Victor-Ioan Macovei
*Computational Physics MSc*
*Babeș-Bolyai University*

**July 9, 2024**

**HOMEWORK 2 — 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`;
  - Only one `main` file can be tested at a time. It's `isDefault` parameter in `tasks.json` must be set to `true`, while all the others are `false`;
  - 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 e.g. `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 Getting Lost

A random walk consists of a sequence of steps in a mathematical space, where each step is determined by a random process. The study of return probabilities—how often the random walk returns to its starting point—provides insights into the behavior of these processes over time and across dimensions. This exercise focuses on simulating one-dimensional, two-dimensional, and three-dimensional random walks to analyze the probability of returning to the origin after a given number of steps.

## 1.1 Hypotheses and Tasks

(a) Estimate and plot the probability distribution of the number of steps before returning "home";

(b) Prove that in 1D and 2D the probability is zero and in 3D it is 2/3.

### General Approach

The simulation process involves generating random walks for specified step counts and tracking the number of steps taken to return to the origin. The simulations are performed for multiple walks to ensure statistical significance. The data is then saved to files for each dimension (1D, 2D, 3D) and analyzed to calculate return probabilities. The analysis results are visualized using plots to illustrate the return probabilities as functions of the number of steps, leveraging both linear and logarithmic scales.

### Source Files Documentation

`main.cpp` - Entry point for running random walk simulations:

- **void initialize_simulation_parameters():**
  - Parameters: `int& max_step_count`, `int& step_increment_count`, `int& num_walks_sim`
  - Description: Sets the initial parameters for the simulations, including maximum step count, number of step increments, and number of simulations per increment. These parameters define the scope and granularity of the simulations.
- **void run_simulations():**
  - Parameters: `int max_step_count`, `int step_increment_count`, `int num_walks_sim`, `std::vector<std::vector<int»& return_steps_1D`, `std::vector<std::vector<int»& return_steps_2D`, `std::vector<std::vector<int»& return_steps_3D`, `std::vector<int>& step_counts`
  - Description: Conducts the random walk simulations for 1D, 2D, and 3D cases. It iterates over step increments, performs the walks, and records the return steps.
- **void save_data_to_file():**
  - Parameters: `const std::string& filename`, `const std::vector<std::vector<int»& data`, `const std::vector<int>& step_counts`
  - Description: Saves the simulation data to a specified file. It writes the number of steps and the return steps for each simulation, ensuring the results are stored for further analysis.

`random_walk.h` **and** `random_walk.cpp` - Implements the random walk logic:

- **void random_walk_1D():**
  - Parameters: `int steps`, `std::vector<int>& walk`, `std::mt19937& gen`
  - Description: Simulates a 1D random walk. It generates random steps, updating the position accordingly and recording the walk path.
- **void random_walk_2D():**
  - Parameters: `int steps`, `std::vector<std::pair<int, int»& walk`, `std::mt19937& gen`
  - Description: Simulates a 2D random walk. The walker moves in one of four directions, updating the coordinates and recording the path.
- **void random_walk_3D():**
  - Parameters: `int steps`, `std::vector<std::tuple<int, int, int»& walk`, `std::mt19937& gen`
  - Description: Simulates a 3D random walk. It involves movement in one of six directions, updating the coordinates in three dimensions.
- **int steps_to_return_1D():**
  - Parameters: `int steps`, `std::mt19937& gen`
  - Output: `int`
  - Description: Calculates the number of steps to return to the origin in 1D. It performs the walk and checks the return condition at each step.
- **int steps_to_return_2D():**
  - Parameters: `int steps`, `std::mt19937& gen`
  - Output: `int`
  - Description: Calculates the return steps for a 2D walk. The function tracks the position and checks when both coordinates return to zero.
- **int steps_to_return_3D():**
  - Parameters: `int steps`, `std::mt19937& gen`
  - Output: `int`
  - Description: Determines the steps required to return to the origin in 3D. It monitors the three coordinates and returns the step count upon return.

`analyze_results.py` - Analyzes and visualizes the random walk results:

- **def load_data():**
  - Parameters: `filename`
  - Description: Loads simulation data from a file. It reads the step counts and return steps, preparing the data for analysis.
- **def analyze_data():**
  - Parameters: `data`
  - Output: `list of tuples`
  - Description: Analyzes the loaded data to compute return probabilities. It calculates the proportion of walks that returned to the origin for each step count.
- **def save_analysis():**

- Parameters: `filename`, `results`
- Description: Saves the analyzed results to a file. It writes the step counts and corresponding return probabilities.

