1 Deriving σ^2

From the book we have:

$$x_{(t)} = x_{(t-\tau)} + al$$

therefore:

$$< x_{(t)}^2 >=< x_{(t-\tau)}^2 > + < a^2 > l^2 + 2l < ax_{(t-\tau)} >$$

where:

$$< a^2 >= p < (1)^2 > +q < (-1)^2 >= p + q = 1$$

$$< ax_{(t-\tau)} >= (p-q) < x_{(t-\tau)} >= \frac{l}{\tau}(p-q)^2(t-\tau)$$

therefore by using the above:

$$< x_{(t)}^2 >=< x_{(t-\tau)}^2 > +l^2+\frac{2l^2}{\tau}(p-q)^2(t-\tau) =< x_{(0)}^2 > + \sum_{n=0}^{N-1}(l^2+\frac{2l^2}{\tau}(p-q)^2 n\tau)$$

$$< x_{(t)}^2 >= Nl^2 + 2l^2(p-q)^2\sum_{n=0}^{N-1} n = Nl^2 + l^2(p-q)^2(N-1)N$$

therefore using the definition of deviation:

$$\sigma^2 =< x_{(t)}^2 > - < x_{(t)} >^2= Nl^2((p+q)^2 - (p-q)^2) = 4\frac{l^2}{\tau}pqt$$

2 1D Random Walk

The code used to simulate the one dimensional random walk is quite simple and uses only one function.

2.1 randwalk() function

This function takes The probability of going right(P) and the number of steps as input, then returns two arrays representing the position of the walker and the time. Each step is randomized with regards to the probability. The resulting walk looks like this:

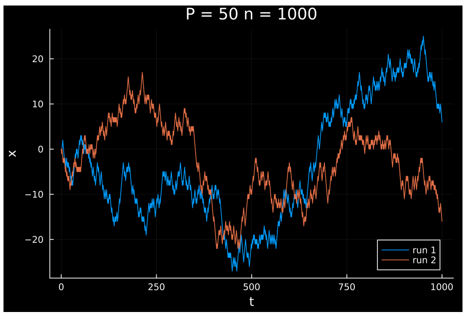

Figure 1: Position based on time for 1D random walk.

2.2 Averaging Loop

We use a loop to run the random walk function multiple times and average over the results. We then use the data to calculate the mean and deviation of the position. We also calculate the theoretical amounts of $< x >$ and σ^2 using the equations below:

$$< x_{(t)} > = \frac{l}{\tau}(p-q)t$$

$$\sigma^2 = \frac{4l^2}{\tau}pqt$$

2.3 Results

Both the simulated and theoretical results are plotted in one graph The fits are

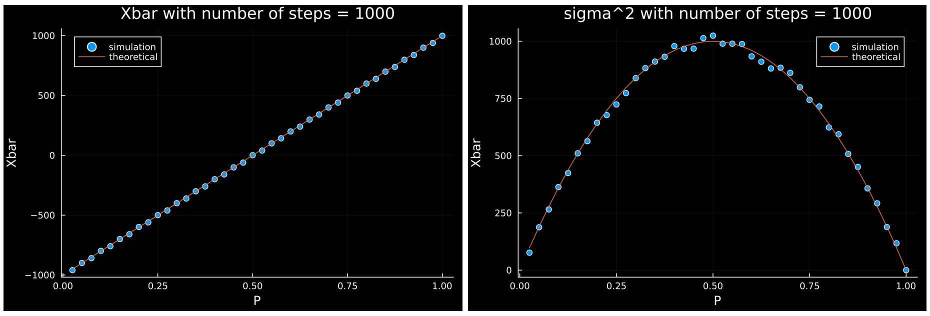

Figure 2: Simulated and Theoretical results for $< x >$ and σ^2.

reasonably accurate.

3 Trapped Random Walk

This section is similar to the previous section with the difference that when the walker reaches a certain point it gets trapped.

3.1 randwalkdeath() function

This function takes P and the initial position as input, then simulates random walk until the walker reaches a trap. Then the function returns the time it took for the walker to get trapped.

3.2 Averaging Loop

We use multiple loops to run the simulation multiple times for each initial state and for different values of P, then average the results and summarize them into an array that could plot all of them easily. We also use a matrix to store the strings used to label the data sets.

3.3 Results

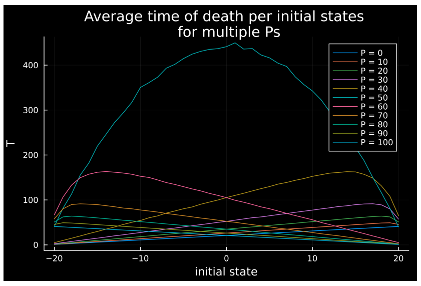

Figure 3: Results for the Simulated Trapped Random Walk.

4 Trapped Random Walk Statistically

We could skip the simulation part and solve the trapped random walk problem statistically by calculating the probability of getting trapped with each number of steps like figure(1) in the book.

4.1 timepassed() function

This function takes P and the initial position as input, then calculates the probability of getting trapped for different times until the probability reaches a cutoff. It then returns the time.

4.2 Main Loop

The loop we use in this section is really similar to the previous part with the difference that we run the function once since it's deterministic. The plotting process is also really similar to the previous part.

4.3 Results

There are differences between the simulated and statistical results but the overall shape is the same.

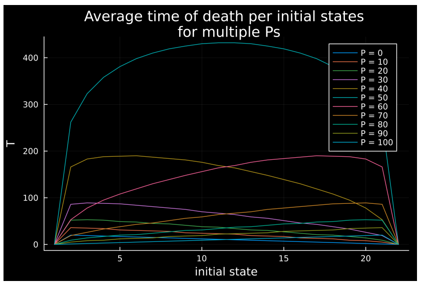

Figure 4: Results for the Theoretical Trapped Random Walk.

