**PHY1003 – Assignment 4 (Python) – Monte Carlo Simulations**

**Exercise 1 -** Part A:

Here two arrays are created and filled with randomly and uniformly distributed numbers between -1 and 1.

Figure 1: Creating arrays filled with random numbers

Here the numpy (np) library is imported and used as these operations are not available in basic Python. The np.random.uniform function here will generate real and random numbers between -1 and 1 (however, this does not include 1, instead a decimal close to 1 will be used). A uniform distribution means that plotted on a histogram a straight line will be seen, each number will be evenly represented.

Due to using a while loop later in the code (calling upon a function to generate the estimate for pi and draw the graph at the same time) to generate 10n sets of co-ordinates a value of 10n-1 must be used for the x and y array lengths, as they are then called upon 10 times each. Thus, when a task asked for 10n array elements there are 10n+1 coordinates plotted.

Part B:

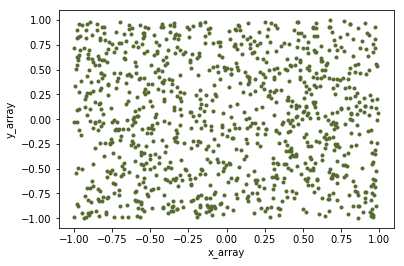


Figure 2: Scatter graph of y\_array against x\_array



Figure 3: Code snippet used to generate graph in Fig. 2

The matplotlib library was imported to allow for graphs to be generated within the program. The above graph has been generated using the x and y\_arrays generated above. Some visual features of the graph have been changed.

Part C:

Circle Equation:

We can use the standard equation of a circle to construct the expression to find which of our points are located in a circle. Firstly, we know that r=1. We also know that since the circle is centred on the origin (0, 0) we know that h and k are equal to zero.

We know that when this equation is true that the circumference of the circle will be defined. Because we also want the points inside the circle to be defined we need to adjust the equation by adding an inequality to state that everything within this radius is included.

This is the expression which defines which pairs of (x, y) points are located within the circle.

Part D:

Here the inequality defined above was used to create a Boolean mask which contains the x and y values which satisfy the inequality. Both x and y values from the co-ordinate pair will need to satisfy the equation to be plotted, rather than just one half of the pair triggering them to be counted as ‘inside’ the circle. Using the plt.plot function again and plotting only the values of x\_array and y\_array which fulfil the condition (this is triggered by the Boolean mask in the square brackets).

Figure 4: Code snippet used to plot circle on the same graph as Fig. 2

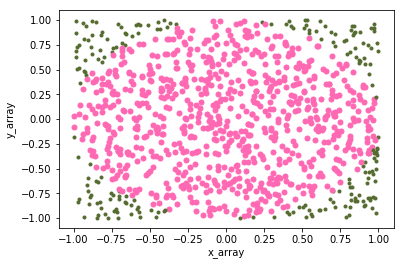
The visual features of the graph such as the marker style, colour and marker size were changed here to allow for distinction between the two sets of data which are now plotted on the same graph.

Figure 5: Graph of circle plot plotted over Fig. 3

Part E:

To use the above plot to estimate a number for pi a ratio of points inside the circle against total points can be calculated and used in lieu of an area. We can derive the formula by considering the area of a circle:

And the area of a square:

We can make the approximation that the ratio of the points (λ) plotted in the circle compared to the total points plotted is equal to the ratio between areas:

We can convert this into the code to the right, which expresses the equation above using a Boolean mask and the length of the arrays to create the ratio.