- **def plot_format():**
  - Parameters: `ax`
  - Description: Configures the plot formatting for consistency and clarity. It sets up visual parameters and styles.

- **def aggregate_data():**
  - Parameters: `results`, `bin_size`
  - Output: `list of tuples`
  - Description: Aggregates data into bins for smoothing. It averages the step counts and probabilities within each bin.

- **def plot_data():**
  - Parameters: `results`, `dimension`, `bin_size=10`, `log_log=False`
  - Description: Plots the return probabilities. It supports both linear and logarithmic scales, providing a clear visualization of the return probability distributions across different dimensions.

## 1.2 Conclusions and Results

The Python script was executed to analyze the distribution of the probability of returning home for different step numbers for all step numbers from 1 to 1000. More than 1000 steps was considered redundant based on visual inspection of the resulted data. The number of simulations for each step increment was chosen to be 100. This are assigned different seeds and averaged to obtain more relevant results.

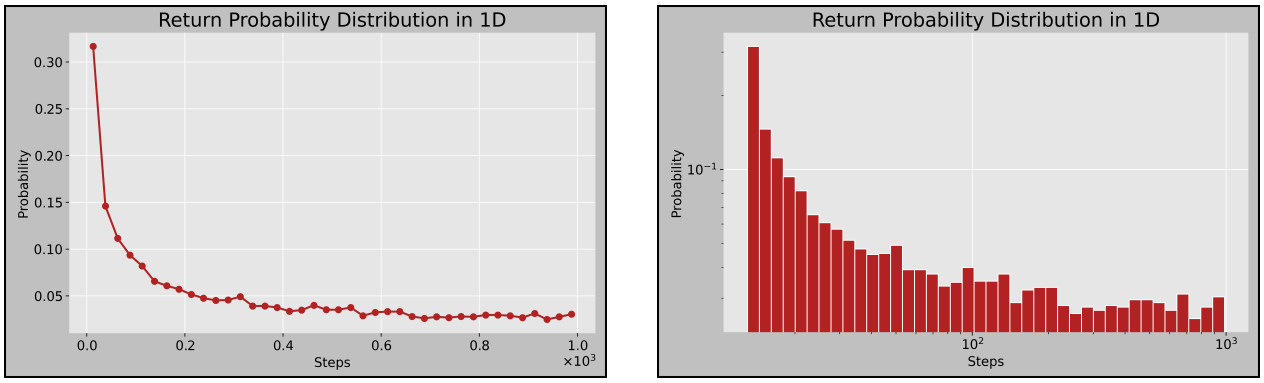


Figure 1: Scalar and log-log plots of 1D probability of getting lost.

The 2D results are not exactly as expected, as they seem to indicate a convergence, similar to the 3D case, which correctly stabilizes around 0.66. This may be to the simulation approach chosen, as instead of allowing the simulation to return to the origin every time and analyzing the distribution of the number of steps required, it (the simulation) was only allowed to run a pre-defined length which was incrementally increased. Further theoretical and experimental analysis are required.

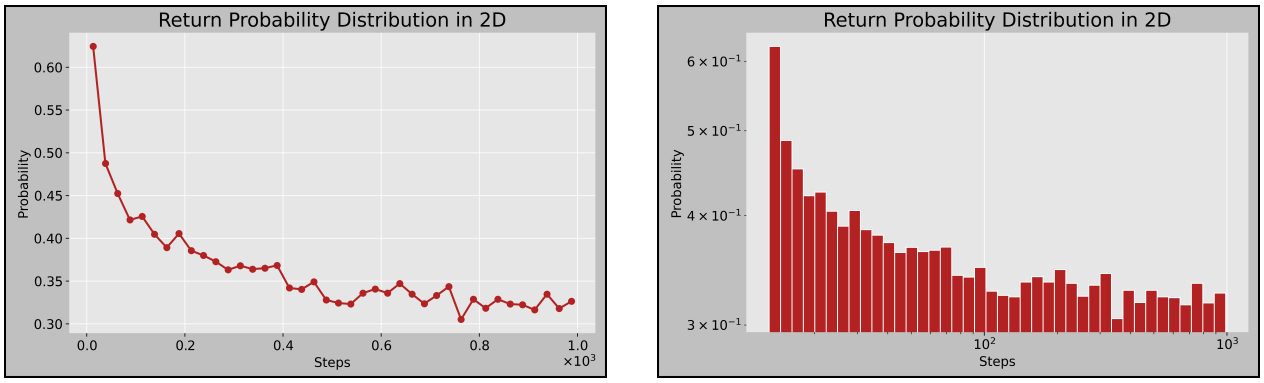


Figure 2: Scalar and log-log plots of 2D probability of getting lost.

