Victor-Ioan Macovei
Computational Physics MSc
Babeș-Bolyai University

March 25, 2024

HOMEWORK 1 — Warm-Up

Requirements and Setup

Resources

- **Development Environment**:
  - Visual Studio Code with C/C++ extension by Microsoft (required but can be replaced);
  - Python: Version 3.x. (required);
  - VS Code Extensions: Python, GitLens, GitHub Pull Requests and Issues (not required);
- **Compiler and Debugger**: GCC for C/C++ and GDB for debugging (required but can be replaced);
- **Version Control**: Git for cloning the repository (not required);
- **Python Dependencies**: Use of `pip` to install requirements from a `requirements.txt` file (required);
- **Continuous Integration**: GitHub Actions for Continuous Integration (not required).

Setup Guide

- **Cloning the Repository**: The repository was created on the local machine using Git. It is available at GitHub - ssmHW;
- **Setting Up the Development Environment**:
  - Visual Studio Code was downloaded and installed;
  - Required extensions, including "C/C++" by Microsoft and optionally Python, GitLens, GitHub Pull Requests, and Issues, were installed through VS Code's Extensions;
- **Installing Compiler and Debugger**: For Windows, MinGW, which includes GCC and GDB, was installed. The MinGW bin directory was added to the system's PATH environment variable;
- **Configuring the Development Environment**: `tasks.json` and `launch.json` were correctly configured, ensuring the `miDebuggerPath` in `launch.json` was updated to the correct path of GDB on the machine, if different from the default;
- **Building and Running the Program**:
  - The program was compiled utilizing the default build task defined in `tasks.json`;
  - For debugging, the Debug view was switched to, the appropriate configuration selected from the dropdown menu;
- **Running Python Programs**: Python programs were run by setting the working directory to `ssmHW` and executing `python week1_WarmUp/ex1_SingleDieThrowing/src/simulate_dice.py` in terminal;
- **Continuous Integration**:
  - GitHub Actions for Continuous Integration was utilized, with workflow configurations found in `.github/workflows`, automatically running specified workflows for Python and C programs upon commits and pull requests;
  - Finally, the project was pushed to GitHub.

1 Single Die Throwing

In a single fair die throw, there are six outcomes (1 to 6), each with an equal probability of $\frac{1}{6}$. This is an example of a discrete uniform distribution. The mean expectation value is 3.5, indicating the mean roll value after many trials. The variance is approximately 2.9167, indicating the spread of outcomes around the mean. This scenario illustrates key concepts of probability, such as random variables, expectation, and variance, in a simple stochastic process.

1.1 Hypotheses and Tasks

(a) Show that the distribution is uniform $\left(\frac{1}{6}\right)$;

(b) Show and find how fast the variance goes to zero.

1.2 General Approach

Simulation employment confirms the equal likelihood of each outcome (1/6 for a fair die) and utilizes the Law of Large Numbers to illustrate the diminishing variance in the average outcomes with increasing dice numbers. The practical approach involves generating random dice outcomes, computing averages and variances, and analyzing these metrics' behavior as the simulation scale expands. This process is supported by plotting the results, which visually depicts the convergence towards theoretical expectations.

1.3 Source Files Documentation

`dice.h`:

- `simulateDiceThrows(size_t diceNr, size_t throwsNr)`: allocates memory for storing averages of dice outcomes, simulates dice throws using random number generation, calculates the average per throw, and returns the array. Outputs a pointer to an array of doubles, each representing the average outcome of `diceNr` dice over `throwsNr` throws;
- `calculateVariance(const double* data, size_t length)`: Computes the mean of the input array, then calculates the average of squared deviations from this mean, yielding the variance. Outputs the variance of the provided `data` set, of length `length`.

`dice.c`:

- contains the logic for generating dice outcomes and computing statistical measures;
- `simulateDiceThrow`: utilizes a loop to generate random numbers within the dice outcome range for each throw, accumulating and averaging these to populate an array;
- `calculateVariance`: iterates over the input array to first find the mean, followed by another iteration to calculate the mean of squared deviations, which determines the variance.

`main.c`: after gathering input on the number of dice and throws, it iterates over a range of dice numbers to simulate throws and compute variances for each case.
`simulate_dice.py`:

- `simulate_throws(throwsNr, diceNrLow, incrementsNr)`:
  - Generates a multi-dimensional array representing dice outcomes across increments, calculates probabilities for single-dice outcomes to verify uniformity, and computes variances for each dice count;
  - Inputs: `throwsNr` (number of throws), `diceNrLow` (starting number of dice), `incrementsNr` (number of dice increments);
  - Output: arrays of dice counts and corresponding variances;
- `plot_variance(dice_counts, variances)`: visualizes the relationship between the number of dice and variance, highlighting the rate at which variance diminishes as more dice are involved.

