CS5016: Computational Methods and Applications
Assignment 1: Monte Carlo Methods

Sujit Mandava
112001043

Q1. My first observation was that computing and plotting the values of log(n) up to 10^6 using either of the methods(Stirling's Approximation or the direct approach) was impossible. Thus, an alternate method to visualize these values would be to take the logarithm of both sides of the equation and plot the values obtained against each other.

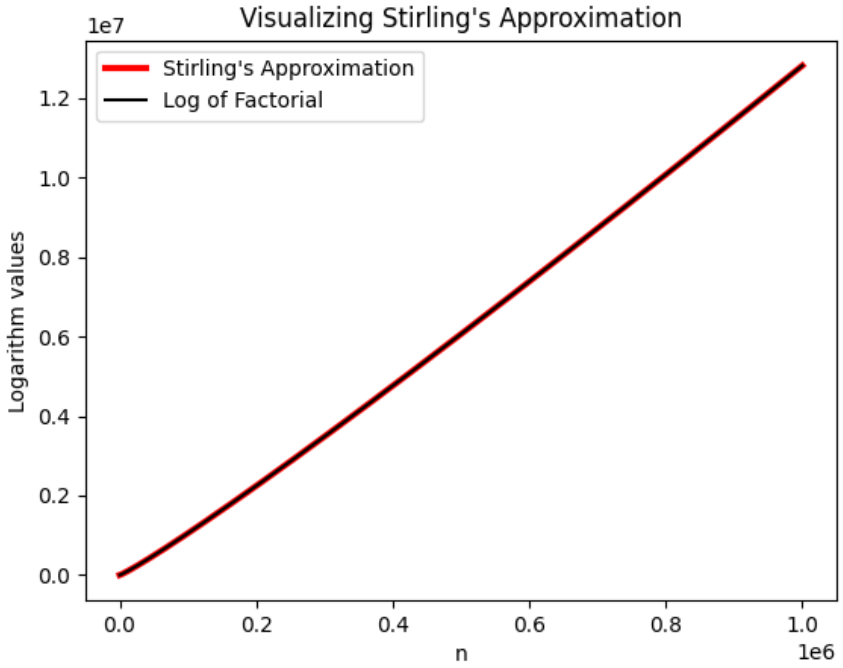
By doing so, we get the following equation:

$$log(n!) = log(\sqrt{2\pi n} * (\frac{n}{e})^n)$$

which evaluates to

$$log(1) + log(2) + \ldots + log(n) = (n + 0.5) * log(n) + 0.5 * log(2\pi) - n$$

We calculate the LHS and the RHS separately and plot them against each other for values of n up to 10^6. Upon doing so, we obtain a graph that looks like this:



The LHS and RHS are computed and stored in lists and plotted using matplotlib.plt.
The LHS is computed by repeatedly adding log(n) to the previous result and incrementing n.
The RHS is computed by using the above-derived formula.

Q2. The Dice class has three attributes: numSides, probSides, and cdf. numSides tells us the number of faces the dice has. probSides tells us the probability of each face being the top face upon rolling it. "cdf" is used to store the cumulative distribution frequency for the dice. It will be used when the dice roll is simulated.

The class has 5 methods:
setProb takes a list of values and assigns these values as the probability of occurrence of each face. However, it only does so if the list satisfies two conditions:

1. If the number of values is equal to the number of sides.
2. If the sum of all values is equal to one.

In case either of the conditions is not met, it raises an exception. It calls another method, checkProbability to check whether the probability distribution is valid or not.

The class constructor method takes one argument: the number of sides. It first sets the numSides attribute. If no value is provided, it defaults the number to 6. If the number given is less than 4, it raises an exception. Then, using the setProb method, it sets the probabilities to that of a fair dice, i.e, equal for all faces.

The roll method is used to simulate the roll of a die. This method emulates the algorithm discussed in class, and thus I will not be elaborating too much on it. Each roll is executed as per the algorithm, and the outcome of every roll is recorded. A list is formed which describes the number of times each face has been rolled. After all the rolls are finished, it takes the final list and plots the number of times each of the faces has been rolled.

The "__str__" method is used to describe how any object of this class would be printed.

Q3. The method estimatePi takes a single argument n: The number of points to be generated for the Monte Carlo method. This method is an exact implementation of the algorithm discussed in class. First, we generate n points using uniform continuous distribution in the square centered around the origin with sides $-1 \leq x \leq 1$, $-1 \leq y \leq 1$. For every new point generated, we check whether it lies within the unit circle centered at the origin ($x^2 + y^2 = 1$). If it lies in this circle, we increase the counter by one. We also store the ratio of the number of points in the circle to the total points generated for every new point generated. Once all points have been generated, calculate the value of pi at each of the ratios using the formula.

Q4. To create the random text generator, we first split the problem into 2 phases: input processing and text generation. The class TextGenerator has one attribute prefixDict used to store the associated prefix dictionary. It has also two methods, assimilateText, and generateText.

The method assimilateText takes in the input file and creates the prefix dictionary for every two simultaneous words. First, it uses the split method to break the entire text into words, using “ ” as the separator. We then check whether the number of words is at least 3, to create the same. If not, we raise an exception.

The keys of our prefix dictionary will be a tuple of strings. We take the first two words of the text to be the first and second elements of our tuple respectively. If this tuple does not exist in our dictionary, we create a new entry with this key whose corresponding value is an empty list. The next word in the text is then added to the list of the current tuple. The current tuple is then updated such that the second word of the previous tuple is the first word of the new tuple, and the word added to the list is the second word of the new tuple. The next word of the text is now the word being added to the tuple’s list. This process is repeated until the end of the text is reached.

The method generateText uses the created prefix dictionary to generate random text. It takes one argument, the number of words, and an optional argument for the starting word. If there is no second argument, it takes a random tuple as the starting tuple. If there is an input, it checks the prefix dictionary for all the tuples whose first word is the given word and chooses one at random. If no such tuple exists, it raises an exception.

Once the tuple is chosen, it first logs the first word of the tuple. Then, from the list associated with the chosen tuple in the prefix dictionary, it chooses a random word. It then updates the tuple so that the second word of the original tuple and the newly chosen word is the first and second words of the updated tuple respectively. This process repeats until the number of words logged is equal to the required number of words to be generated.