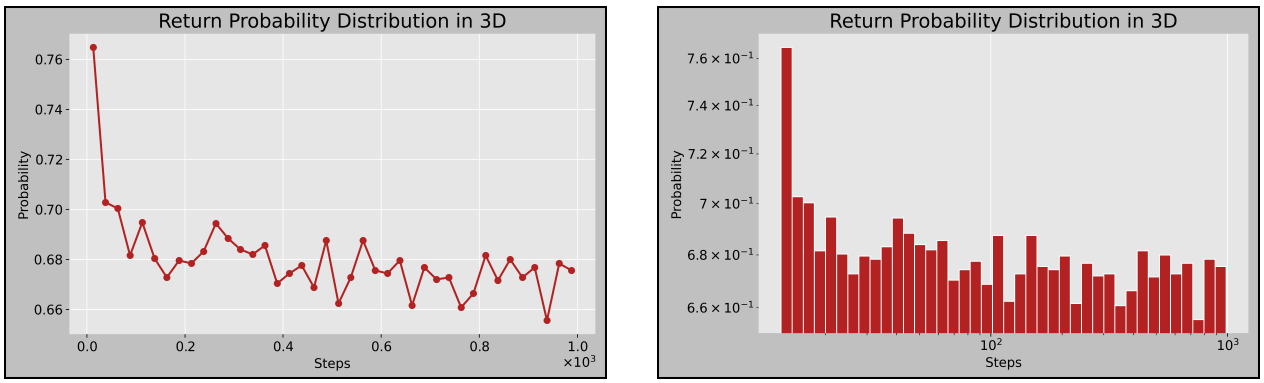


Figure 3: Scalar and log-log plots of 3D probability of getting lost.

# 2 Self Avoiding Walk

Self-avoiding walks (SAWs) are a class of random walks where the path does not intersect itself. These walks are significant in modeling polymer chains in chemistry and various phenomena in statistical mechanics. This exercise involves simulating self-avoiding walks in one and two dimensions, recording the steps before the walk gets blocked, and analyzing the walk paths to understand their properties and the probability of blocking.

## 2.1 Hypotheses and Tasks

Study the self-avoiding random walk in one and two dimensions (lattice sites can be visited only once):

(a) $\langle d^2(N) \rangle$ dependence;

(b) probability distribution of the number of steps before getting blocked;

(c) probability of getting blocked.

## General Approach

The approach involves simulating self-avoiding walks in 1D and 2D. The simulation tracks the steps until the walk gets blocked due to revisiting a previously occupied position. Each simulation's path and steps before blocking are recorded. The data is then analyzed to compute mean squared distances and the distribution of steps before blocking. The results are visualized using plots to provide insights into the behavior of self-avoiding walks.

## Source Files Documentation

`main.cpp` - Entry point for running self-avoiding walk simulations:

- **void main():**
  - Parameters: None
  - Description: Prompts the user for simulation parameters (dimensions, maximum steps, number of simulations). Runs the simulations, records the results, and saves them to a file.

`SelfAvoidingWalk.h` **and** `SelfAvoidingWalk.cpp` - Implements the self-avoiding walk logic:

- **SelfAvoidingWalk(int dimensions):**
  - Parameters: `int dimensions`
  - Description: Constructor that initializes the walk with the specified dimensions. Sets initial values for steps, blocked steps, and the current path.
- **void reset():**
  - Parameters: None
  - Description: Resets the walk to the initial state. Clears the path and visited sets, and reinitializes the starting position.
- **bool move():**
  - Parameters: None
  - Output: `bool`
  - Description: Attempts to move to a new position. If the new position has been visited before, the move is blocked. Otherwise, updates the path and visited sets.
- **std::pair<int, int> get_next_move() const:**
  - Parameters: None
  - Output: `std::pair<int, int>`
  - Description: Generates the next move based on the current position and allowed directions. Randomly selects a direction and returns the new position.
- **bool is_blocked() const:**
  - Parameters: None
  - Output: `bool`
  - Description: Checks if the walk is blocked by attempting all possible moves. Returns true if all moves lead to previously visited positions.
- **void run(int max_steps):**
  - Parameters: `int max_steps`
  - Description: Runs the self-avoiding walk for up to `max_steps`. Continues moving until the walk is blocked or the step limit is reached.
- **int get_steps_before_block() const:**
  - Parameters: None
  - Output: `int`
  - Description: Returns the number of steps taken before the walk gets blocked.