1.4 Conclusions and Results

The Python script was executed to analyze the uniform distribution of dice outcomes with 100000 throws and 50 increments with the command `python simulate_dice.py -throwsNr=100000 -incrementsNr=50`. The variance's exponential decay was observed through log-scaled plotting, indicating an exponential decay with a rate of 1/diceNr, as shown in Figure 1.

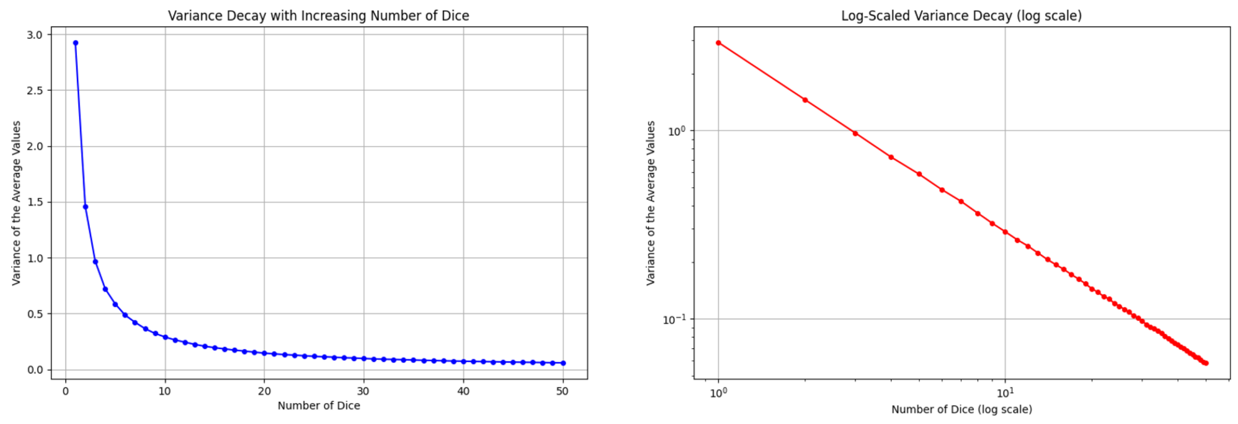

Figure 1: Variance of averages as a function of the number of dice.

The comparison between C and Python implementations highlighted C's detailed control over execution and memory for high-performance tasks, versus Python's convenience and rapid development capabilities with libraries like `NumPy` and `Matplotlib`.

2 Multiple Dice Throwing

This exercise explores the Central Limit Theorem (CLT) by simulating multiple dice throws, focusing on the total points obtained to observe the emergence of a normal distribution as the number of dice increases.

2.1 Hypotheses and Tasks

Demonstrate the Central Limit Theorem using the total points obtained when throwing simultaneously with several dice.

2.2 General Approach

The primary objective is to illustrate the CLT through practical simulation, tracking how the sum of dice outcomes approximates a normal distribution with an increasing count. The exercise employs both a C suite (comprising dice.c, dice.h, and main.c) for simulation and data generation, and Python scripts (plot_results.py and simulate_dice.py) for analysis and visualization. Key steps include initializing the random number generator, performing a specified number of experiments with varying dice counts, recording outcomes, and analyzing results through histograms to visualize the distribution pattern.

2.3 Source Files Documentation

`dice.h` and `dice.c`:

- `void seed_random_number_generator();` Initializes the random number generator using the current time, ensuring that each simulation run produces unique outcomes;
- `int throw_dice();` Simulates throwing a single dice, returning a random integer between 1 and 6. This function embodies the randomness of a dice throw;

- `void simulate_throws(int n_experiments, int dice_count, int* results);` Performs `n_experiments`, each involving the throw of `dice_count` dice. It calculates the sum of dice outcomes for each experiment and stores these sums in the `results` array, directly contributing to the empirical illustration of the CLT by aggregating outcomes.

