Yolia Simon - Assignment report (Option 4: Simulation of a random walk):

This project simulates a random walk simulation in C for a particle in one, two, or three dimensions. The simulation allows users to configure various parameters, including the number of steps, lattice spacing (Δx), time step (Δt), and diffusion constant (D). The program supports self-avoiding random walks in 2D and 3D, where the particle cannot revisit previously occupied positions. Results are saved in a CSV format for visualization using Python or other plotting tools. The program also asks the user to

Header files:

To start, I used 4 header files including <stdio.h>. I used <std.lib.h> because it provides functions for memory management, program control and random number generation. I used <stdbool.h> to introduce the bool data type and true/false constants to my program. This improves readability and enables logical conditions like the self-avoidance logic that is found later on in the program. I used <string.h> to process user input and filenames (which are strings) in my program.

Functions:

Next, I declared 5 prototype functions outside of the main program. These functions are later defined at the bottom of the outside of the main program:

1. void random_walk_1D(int steps, double dx, double dt, double D, const char *filename)
2. void random_walk_2D(int steps, double dx, double dt, double D, const char *filename, bool self_avoiding)
3. void random_walk_3D(int steps, double dx, double dt, double D, const char *filename, bool self_avoiding)
4. bool is_visited(int x, int y, int z, int **visited, int visited_count)
5. void print_py_code(int dimension)

For 4 of these functions, I used the 'void' data type because those functions don't need to return a value. These functions perform their tasks but don't send any values back to the calling code. For one of the functions, I used the 'bool' data type to represent and handle logical conditions in a clear and intuitive way all while improving the code readability.

1. The first function defined is the 1D random walk function which simulates a random walk in one dimension. The particle can move left, right, or remain stationary. It takes in parameters for the number of steps, lattice spacing (dx), time step (dt), diffusion constant (D) and the filename. I used the 'int' data type for the number of steps as it is a discrete value. I used the 'double' data type for dx, dt, and D to allow for precise calculations of probabilities and to handle larger, fractional values that 'int' or 'float' types would not be able to store without overflowing. The 'char*filename' means that 'filename' is a pointer that points to the memory location where the string (which represents the filename) is stored. I added the 'const' data type to this to ensure that the string pointed to by 'filename' cannot be modified by the function which adds a layer of safety to the code. The function starts by opening a file in write mode. It takes the variable 'filename' as the name of the file. I used an 'if' statement to print an error message and terminate the program if the file cannot be opened. I used 'fprintf' to write the column headers to the file which makes the file easy to interpret when visualizing or analysing the data. Using the 'int' data type, I then initialised a variable 'position' to 0 to track the particle's current position which starts at zero. I initialised 'p' which is the probability of moving left or right based on the diffusion equation:
$$p = \frac{D \cdot \Delta t}{\Delta x^2}$$
The variable 'p' is calculated as a 'double' data type for precise control over movement probabilities. After this, I used a 'for' loop to iterate over the number of steps while the integer *i* is less than the number of steps. The increment operator '++' is used in the 'for' loop to increase by 1. Inside the 'for' loop, I initialised the variable 'r' which generates a random number to decide whether the particle should move right, left or stay in place. The 'if/else if' statement updates the position of the particle. The current step index *i* and the particle's position is then written to the file using the decimal format specifier '%d'. Once the loop is complete, the file is closed to ensure that the data is saved properly and to avoid data corruption.

2. The second function defined is the 2D random walk function which simulates a random walk in two dimensions with an option for self-avoidance. In a 1D random walk, self-avoidance is unnecessary because there is only one possible direction for the particle to move: forward or backward along the same axis. Therefore, self-avoidance options have only been made available for 2D and 3D random walks. This function takes in the same parameters as the 1D random walk function as well as an additional 'self_avoiding' variable. I used a 'bool' data type for this variable to determine if the particle avoids revisiting positions. This function starts in the same way as the 1D random walk function by