- **void save_results(const std::string &filename, const std::vector<int> &results) const:**
  - Parameters: `const std::string &filename`, `const std::vector<int> &results`
  - Description: Saves the results of the simulation to a file. Appends the results to the specified file.
- **const std::vector<std::pair<int, int» &get_path() const:**
  - Parameters: None
  - Output: `const std::vector<std::pair<int, int» &`
  - Description: Returns the path taken by the walk as a vector of coordinate pairs.

`process_output.py` - Analyzes and visualizes the self-avoiding walk results:

- **def parse_line():**
  - Parameters: `line`
  - Output: `list of tuples`
  - Description: Parses a line of text to extract coordinate pairs. Converts the extracted strings to integer tuples representing positions.
- **def plot_format():**
  - Parameters: `ax`
  - Description: Configures the plot formatting for consistency and clarity. Sets up visual parameters and styles.
- **def plot_results():**
  - Parameters: `filename`
  - Description: Reads simulation data from a file, calculates mean squared distances, and plots the results. Also plots the distribution of steps before blocking and prints the probability of getting blocked.

## 2.2 Conclusions and Results

The results are pretty clear in the sense that the number of steps until self intersecting distribution highly favors short distances in both 1D and 2D. The mean squared distance at the end of the path is higher in the 2D case than the 1D case, but they are both still very low, indicating that both paths terminate quickly. The probability of getting blocked across the 100000 simulated steps yielded 100% every time.

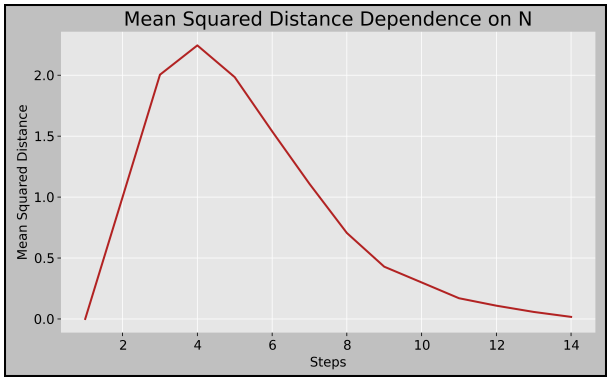


Figure 4: Mean squared distance of 1D self avoiding path.

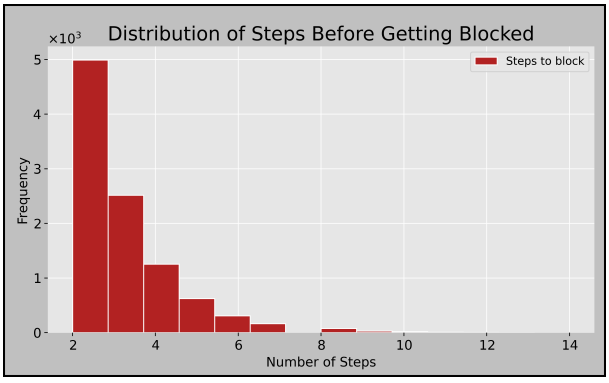


Figure 5: Steps until self intersecting in 1D paths.

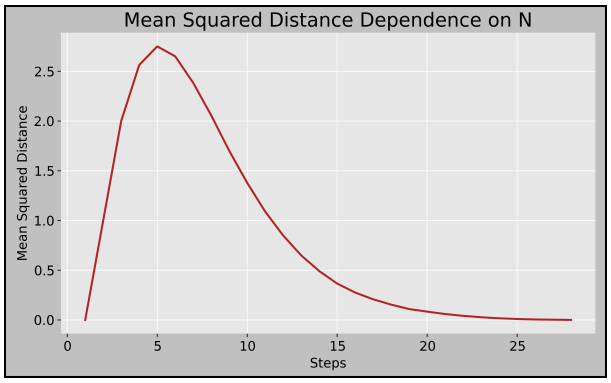


Figure 6: Mean squared distance of 1D self avoiding path.