`main.c`: This file acts as the entry point for the simulation, managing memory allocation for storing results, invoking the simulation function `simulate_throws`, and writing the outcomes to a file. It demonstrates handling large datasets and file I/O operations in C, optimized for performance and scalability.
`plot_results.py`: Reads the summed outcomes of dice throws from a file and plots a histogram to visualize the distribution of these sums, leveraging `matplotlib`. The histogram provides a graphical representation of the outcome distributions, serving as a visual confirmation of the CLT.
`simulate_dice.py` (Standalone):

- `np.random.randint(1, 7, size=(throwsNr, diceNr))` in `simulate_dice` function generates a matrix of random dice throws, which are summed to model the total outcomes per experiment. This approach simulates the statistical experiment of throwing multiple dice simultaneously;
- `plt.hist(sums, bins='auto', ...)` in `plot_histogram` visualizes the distribution of dice sums, providing insights into the distribution shape and its alignment with the predictions of the CLT;
- The main function configures the script to run with customizable parameters for the number of throws and dice, showcasing Python's adaptability for simulation and analysis with command-line arguments.

2.4 Conclusions and Results

The resultant plot supports the CLT. The histogram of summed dice outcomes 2 converges towards a normal distribution as the number of dice increases, visually affirming the Central Limit Theorem's predictions.

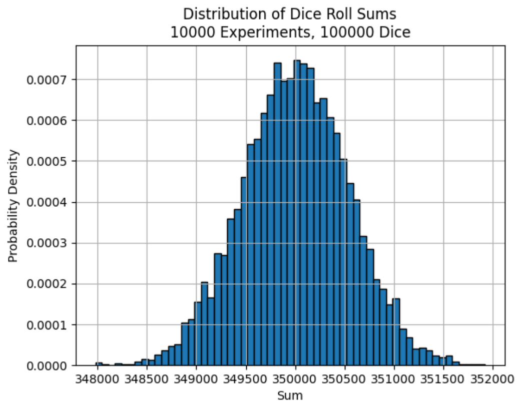

Figure 2: Variance of averages as a function of the number of dice.

The execution time difference between the Python (6 seconds) and C (3 seconds) scripts underscores the efficiency and performance optimization inherent to C, especially for computational tasks not involving graphical output. Conversely, Python, with its `NumPy` and `Matplotlib` libraries, excelled in rapid development and ease of data visualization, despite its longer execution time.

3 The Secretary Problem

We have to choose the best possible candidate from N applicants, but we can only interview each applicant once, if at all, and have to reject or employ them on the spot. The strategy involves declining the first k applicants regardless of their qualifications. Subsequently, the position is offered to the first candidate who surpasses the performance of these initial k applicants. Any remaining candidates will be dismissed without an interview. Should no subsequent applicants exceed the benchmark set by the initial k, we must to hire the last remaining applicant.

3.1 Hypotheses and Tasks

(a) What is the $\frac{k}{N}$ ratio that maximizes the chance that we employ the best applicant?

(b) Provide an analytical proof of the above result (bonus);

(c) What is the $\frac{k}{N}$ ratio that maximizes the average quality (rank) of the accepted applicant?

General Approach

This approach runs simulations to identify the best stopping rule for choosing a candidate in the Secretary Problem. It tests different selection points to find those that maximize the chance of selecting the top candidate or improve the average candidate quality. The method involves creating random candidate rankings, applying various selection thresholds, and analyzing the outcomes.

Source Files Documentation

`secretary.h` **and** `secretary.c` - Implement simulation logic for calculating optimal stopping points in the Secretary Problem:

- `void simulate_candidates(int N, int k)`: Executes the simulation by generating an array to represent candidate rankings from 1 to `N`, then shuffling this array for randomization. The shuffling uses a Fisher-Yates algorithm, enhanced by the randomness from `srand(time(NULL))`, to ensure each simulation run is unique. After observing the first `k` candidates, the function selects the next candidate better than those seen, aiming to simulate realistic decision-making dynamics.

- `void find_ratios()`: Executes the main simulation loop, systematically testing each value of `k` from 1 to `N` across `simulations` iterations. It calculates success rates (choosing the best candidate) and average ranks, dynamically updating `*optimal_ratio_best` and `*optimal_ratio_average` based on these outcomes. The loop employs a linear search strategy, minimizing computational complexity while maximizing statistical reliability and insight into the Secretary Problem's stochastic nature.

`main.c` - Orchestrates user input and simulation execution:

- Prompts for `N` and `simulations`, using standard I/O functions `printf` and `scanf` for interaction. This user-driven input allows customization of the simulation's scope and scale.

- Calls `find_ratios` with user-defined values, translating theoretical analysis into actionable simulation results, displayed via `printf`, providing immediate, comprehensible feedback on optimal selection strategies.