opening the file in write mode, including an error check and writing column headers to the file. I initialised the variables 'x' and 'y' to 0 to track the particle's current position which starts at the origin (0,0). The variable 'visited_count' tracks the number of unique positions visited. '**visited' is a double pointer which is used to represent an allocated 2D array where each element of the first pointer points to the allocated 1D array of coordinates. It is initialised to 'NULL' as a safety measure to ensure that the pointer doesn't point to garbage memory address. I used the 'int' data types for these variables because they are discrete values. Then I used an 'if' statement with a 'for' loop inside it to allocate memory to every row if self-avoidance is enabled. The 'if' statement also includes input validation where if 'malloc' fails (for example, due to a lack of memory) it returns as 'NULL'. If 'visited' is 'NULL', the program will print an error message using the 'perror' feature, close the file and exit the entire program. After this is checked, the function tracks the starting position (0, 0) by using the increment operator '++'. Similarly to the 1D random walk function, I initialised the probability 'p' and used a 'for' loop to iterate over the number of steps. Inside this 'for' loop, I initialised the variable 'r' which does the same thing it does in the 1D random walk function except that as well as determining whether the particle should move left or right, it also determines whether it should move up or down. I used the 'int' data type for the variables 'dx' and 'dy' because they are discrete values. I initialised these variables to 0 and they are updated based on the chosen direction in each iteration using the add AND assignment operator '+='. I used an 'if' statement to record the particle's position to the 'visited' variable only when 'self_avoiding' is true. Then, I used 'fprintf' to write the current step index *i* and the particle's x and y positions to the file using decimal format specifiers '%d'. After this, I used an 'if' statement which iterates over the number of steps and frees the allocated memory for the 'visited' array after completing the simulation. Finally, the file is closed to ensure the data is saved and to avoid data corruption.

3. The third function defined is the 3D random walk function which simulates a random walk in three dimensions with an option for self-avoidance. This function takes in the same parameters as the 2D random walk function. This function works in the exact same way as the 2D random walk function with the addition of the z- axis. In this function, the variable 'r' determines whether the particle should move along the x-axis, y-axis or z-axis. The function also uses an allocated 3D array (int **visited) for self-avoiding walks.

4. The fourth function defined is the 'is_visited' function which determines if a specific lattice point (x, y, z) has been already visited during a self-avoiding walk. This function takes in parameters for x, y, z (depending on the dimension requested can be ignored), '**visited' and 'visited_count'. I used the 'int' data type because the values are discrete. '** visited ' is an allocated 2D array that records the coordinates of all the previously visited positions while 'visited_count' is the number of positions currently stored in the 'visited' array. The function starts with a 'for' loop that iterates over all of the rows in the 'visited' array up to 'visited_count' increasing by 1 each time by using the increment operator '++'. I used an 'if' statement inside the 'for' loop to compare the input coordinates with the coordinates that are stored in the current row of the 'visited' array ('visited[i]'); if the x, y and z coordinates match, the position is considered as visited and the function will return the bool 'true'. If a match is not found, the loop continues and returns the bool 'false' after the loop is completed without finding a match which indicates that the position has not been visited yet.

5. The final function defined is the 'print_py_code' function which generates a Python script for visualizing the simulation results. The function takes in the parameter 'dimension' which determines the dimensionality of the simulation (1D, 2D, or 3D) to generate the appropriate script. I used the 'int' daa type to store the dimensionality as a whole number (1, 2 or 3). The function starts by opening a .txt file named 'visualisation_python_code' in write mode. 'fopen' is checked in case it returns 'NULL' and an error message is displayed if the file cannot be opened correctly. I used a 'switch' statement to determine the dimension of the random walk and write the corresponding Python visualization code to the file. The 'default' case handles unexpected values for the variable 'dimension' by printing an error message and exiting the entire program. The file is then closed to prevent data corruption and clear instructions on how to use the Python code are printed to the console using 'printf'.

Main program:

To start with, I declared the variables 'steps' and 'dimension' using an 'int' data type as these values are discrete. I declared the variables 'dx', 'dt' and 'D' using the 'double' data type to allow for fractional values. I initialised 'filename', 'mode' and 'choice' as arrays to store user responses using the 'char' data type. I initialised the variable 'self_avoiding' as 'false' using the

‘bool’ data type to simplify conditional checks and improve code readbility. Finally, I initialised the variable ‘valid_filename’ as ‘false’ using the ‘bool’ data type to check if the filename is valid.

