This isn't quite true since some effort must be expended to melt the crystal into a a random configuration before sampling starts by running the simulation for a number of steps before sampling starts. The code used in these sessions uses this method:

## 7 Running the Monte Carlo Simulation

Download the prepared simulation files as a zip archive from `https://github.com/tranqui/monte_carlo`. Unzip the files, open a command prompt/terminal and navigate to the folder. The code is set up with some automated tests to make sure that your Python distribution is operating correctly and all of the code is present and working. To run the tests, call the main script from the terminal without any arguments:

```
python montecarlo.py
```

The result should be a series of tests, completed successfully. The scripts, g.py, lattice.py, atom.py and montecarlo.py contain functions which make the simulation work. All of these functions are called from the script main.py. You should only need to edit main.py. If you are keen, have a look at the rest of the files and try to see how they work.

**Task 9. lattice.py is the simplest of the scripts, open it in a text editor and have a look at the code. See if you can follow what it is doing using the explanation in section 6.2**

Open the script main.py in a text editor. The first 20 or so lines are the docstring. This describes the basic functions of the program. Below the docstring are the import statements which import libraries and functions, some from big packages like numpy and some from the Monte Carlo files in our package. Below this are the main variables which provide the input parameters for the simulation. To run the simulation with completely default values, run the script from the console:

```
python main.py
```

This should print a radial distribution function to the screen and output a coordinate file called “trajectory.atom" to the same directory the code was run from. This file contains snapshots of the positions of the particles at different times throughout the simulation.

Open Ovito and load this file using the "Load File" option from the "File" menu. This should display a snapshot of the particles in the simulation box. The first step in Ovito is to make sure the particles are displayed as the correct size. On the top right of the screen there is a box with a number of options. Click the second one down "Particles" to open the particles menu below. Set the "Default Particle Radius" to 0.5. This represents the correct size of the particles in the simulation. After this we need to load all of the snapshots in the trajectory, at the moment only the first one is displayed. Click on trajectory.atom in the top right menu just below the particles button and then in the menu which appears below, tick the small box which says "File contains time series". This will load all snapshots in the trajectory file. You can now scroll through the trajectory using the scroll bar beneath the main display windows.

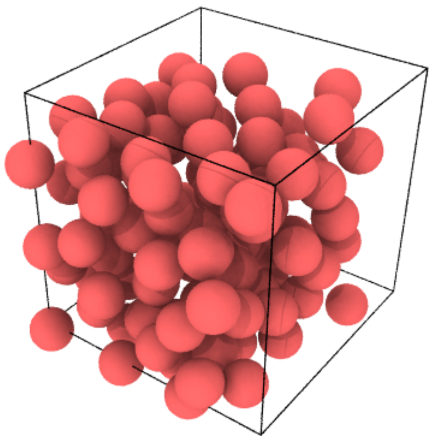


Figure 3: A rendering of one snapshot of the sample simulation produced by Ovito.

# A Stochastic determination of pi hints

## A.1 Hint 1

Try writing out an algorithm in pseudocode. This means writing out a series of logical steps in simple words first without worrying about the exact programming syntax. For a simple loop this might look something like:

```
import needed_libraries

total = 0

for loop from 1 to 10
  add loop value to my total
print the total to the screen
```

Once you have got the basic program flow sorted, you can then convert your pseudocode to valid Python.

## A.2 Hint 2

This is a basic pseudocode example of the algorithm.

```
import needed_libraries

num_inside=0
num_interations = 1000
x_coord = 0
y_coord = 0

for loop from 0 to numiterations
  set x_coord using random number
  set y_coord using random number
  check distance of random coord from center
  if distance <= 1 then
    increment num_inside by 1

my_pi = 4* ratio of inside to outside
print my_pi
```