`plot_results.py` - Visualizes simulation outcomes with `matplotlib`:

- `read_results(filename)`: Extracts and organizes data from simulation output files using Python's file handling mechanisms. This function demonstrates data manipulation and preparation skills, crucial for accurate and insightful data visualization.

- `plot_graphs(success_file, total_rank_file)`: Generates two distinct plots to visualize the success rate and the average rank for various `k` thresholds. This function showcases effective use of `matplotlib`'s plotting capabilities, transforming raw data into meaningful graphs that highlight trends and outcomes crucial for decision-making in the Secretary Problem. The plotting parameters, such as `marker='.'` and `ms=8`, are chosen to enhance readability and interpretability of the plotted data.

3.2 Conclusions and Results

For the goal of maximizing the probability of selecting the best applicant, the `k/N` ratio remained relatively constant, with a success rate around 37% across various simulations. In contrast, the `k/N` ratio aimed at maximizing the average rank of selected candidates exhibited significant variability, heavily influenced by the number of simulations rather than the candidate pool size. The initial simulation parameters (`N`=1000, `simulations`=10000) yielded a suboptimal success rate of 32% for selecting the best applicant, with preliminary data suggesting an unexpectedly low `k/N` ratio for optimizing the average rank (accepting one of the first few candidates encountered).

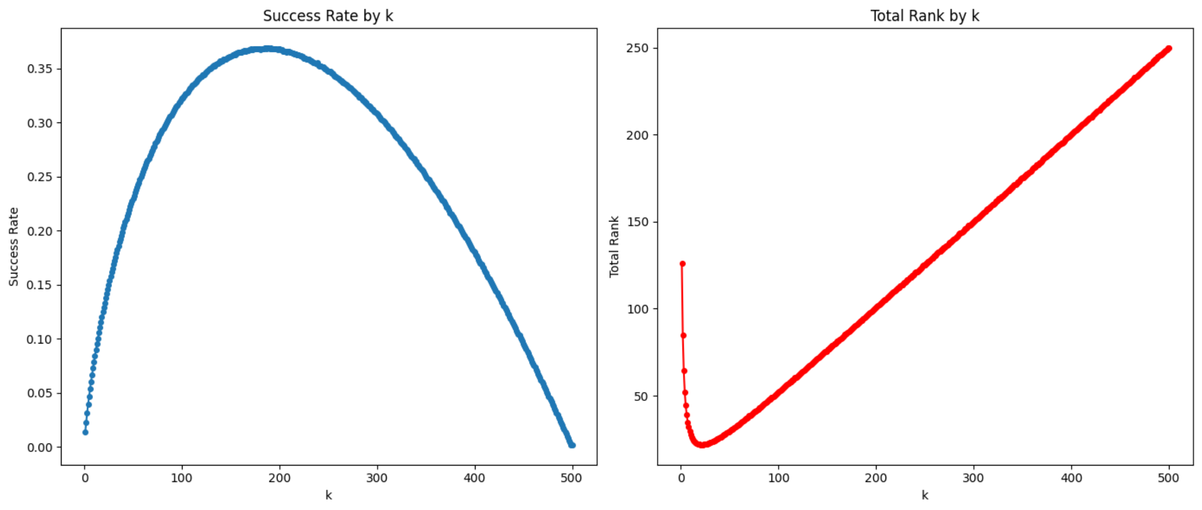

Figure 3:

Subsequent adjustments to the simulation parameters and code refinements led to improved performance metrics. A comprehensive simulation with `N`=1000 and `simulations`=2000000 (Figure 3) finalized the success rate for selecting the best applicant at 36.88%, aligning with the theoretical prediction of `k/N` $\approx e$ (`k/N` = 2.62). Moreover, a `k/N` ratio of 23.8 was identified for maximizing the average rank, translating to a strategy of rejecting the first 21 candidates and then selecting the next best. This approach ensures that the average hired candidate ranks within the top 4.362% of all candidates, validating the simulation's effectiveness in identifying optimal selection strategies. Due to the computational intensity of these simulations, the C programming language was chosen for its execution efficiency, relegating Python solely to the role of plotting the results.

4 Plain Random Walk

This investigation delves into the statistical properties of a plain random walk on one and two-dimensional lattices. The convergence of the one-dimensional walk to a Gaussian distribution underscores its adherence to the central limit theorem for large step numbers. In two dimensions, the study quantifies the walk's diffusive behavior via the mean squared displacement, $\langle d^2(N)\rangle$, on a square lattice and examines the effects of varying directional choices (including the four cardinal directions, four diagonals, all eight directions, and the inclusion of halting) on the walk's spatial distribution.