Using ‘printf’, the console displays a description of the program to the user which outlines its functionality. Then, I used ‘printf’ to ask the user to enter each input needed to simulate the random walk (number of steps, lattice spacing, time step, diffusion constant and dimension). For each input, I used an ‘do while’ loop to validate the user input. The ‘do…while’ loop ensures that if the wrong input is entered, the program will prompt them to re-enter until they provide a valid positive integer. The ‘if..while’ loop also activates which displays an error message and deletes the incorrect input from the system. After the user enters an input for the dimension, if the dimension is greater than 1 (either 2 or 3), the user will then be asked if they would like to enable a self-avoiding random walk. By making this option available only when the dimension is greater than 1, it ensures that unnecessary memory isn’t allocated if the 1D random walk is chosen. I then used a ‘while’ loop to ensure that when the user inputs the name of their data file, it includes ‘.csv’ and it is at least more than 4 characters long to accommodate ‘.csv’ on the end of the string. I used ‘printf’ to display a message that indicates that the simulation is currently running.

The next section in the program runs the simulation based on the dimension the user has asked for. I used a ‘switch’ statement to determine the dimension and call the corresponding function to complete its task. Once the function completes its task, ‘break’ exits the loop. ‘printf’ then prints a message to the console indicating that the simulation is complete and the output file has been generated successfully.

The final section of the program asks if the user wants a Python script for visualization. If the user responds with "yes," the program calls print_py_code to generate a script corresponding to the chosen dimension. If the user responds with “no”, the loop breaks and continues on. I used a ‘while’ loop to rejects invalid inputs and loops until a valid response is given. Finally, a farewell message is printed to the console, signalling the end of the program.

Memory requirements:

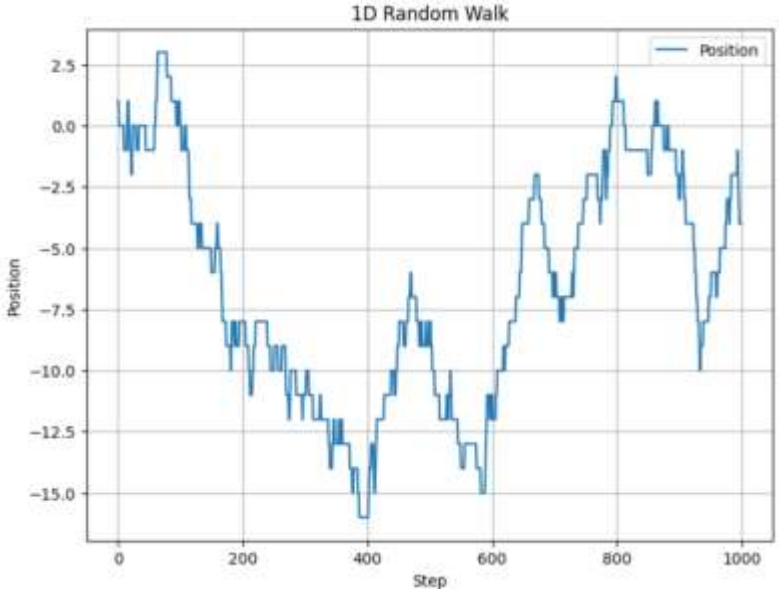
The program's memory requirements depend significantly on the type of random walk being simulated and whether self-avoidance is enabled. For non-self-avoiding walks, memory usage is minimal, as the program only needs to store the particle's current position and write the results directly to the CSV file. In these cases, the memory usage is constant, regardless of the number of steps.

For self-avoiding random walks, the program allocates memory to store previously visited lattice points. This memory usage increases with the number of steps as each step can add a new position to the visited list. In 2D, each position requires 2×sizeof(int) bytes, while in 3D, it requires 3×sizeof(int). Therefore, memory requirements can increase quickly for multiple steps, especially in 3D. Additionally, maintaining an extensive 'visited' array increases the possibility of memory fragmentation and allocation failures, particularly on systems that have limited RAM.

Limitations and Improvements that could be made: One limitation of the program is its scalability, especially for self-avoiding random walks where the linear search through the visited list for each step results in a great increase in the runtime along with the number of steps. This makes large-scale simulations impractical. Additionally, the 'rand()' function for generating random numbers could introduce biases, as it might not be suitable for high-quality randomness in large simulations. Another limitation is the lack of direct integration with visualization tools such as Python, as users must manually run the generated Python scripts from .txt files for analysing results. This can be time-consuming and inefficient. To improve my program, I could replace the linear search with a more efficient data structure, such as a hash table, to optimize visited-position checks for self-avoiding walks and reduce runtime complexity. For non-self-avoiding walks, the memory allocation could be further optimized to avoid allocating unnecessary arrays like 'visited' when self-avoidance is not enabled. Finally, I could find a way to integrate Python-based visualization directly into the program so it can streamline the workflow and make the tool more efficient and user-friendly for visually presenting and analysing data.