5 2D Random Walk

Now we simulate a random walk on a 2d grid. the code for this section is also simple and needs minimal explanation.

5.1 randwalk2D() function

this function only takes the number of steps as input, the simulates the random walk by choosing one of the four directions randomly with equal probability and moving in that direction. Then it returns the squared distance from the origin r^2.

5.2 Averaging Loop and Theoretical Values

We use a loop to run the simulation a large amount of times and for different step counts. We also calculate the corresponding data using the theoretical formulas and plot them both on a single graph.

5.3 Results

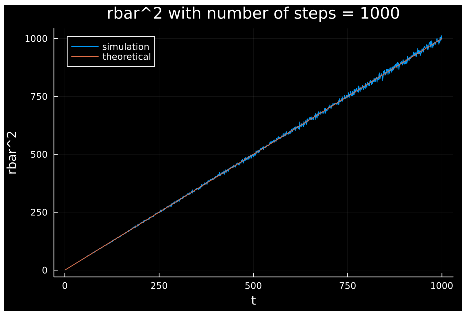

Figure 5: Results for the 2D random walk including the theoretical values.

6 Random Walk with Growth

For this problem we use a matrix which we define the width and height of it. The bottom edge will be the initial seed. The program is written in a way that cells with the value -1 are blocked and the walker could not enter them and if that path is chosen, the walker will wait and decide again. The three other edges are filled with this value. Cells with value 0 are empty and cells with positive values make up the cluster.

6.1 maxheight() function

This function takes the matrix as input and scans it horizontally and returns the first height in which the entire row is empty which is also the maximum height of the cluster plus one.

6.2 checkneighbours() function

This function takes the matrix and the coordinates of a point. then it checks the neighbouring cells for an attachable cell. Then it returns one if such a cell exists, and zero if it doesn't.

6.3 randomwalk() function

This function takes the matrix as input, then it releases a random walker whose height is 20 units above the maximum height of the cluster and its horizontal position is randomized. The walker continues it's journey until it attaches to the cluster. Each walker that gets attached assigns a larger number to the cell which creates a color gradient when plotting.

6.4 Main Loop and Plotting

The main loop is extremely simple and just runs the random walk function multiple times so that enough points are deposited. We also use a heatmap to plot our results.

6.5 Results

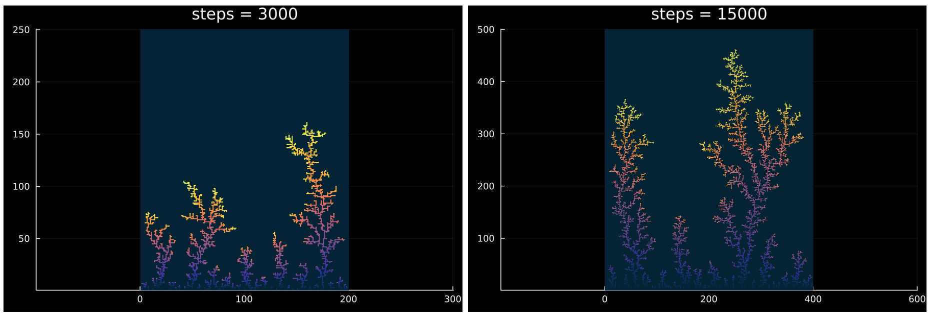

Figure 6: Results for random walk with growth.

7 Self Avoiding Random Walk

For this section we simulate a 2d random walk with equal probabilities with the added rule that the walker should avoid points it has already passed. For this we use a sufficiently big matrix so that the edges don't come into effect and we label the blocked cells as 1.

7.1 count() function

This is a recursive function that takes the matrix, the number of remaining steps and the coordinates of the walker as input, Then it takes all the paths possible all the while avoiding the blocked cells. Each time the function gets zero as the input of the remaining steps variable, it adds one to the counter which is a global variable. When all the steps are taken we are left with the number of paths for a given number of steps. (It is important to deepcopy() the matrix to avoid scope issues in Julia).

7.2 Main Loop and Plotting

We use a loop to run the function for different number of steps, and to generate the number of paths that a free random walker could take into an array. Then we plot the results of the number of paths based on the number of steps and the relative results of the number of bounded paths and free paths based on the number of steps.

7.3 Results

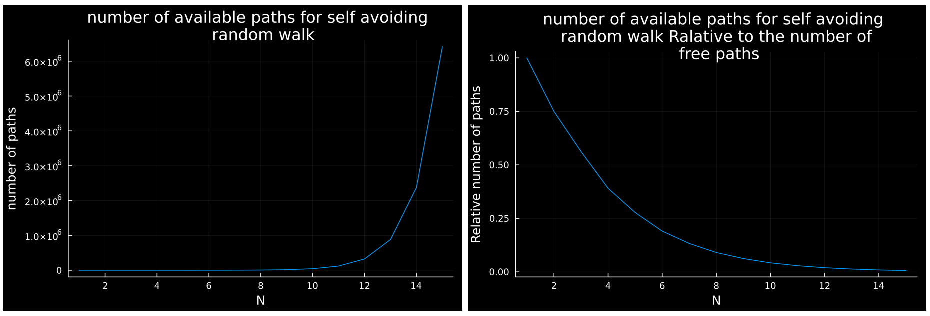

Figure 7: Results for self avoiding random walk.

The relative results go to zero with the increase in steps.