4.1 Hypotheses and Tasks

(a) Show that the probability distribution of the final position in one dimension is normal;

(b) Study the $\langle d^2(N)\rangle$ dependence for the plain random walk in two dimensions. Use a square lattice;

(c) Study the effect of the direction set (all directions have the same probability):

- four main directions (up, down, left, right) and halt;
- four diagonals (up-right, up-left, down-right and down-left);
- all eight directions;
- all eight directions and halt.

(a) Show that the distribution is uniform ($\frac{1}{6}$);

(b) Show and find how fast the variance goes to zero.

4.2 General Approach

For one-dimensional walks, binary steps (-1 or 1) are randomly generated for a specified number of steps and simulations, with the final positions computed by cumulatively summing these steps. These final positions are saved to plot the distribution's convergence to a Gaussian profile. In two dimensions, random walks are simulated for each set of directions by selecting steps from the predefined direction sets, cumulatively summing these to track the walker's path. The mean squared distance from the origin is calculated at each step, averaged across all simulations to quantify the walk's diffusive behavior. Each set of simulations generates data that is systematically saved, facilitating further analysis of the walks' statistical properties and their dependence on the chosen parameters.

4.3 Source Files Documentation

`init.h` and `init.c` - Initial setup for simulations:

- `void initialize_seed();`: Sets up the random number generator using the current time. This is crucial for ensuring that each simulation run produces unique outcomes by providing a different seed for the random number functions.

`simulation.h` and `simulation.c` - Core simulation functions for random walks:

- `static int random_step();`: Generates a random step for 1D walks. This helper function is critical for the simulation, deciding each move's direction by returning either -1 or 1.

- `void simulate_1d_walks(int steps, int num_simulations, const char* output_path);`: Conducts a series of 1D random walk simulations. For each simulation, it accumulates steps to calculate the final position and writes these positions to an output file, which is key for analyzing the distribution of final positions.

- `void simulate_2d_walks(int max_steps, int num_simulations, int (*directions)[2], int num_directions, const char* output_path);`: Performs 2D random walk simulations, utilizing a directions array to support various movement schemes. It tracks the squared distance from the origin to compute the average squared distance for a set number of steps, writing these to a file for later analysis. This function allows for exploration of different walking strategies by varying the direction set and step count.

`main.c` - Entry point that integrates the random walk simulations:

- Initializes the random seed to ensure varied simulation runs.

- Collects user inputs for simulation parameters, like the number of steps and simulations for both 1D and 2D random walks.

- Calls `simulate_1d_walks` and `simulate_2d_walks` functions with user-defined parameters and specific direction sets, directing output to respective files. This structured approach facilitates conducting and organizing multiple simulation scenarios.

`walk_plotting.py` - Visualization of random walk simulation results:

- `plot_1d_histogram(filepath);`: Creates and saves a histogram of the final positions from 1D random walk simulations. This visualization highlights the distribution of ending locations, providing insights into the randomness and tendency of the walk.
- `plot_2d_steps_vs_distance(filepaths, titles);`: Generates plots of average squared distance versus the number of steps for various 2D random walks. By comparing different movement strategies through graphical representation, this function elucidates the impact of direction choices on walk dispersion.

4.4 Conclusions and Results

In this computational study, one-dimensional random walk simulations were carried out using C, with parameters set to 50 steps and 10000000 simulations, to validate the emergence of a Gaussian distribution of final positions. This distribution is depicted in Figure 4, confirming the adherence to theoretical expectations.

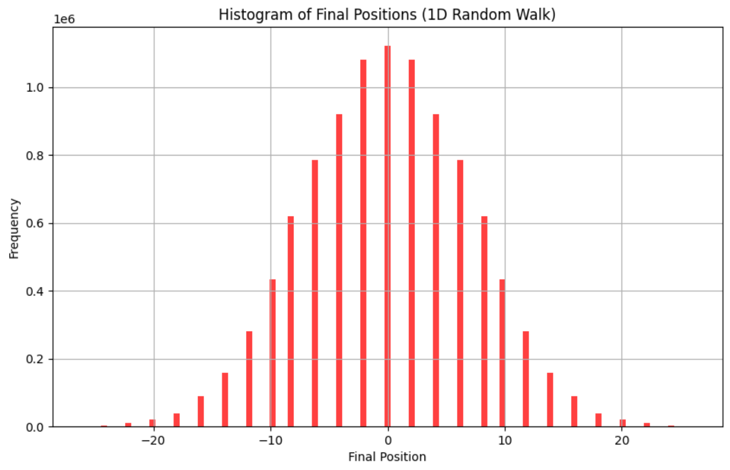

Figure 4: Histogram showing the distribution of thhe final position of a 50 steps simulation.