Test parameters:

Using my code, I have produced these test graphs for each random walk case using the following parameters for each.



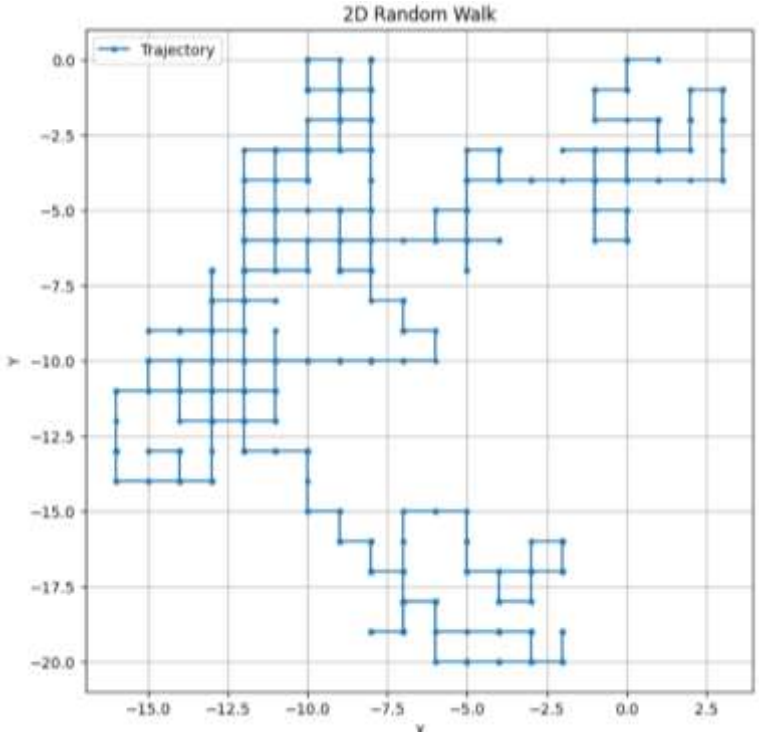
1D Random Walk

Steps: 1000

Lattice Spacing (Δx) 1.0

Time Step (Δt): 0.1

Diffusion Constant (D): 1.0



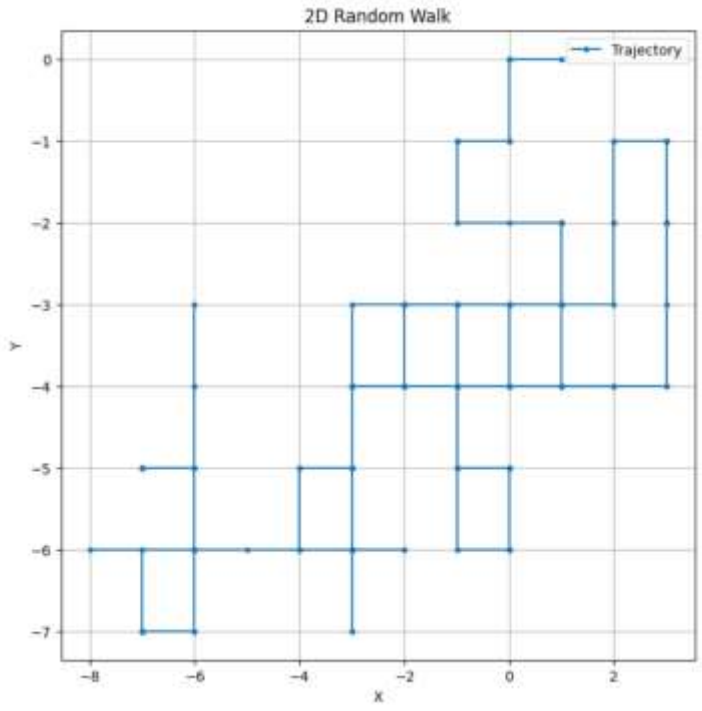
2D Random Walk (Non-Self-Avoiding)

