This task focuses on understanding how distributions, the law of large numbers, computational complexity, and accuracy interplay in simulating random processes, specifically cell movements on a grid. It aims to demonstrate how bias emerges and diminishes in random processes and how uniformity develops with increasing iterations. Depending on how the randomness is implemented, it could lead to a variety of distributions, such as the normal distribution, which describes data using a probability density function with mean μ and standard deviation σ and is often used in random number generation (Shiflet & Shiflet, 2014). The uniform distribution ensures that all outcomes have an equal probability in discrete cases, or that equal-length intervals have an equal chance of occurring in continuous cases (Shiflet & Shiflet, 2014). The Bernoulli distribution arises when modelling binary outcomes, and its conditional version, the conditional Bernoulli distribution, is useful when several Bernoulli random variables are conditioned on their sum equalling a specific value, with applications in sampling and hypothesis testing (Chen and Liu, 1997). The beta-binomial distribution applies when the probability of success in a binomial process itself follows a beta distribution, offering a model for over-dispersed data, common in biological and reliability studies (Skellam, 1948; Altham, 1978). Lastly, the Poisson distribution generalizes the binomial distribution by allowing different probabilities of success in each trial, which is useful in survey sampling and logistic regression (Chen and Liu, 1997). These varying distributions help explain how randomness in cell movement can lead to different patterns, and how bias and uniformity evolve as the process progresses. In practice however never can a distribution truly be split uniformly however due to the law of large numbers the outcomes will be slowly converging towards an even probability; so, when trying to find if the distribution is uniform a chi-square test can be implemented the purpose of the chi-square test is to compare the expected and observed outcomes. The chi-square test gives a good indication that .When testing how the distribution was done using the numpy random function “numpy.random.uniform” which in theory should give each of the directions an equal chance of occurring. When it comes to the chi-squared test across three runs it becomes clear that the distribution is uniform with each checkpoint however when you look at the graphs and look at the movement as well the distribution(See Figure 1,2 and 3) it becomes clear that there is a bias this however may be due to the small step size. This sample size may introduce statistical noise that can be reduced by increasing the step size and also increasing the grid size, increasing the step size allows the Law of Large Numbers to impact the findings because as the simulation has more time to run it will converge towards the mean which in this case would be 25% per direction. Increasing the grid size would also mean that it would be easier to get 25% per direction as when the simulation hits the border it can no longer head in that direction. This can be seen in Figures(4,5 and 6) when the sample and grid size are increased, and the distribution is much more uniform. This additional complexity however isnt very impactful because of the simple nature of the task.

## Task 1\_2

When it comes to the additional directions a new way of getting directions is needed to be taken because the generation of a two binary numbers is no longer appropriate instead it is implemented as seen below.

|  |
| --- |
| *Function task\_1\_2(total\_steps, checkpoints, run\_id):*  *Initialize grid\_size as 100*  *Define directions\_map for 8 directions*  *Define moves for each direction (Right, Left, Down, Up, Down-Right, Down-Left, Up-Right, Up-Left)*  *Set initial position (x, y) at the center of the grid (grid\_size // 2, grid\_size // 2)*  *Create an empty list called positions and add initial position (x, y)*  *Create an empty list called directions to store direction history*  *Create an empty dictionary called step\_data to store direction count at each checkpoint*  *For step from 1 to total\_steps (inclusive):*  *Generate a random number between 0 and 1 (rand)*  *Map rand to one of 8 directions using directions\_map and store the direction index*  *Get the corresponding move (dx, dy) from the moves list based on direction index*  *Update the current position (x, y) by applying move (dx, dy)*  *Ensure the position (x, y) stays within the grid bounds (0 <= x < grid\_size and 0 <= y < grid\_size)*    *Append the direction to the directions list*  *Append the new position (x, y) to the positions list*    *If step is in the checkpoints:*  *Count the occurrences of each direction in directions list*  *Store the direction count in step\_data for the current step*    *Return step\_data containing direction counts at the specified checkpoints* |

In practice, the movement isn’t quite uniform in the early steps of 1000 this is due to the addition of the multiple directions making the simulation slightly more complex however the overall computational cost still isn’t very high. But, the for the later step it starts to converge and look more uniform before finally the extra steps allow it to become very uniform though the chi-squared test displays that throughout the checkpoints the distribution is uniform. This makes tasks 1.1 and 1.2 very similar to each other with each of them using the

However the code could be implemented in other ways for example having a random function that selects a choice of direction, a generation of an 360 degree angle, these could be implemented as you see below.

|  |
| --- |
| *Function task\_1\_2:*  *Initialize list of directions: ["Right", "Left", "Down", "Up", "Down-Right", "Down-Left", "Up-Right", "Up-Left"]*  *Initialize list of moves: [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (-1, 1), (1, -1), (-1, -1)]*    *For each step in range from 1 to total\_steps:*  *Randomly choose a direction from the list of directions*  *Find the corresponding move (dx, dy) using the chosen direction*  *Update the position (x, y) by adding (dx, dy), ensuring the new position stays within grid bounds*    *If step is in checkpoints:*  *Record the count of directions up to this point in step\_data*    *Return step\_data* |

|  |
| --- |
| *Function task\_1\_2:*  *Initialize list of directions: ["Right", "Down-Right", "Down", "Down-Left", "Left", "Up-Left", "Up", "Up-Right"]*  *Initialize list of moves: [(1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1), (0, -1), (1, -1)]*    *For each step in range from 1 to total\_steps:*  *Generate a random angle between 0 and 360 degrees*    *Map the angle to one of 8 directions:*  *- Add 22.5 degrees to center the intervals*  *- Use integer division by 45 to find the index for the direction*  *- Use modulo operation to ensure the index is within range 0 to 7*    *Find the corresponding direction and move (dx, dy)*  *Update the position (x, y) by adding (dx, dy), ensuring the new position stays within grid bounds*    *If step is in checkpoints:*  *Record the count of directions up to this point in step\_data*    *Return step\_data* |

The separate complexities of each method are the implemented pseduo code requires typecasting and multiplication compared to the random choice which is the simplest method as it only needs the random generator; finally the third option requires the mapping of the different angles as well as the typecasting.