CS5016 : Computational Methods and Applications
Assignment 1 : Monte Calro method

112001051
VM Sreeram

Question 1

The direct application $n!$ and Stirling's formula does not work for large n, as it causes overflow in Python. Hence, it was decided to apply $\log_e$ on both $n!$ and Stirling's formula to compare. Both LHS and RHS are computed and stored in list for each i from 1 to n. This list is plotted using Matplotlib, and is shown as log plot in Figs 1.1-1.3. The relative error, and hence percentage error was computed for each i from 1 to n, and is plotted as percentage error plot in Figs 1.1-1.3.

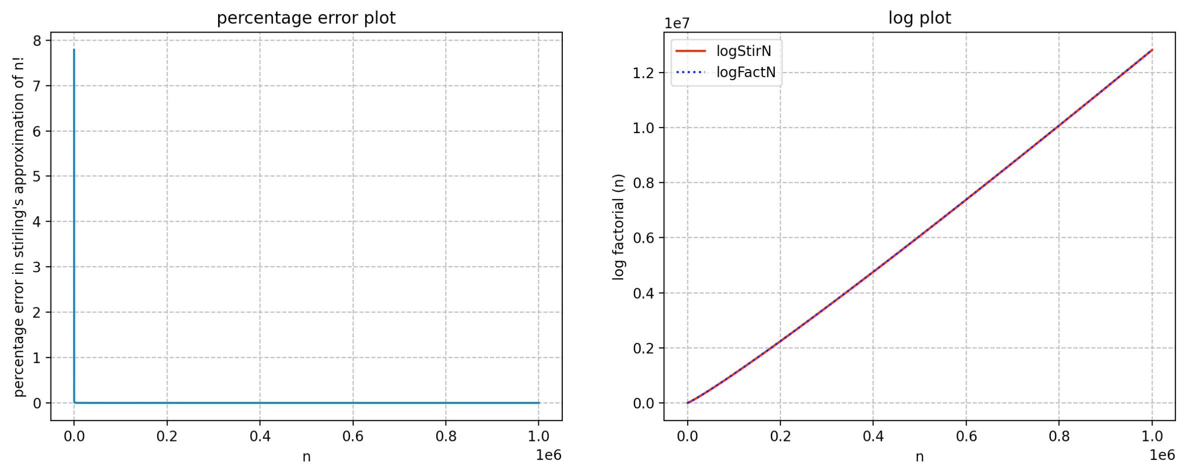

Fig 1.1. Output for n=10^6. The percentage error at n=10^6 is 1.9×10^{-5}%.

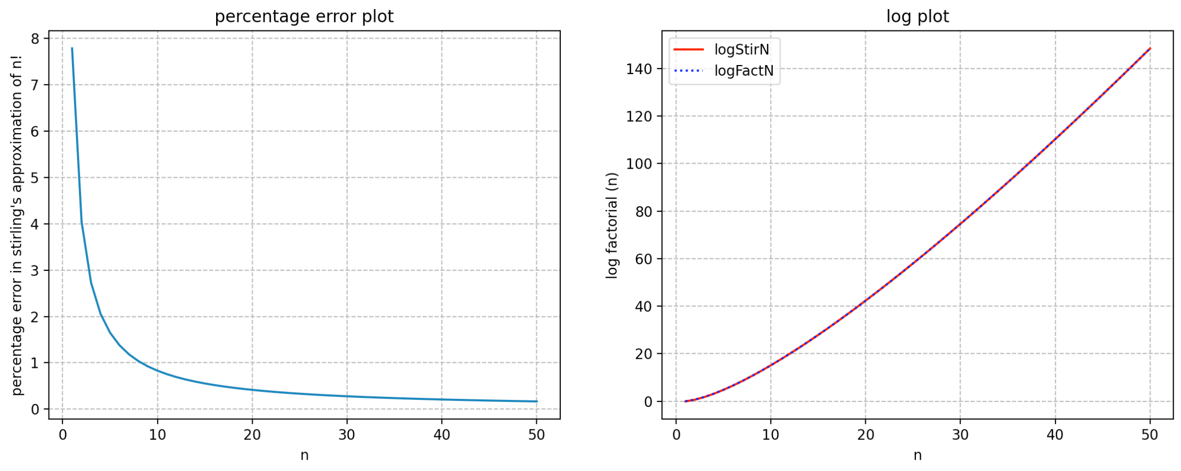

Fig 1.2. Output for n=50. The percentage error at n=50 is 0.166%.