Figure 6: Formula to estimate value of pi

Part F and G:

|  |  |
| --- | --- |
| # | Pi Est. (10 Values) |
| 1 | 3.28 |
| 2 | 3.36 |
| 3 | 3.04 |
| 4 | 3.0 |
| 5 | 2.96 |
| 6 | 3.08 |
| 7 | 3.04 |
| 8 | 3.2 |
| 9 | 3.0 |
| 10 | 3.44 |

The program was iterated over 10 times using a while loop and a counter to generate ten estimates for pi using ten values each for x and y. The mean value and standard deviation were calculated using the numpy functions, these values are 3.14 and 0.160249804992081, respectively. The accepted value of pi is 3.14159 (5 d.p.), indicating that this is a generally accurate result.

|  |  |
| --- | --- |
| # | Pi Est. (100 Values) |
| 1 | 3.176 |
| 2 | 3.176 |
| 3 | 3.12 |
| 4 | 3.128 |
| 5 | 2.16 |
| 6 | 3.14 |
| 7 | 3.172 |
| 8 | 3.032 |
| 9 | 3.132 |
| 10 | 3.176 |

When the program was iterated over another 10 times but this time using 1000 x and y values each. As can be seen from the graph a much more defined shape was shown and the resolution of the ratio being used as an area is much greater. The mean value was calculated to be 3.1412000000000004. This value can be rounded to 3.1412 (4 d.p.) which we can see is more precise than the previous value. This comes as a result of increasing the resoltion of the graph. The standard deviation has also decreased to 0.04191610668943387 showing that a more accurate result has also been achieved.

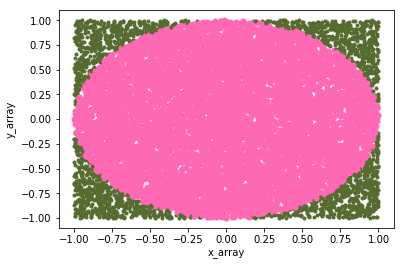


Figure 7: Plot of Fig. 5 with 1000 values of x and y

Part H:

Increasing the number of points from 10 to 1000 reduced the standard deviation from ~0.1602 to ~0.0419. This seems to indicate that increasing the number of points by a factor of 102 decreases the standard deviation by a factor of 4 (changes by 0.25). It can be approximated that the standard deviation after using 10000 (or 10\*103) will be approximately 0.02095. However, this is assuming that there is a linear relationship.

A much more general trend of the standard deviation decreasing can be seen. This is due to the estimation receiving outlier answers less often than answers which are close to the mean value, a consequence of the random nature of the points generated.

**Exercise 2**

For this exercise a program must be used to simulate the result of rolling 3 dice. Here an array was created using random integer values. The higher limit is not included in the random generation, therefore using a lower limit of 1 and an upper limit of 7 means that numbers 1, to 6 (inclusive) are available. Using n in place of how many values to generate means that this program can be easily modified to create a more accurate model.

Figure 7: Using RNG to simulate the rolling of 3 dice

The probabilities generated below were done so using np.random.seed(12345).

Part A:

Here the likelihood of rolling a sum larger than 10 is predicted using the program. A new array is created which is the sum of the arrays containing each die. A Boolean mask is then created to check against which values in the new array are greater than 10. As a True value is equal to 1 and a False value is equal to 0 we can using the len() function to determine the number of rolls (each die rolled once) which satisfy the inequality.

The probability over 10,000 rolls comes out as 0.49932. This is close to a theoretical value of 0.5.

Part B:

For this part the probability that each die will show the same number is calculated. Here the main problem in coding came when trying to compare three arrays. Using three logical operators (ie. ==) produced an error, as did using &&. This lead to creating a fourth array, which is a combination of the first two, and comparing it to the third. This was a very roundabout way of achieving the same result that using a single & would have had (as was discovered later).

The probability over 10,000 rolls comes out as 0.02794. This is close to a theoretical value of .

Part C:

This part asked for the probability that die 3 > die 2 > die 1. The same problem encountered in Part B with comparing three arrays was present here too. The same method was used to compare the arrays, creating new Boolean masks which are combinations of previous masks and comparing those instead. When doing this form of program again a single ampersand will be used instead as it should increase the program speed and it would be easier to code.

The probability over 10,000 rolls comes out as 0.0.9279. This is close to a theoretical value of .