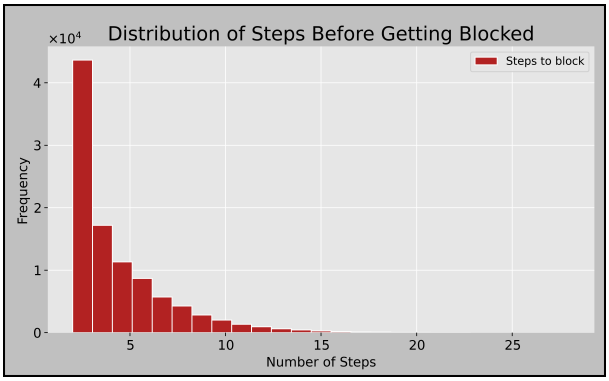


Figure 7: Steps until self intersecting in 2D paths.

# 3 Voronoi Tessellation

Voronoi tessellation divides a plane into regions based on distance to a specified set of points. Each region contains all points closer to one particular seed point than to any other. This exercise involves studying the cell size distribution in Voronoi tessellation. In 1D, the distribution is expected to follow the form $xe^{-x}$, while in 2D, it is expected to follow $x^{5/2}e^{-x}$. These distributions provide insights into the geometric properties of Voronoi cells.

## 3.1 Hypotheses and Tasks

Study the cell size distribution for the Voronoi tessellation:

(a) 1D: show that it is of $x \cdot e^{-x}$ type;

(b) 2D: show that it is of $x^{\frac{5}{2}} \cdot e^{-x} type$;

## 3.2 General Approach

The approach involves generating random points and computing the Voronoi tessellation for these points in 1D and 2D. The cell sizes are then calculated, aggregated, and analyzed to fit the expected distributions. The results are visualized using histograms and compared against the theoretical distributions to validate the expected forms. The analysis also includes eliminating outliers to ensure the accuracy of the fit.

## 3.3 Source Files Documentation

`voronoi_analysis.py` - Simulates Voronoi tessellation and analyzes cell size distribution:

- **class VoronoiSimulation:**
  - Parameters: `num_points`, `canvas_size`, `num_bins`, `num_simulations`, `max_cell_size_1D`, `max_cell_size_2D`
  - Description: Initializes the simulation with the given parameters, sets up arrays for storing points and cell sizes, and defines bin edges for histograms.
- **def generate_points():**
  - Parameters: None
  - Description: Generates random points for 1D and 2D simulations. Uses a fixed seed for each simulation to ensure reproducibility.
- **def compute_1D_voronoi():**

- Parameters: None
- Description: Computes Voronoi tessellation in 1D by sorting the points and calculating the intervals between consecutive points.

- **def compute_2D_voronoi():**
  - Parameters: None
  - Description: Computes Voronoi tessellation in 2D using the `scipy.spatial.Voronoi` function. Calculates the area of each Voronoi cell.

- **def compute_polygon_area():**
  - Parameters: `vertices`
  - Output: `float`
  - Description: Static method that computes the area of a polygon given its vertices. Uses the shoelace formula for the calculation.

- **def eliminate_outliers():**
  - Parameters: None
  - Description: Eliminates outlier cell sizes that exceed the maximum specified cell size. Adjusts the maximum cell size based on actual simulation data.

- **def aggregate_distributions():**
  - Parameters: None
  - Description: Aggregates the cell size distributions into histograms. Sums the histograms across all simulations and normalizes by the number of simulations.

- **def normalize_distributions():**
  - Parameters: None
  - Description: Normalizes the aggregated histograms to ensure they represent probability distributions.

- **def fit_distributions():**
  - Parameters: None
  - Description: Fits the aggregated cell size distributions to the expected theoretical forms using non-linear curve fitting. Extracts the scale and width parameters for the fits.

- **def plot_histograms():**
  - Parameters: None
  - Description: Plots the normalized histograms for 1D and 2D Voronoi cell size distributions. Overlays the fitted theoretical distributions for comparison.

- **def plot_voronoi_diagram():**
  - Parameters: None
  - Description: Plots a Voronoi diagram for a single 2D simulation. Colors the cells distinctly and visualizes the tessellation structure.