For two-dimensional simulations, parameters were expanded to 100 maximum steps and 1000000 simulations for each directional scenario. The analysis revealed (Figure 5) a direct proportionality between the mean squared displacement, $\langle d^2(N) \rangle$, and the number of steps, with distinct slopes of 0.8 for the four cardinal directions with halting, 2 for the four diagonals, 1.5 for all eight directions, and 1.3 for the eight directions with halting. These variations in slope underscore the impact of directional choices on the diffusive properties of the walk.

Parallel simulations executed in Python, aiming to replicate the conditions of the C simulations, successfully mirrored the results. Although the Python implementation was noted for its concise and efficient codebase, it was also marked by longer execution times in comparison to the C implementation.

Submitted by Victor-Ioan Macovei on March 25, 2024.

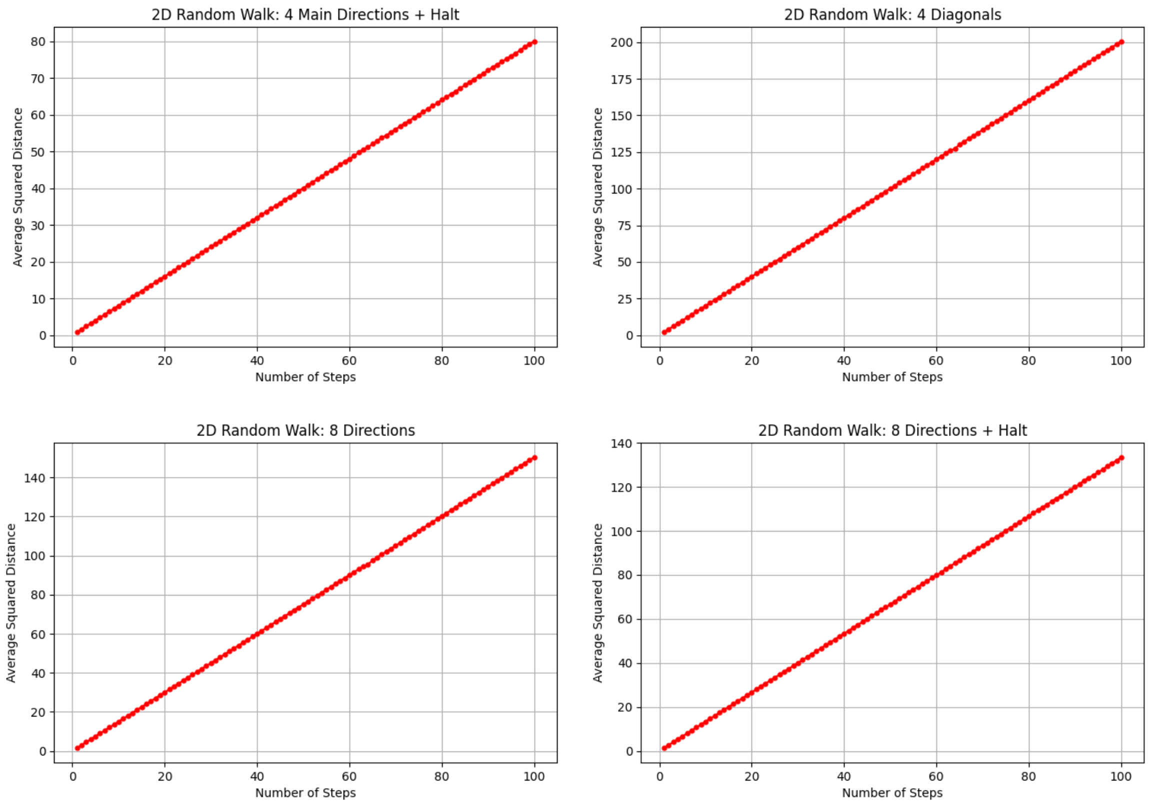

Figure 5: $\langle d^2(N) \rangle$ as a function of the number of steps for multiple direction sets.