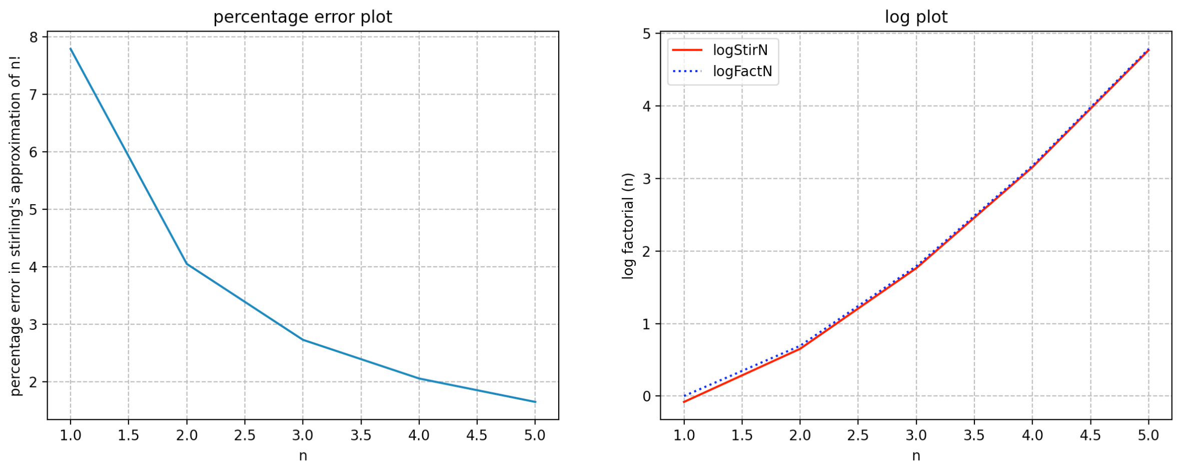

Fig 1.3. Output for n=5. The percentage error at n=5 is 1.65%.

The percentage error in Stirling's approximation of $n!$ wrt $n!$ converges to 0 for even very small n. That's the reason of sharp elbow in Fig 1.1. The plot for n=50 (Fig 1.2) with smoother elbow is also given for ease of comparison. Fig 1.3. is also displayed as it shows the separation in log plots.

Hence, Stirling's approximation of $n!$ is very close to the actual value of $n!$ for large n.

<u>Question 2</u>

It is assumed that the die should have more than or equal to 4 faces. Floating point faces are supported, provided it is not non-zero after decimal point. Also, if sum of provided probability distribution does not sum up to unity or if the length of probability distribution provided mismatches with number of faces, no assumptions are made and an exception is raised, as the incorrect/assumed output may mislead the user who may have did some unintentional mistake.

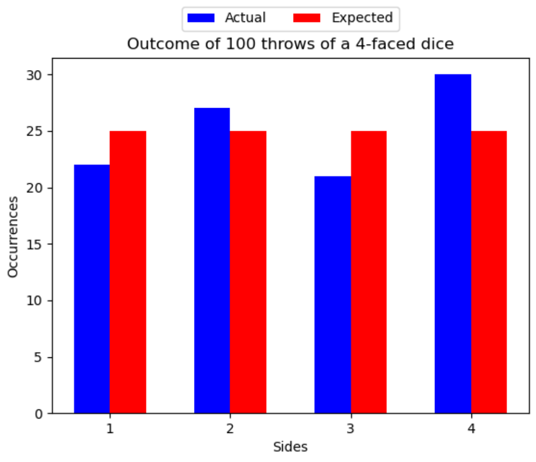

Fig 2.1. Dice with 4 faces with probability distribution {0.25, 0.25, 0.25, 0.25}