### 3.4 Conclusions and Results

The simulations were executed on 100 points. The 100 simulations were aggregated and plotted in 9. Although the bin values were normalized and the shapes of the plots and the expected lines are similar, an exact match was not achieved.
*Submitted by Victor-Ioan Macovei on July 9, 2024.*

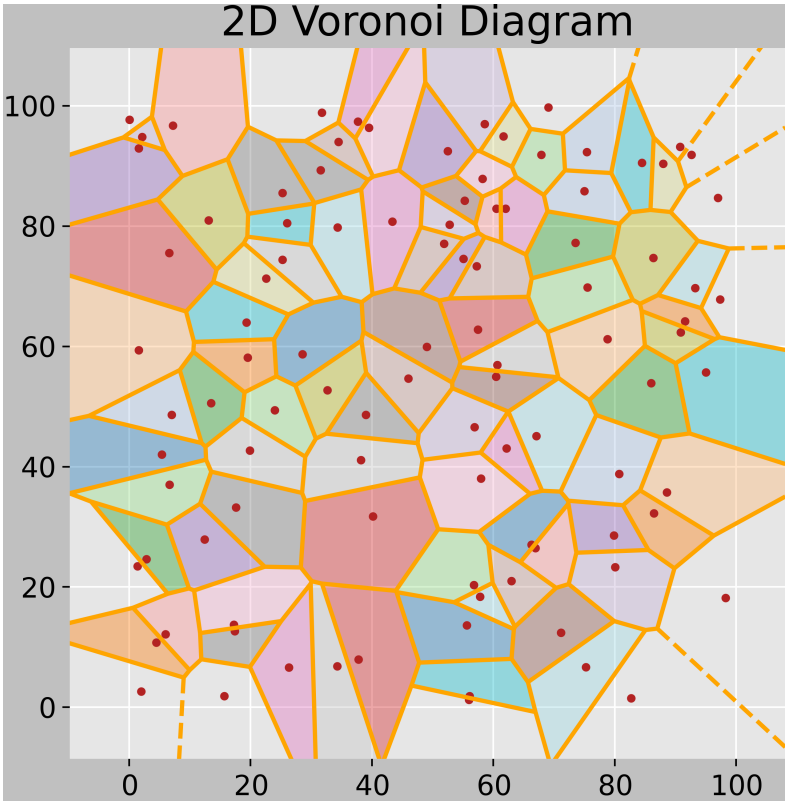


Figure 8: Voronoi diagram for an arbitrary 2D simulation.

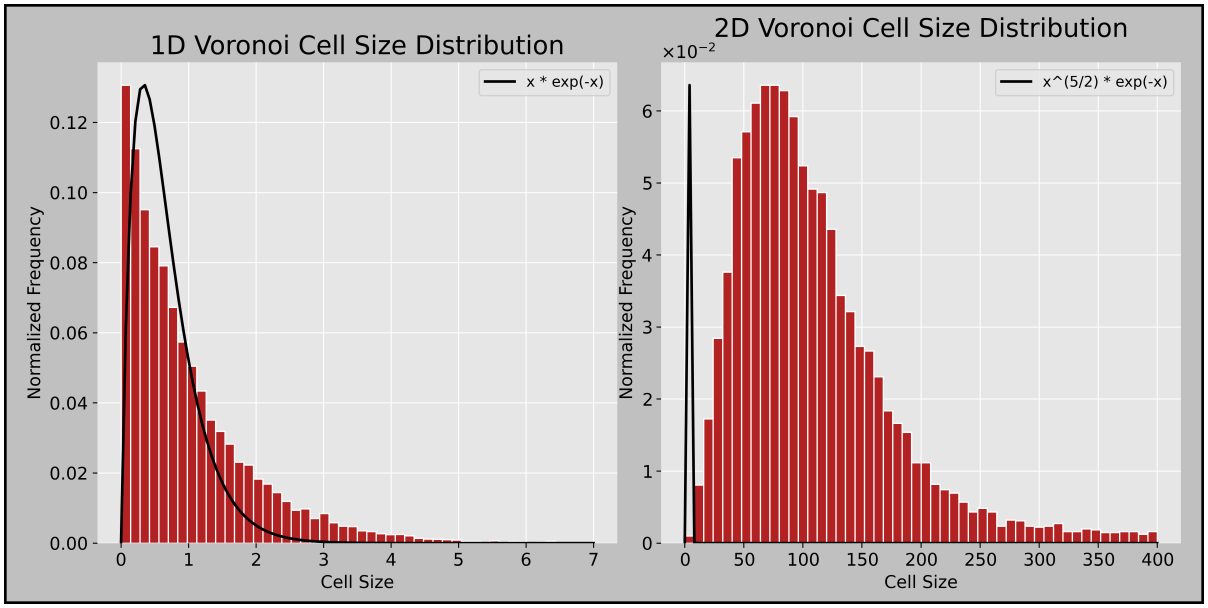


Figure 9: Histograms for 1D and 2D simulations.