Steps: 1000

Lattice Spacing (Δx) 1.0

Time Step (Δt): 0.1

Diffusion Constant (D): 1.0



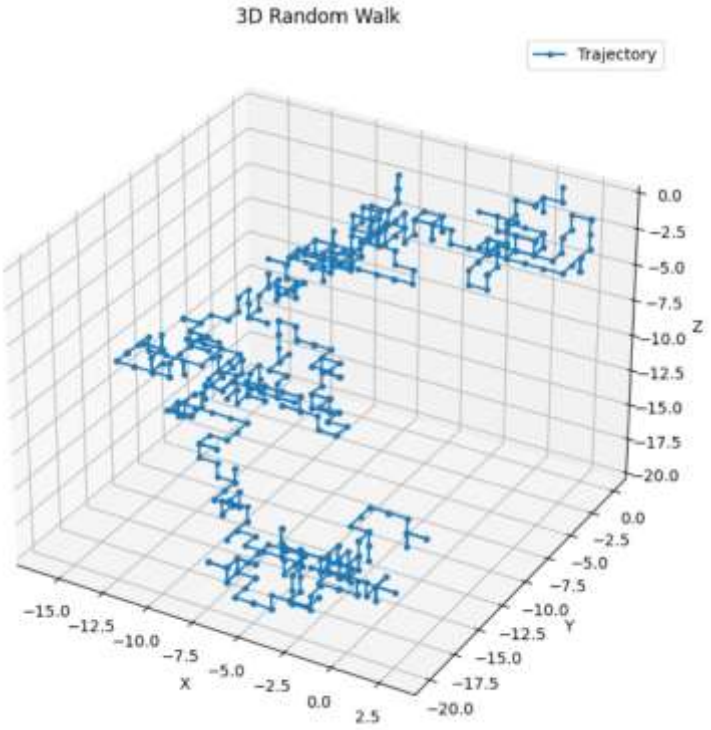
2D Random Walk (Self-Avoiding)

Steps: 200

Lattice Spacing (Δx) 1.0

Time Step (Δt): 0.1

Diffusion Constant (D): 1.0



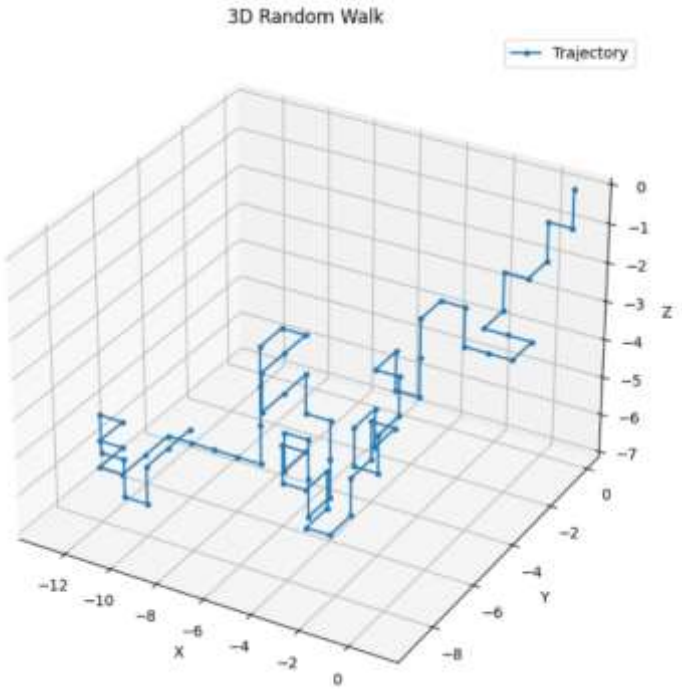
3D Random Walk (Non-Self-Avoiding)

Steps: 1000

Lattice Spacing (Δx) 1.0

Time Step (Δt): 0.1

Diffusion Constant (D): 1.0



3D Random Walk (Self-Avoiding)

Steps: 200

Lattice Spacing (Δx) 1.0

Time Step (Δt): 0.1

Diffusion Constant (D): 1.0

Results and Observations:

In the 1D random walk, the particle alternates between moving left or right, producing a zig zag trajectory. In the 2D case, non-self-avoiding walks show random paths that occasionally revisit positions, forming dense clusters while self-avoiding walks terminate early if the particle becomes trapped in a dead end. In the 3D case, the particle explores 3D space more freely than in 2D, with self-avoiding constraints reducing the likelihood of trapping.