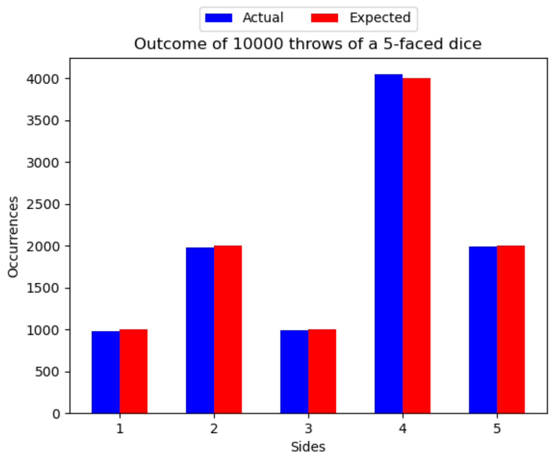

Fig 2.2. Dice with 5 faces with probability distribution {0.1, 0.2, 0.1, 0.4, 0.2}

It was observed that the relative error in actual number of occurrences and expected number of occurrences tends closer and closer to zero as number of throws increases.

<u>Question 3</u>

`np.random.uniform()` was used to uniformly generate points in a unit square centred at origin. It was checked whether or not the generated point is inside a unit circle centred at origin. Then the following approximation of π (Monte Carlo approximation) was applied.

$$\pi \approx \frac{4 \times \text{no of points within the unit circle}}{\text{no. of points within the unit square}}$$

The values of π corresponding to each number of points generated were stored in a list. This is plotted and is shown in Fig 3.1.

It can be observed that inspite of significant variations in the approximation for smaller number of points, the approximated value of π comes closer and closer to the actual value of π. Hence the correctness of Monte Carlo approximation is demonstrated.

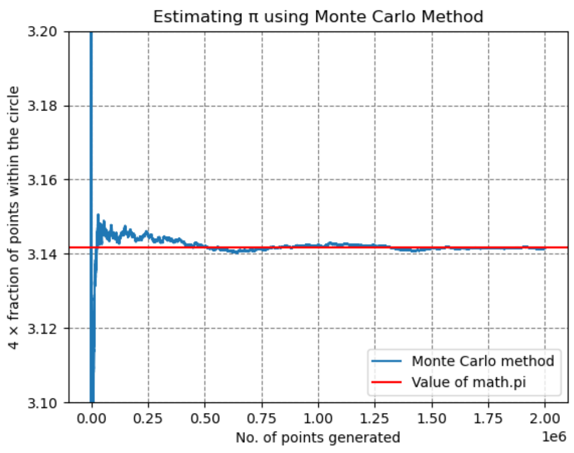

Fig 3.1. Estimation of π using Monte Carlo method.
Estimated π = 3.141496, math.pi = 3.141592653589793.
The final percentage error for this invocation is ~0.003%.

Question 4

`assimilateText()` creates a prefix dictionary which maps the list of third word for each tuple of first two words. A warning is issued when `assimilateText()` is called multiple times, as assimilating same file multiple times just adds same group of words multiple times for each key, and the relative frequency will remain the same (the expectation is maintained). Also, assimilation is programmed to fail if number of words in the file is less than 3, since the logic given in the problem statement can't be implemented.

`generateText()` generates random text based on the triplets in the dictionary. It is enforced that if it wasn't assimilated before, generateText raises exception. If a start word isn't given, a key is chosen in random from the dictionary to be the first pair of words. Cases where further generation can't be done due to missing key in dictionary, or such a start word doesn't even appear anywhere are also handled.