(a) Dependency of Average Squared Position and Absolute Position in 1D

To analyse the relationship between the average squared position (< x^2>) and the average absolute position (<|x|>) with respect to simulation time (T) and diffusion constant (D) in a one-dimensional random walk, the program was run with various values for T, D and Delta t. The particle's trajectory was simulated over many independent runs to obtain statistically

significant results. The final position at each time step was recorded, and the squared position and absolute position were averaged across all runs.

The results showed that $< x^2 >$ grows linearly with T and is directly proportional to D, confirming the theoretical relationship $<x^2>$ = 2DT. This dependency arises because T determines the total number of steps, and D, through the diffusion equation, directly affects the likelihood of movement. On the other hand, <|x|> increases approximately with $\sqrt{T}$, as <|x|> scales with the square root of the mean squared displacement. This finding supports the theoretical relationship <|x|> $\propto \sqrt{DT}$.

(b) Extension to 2D and 3D

The results for $<x^2>$ and <|x|> extend naturally to higher dimensions. For two-dimensional random walks, $<r^2>$ = $<x^2 + y^2>$ grows as 4DT, reflecting the contribution of diffusion in both the x and y directions. Similarly, <|r|> = $<\sqrt{x^2 + y^2}>$ scales with $\sqrt{DT}$, but the geometric averaging over two dimensions slightly reduces the magnitude compared to the 1D case. For three-dimensional random walks, $<r^2>$ grows as 6DT, consistent with the addition of the z dimension. The scaling for <|r|> remains $\sqrt{DT}$, adjusted for three spatial dimensions.

For self-avoiding random walks in 2D and 3D, the relationship between $<r^2>$ and T deviates from linearity, as the particle becomes increasingly restricted in its movement. Over long timescales, the growth of $<r^2>$ slows significantly compared to non-self-avoiding walks, leading to sub-diffusive behavior. This is because the particle's constrained motion results in fewer accessible lattice points, especially in 2D, where the probability of trapping is higher.

(c) Non-Symmetric Random Walks

In non-symmetric random walks, the probability of moving in one direction is greater than that of moving in the opposite direction. For example, if the probability of moving right is p_+ = 0.6 and moving left is p_p_- = 0.4, the particle exhibits a biased trajectory. Running the simulation under such conditions reveals that $<x^2>$ still grows linearly with T, as the random nature of the steps ensures that fluctuations in position are proportional to time. However, the average absolute position <|x| > becomes dominated by the bias, leading to an approximately linear growth with T rather than the square-root scaling observed in symmetric walks.

In higher dimensions, introducing bias along one axis produces anisotropic diffusion, where the particle's displacement is larger in the biased direction compared to the others. For example, in a 2D random walk with $p^+{}_x$ = 0.6 and $p^-{}_x$ = 0.4, while $p^+{}_y$ = $p^-{}_y$ = 0.5, the mean squared displacement along x, $<x^2>$, grows more quickly than $<y^2>$. This directional preference introduces asymmetry in the trajectory, which becomes evident in trajectory plots and quantitative metrics.

Overall, these observations demonstrate how the properties of random walks vary across dimensions, constraints, and biases. The program's ability to handle large-scale simulations and generate CSV outputs enables detailed analysis and visualization of these phenomena.

References

- Berg HC. *Random walks in biology*. Princeton: Princeton University Press; 1993.
- Weiss GH. *Aspects and applications of the random walk*. Amsterdam: North-Holland Publishing; 1994.
- Spitzer F. *Principles of random walk*. 2nd ed. New York: Springer; 2001.

- Metzler R, Klafter J. The random walk's guide to anomalous diffusion: a fractional dynamics approach. *Phys Rep*. 2000;339(1):1-77.
- Redner S. *A guide to first-passage processes*. Cambridge: Cambridge University Press; 2001.
- Codling EA, Plank MJ, Benhamou S. Random walk models in biology. *J R Soc Interface*. 2008;5(25):813-34.
- Montroll EW, Weiss GH. Random walks on lattices. II. *J Math Phys*. 1965;6(2):167-81.
- Pearson K. The problem of the random walk. *Nature*. 1905;72(1865):294.
- Weisstein EW. Random Walk--1-Dimensional [Internet]. mathworld.wolfram.com. Available from: https://mathworld.wolfram.com/RandomWalk1-Dimensional.html