Drunken sailor problem
Scientific Calculation II final project

Wille Alapappila

December 19, 2020

Contents

1 Introduction

In this problem we consider a sailor returning home from a bar 10 blocks away from shore in a infinite city with an infinite coastline along y-axis. If the sailor manages to navigate to shore before 10 hours have passed, she will make it in time for the ships departure. The sailor chooses a random direction at every crossroad. If she walks for 50 years she will die.

In the second part of the problem the sailor knows not to go where she has been before so that at each crossroad she chooses randomly between the available options, i.e. between the directions that don't lead to a square she has been to before.

Even though the problem may seem like a simple curiosity, it demonstrates the concept of random walk and self-avoiding walk (SAW) quite well. The random walk in turn can be used to simulate Brownian motion and the motion of stock markets, for example.

Due to the nature of a random walk, there aren't a lot of mathematical models that accurately predict properties of the phenomenon. Thus, we are inclined to use computational methods to study the random walk.

Our goal with the simulation is to compare the sailor's odds to board the ship in time between the normal random walk and the SAW and the time it takes her to get there. We aim to visualize the differences between the types of random walks with plots.

2 Methods

We start from the bottom module, timemod.mod. Its only real task is to convert time from minutes to a more intuitive form, e.g. x years, y days. This is done with the convert_time() function. It is a fairly large loop that considers each different possibility of options separately so that we don't have for example zero days and six hours.

A second module equally low in the chain is the mtmod.mod module provided by course materials. We use its igrnd() function to randomize direction in the movement module and the main loop.

A module one above the previous two is movement.mod. There we first define the type position_vector with components x and y, which makes it easy to do coordinate calculations.

The main content of the movement module are two functions, update_position() and update_position_saw(). update_position is used in the regular random walk simulation as well as the first step of SAW simulation. It randomizes the chosen direction and returns the coordinate value of the new square.

update_position_saw() is the same function as update_position except it doesn't randomize anything internally. It takes arguments position and direction, with which it calculates the new position. The randomisation of direction in SAW is done inside the main loop.

The top module is coordinates.mod. There we define the coordinate system where the sailor lives as well as a four component list that contains our direction. We represent the direction by assigning the value 0 or 1 to the components of the list. The first component represents right, second component left, third component forward and fourth component backward direction.

Inside the module we have subroutines update_grid() and available_test() in addition to function find(). update_grid takes the coordinate matrix which is filled with zeros and changes the current position to a one. This enables us to track where the sailor has travelled for the SAW.

available_test() takes the position, grid and available directions list as inputs and checks if the squares next to the current square are free to move to. It returns a list with 1's in place of occupied directions and 0's in place of available squares.

The find() function returns the index of the first values in a list for which the value is some specified number.

3 Implementation of the methods

In the main program we use all the modules discussed above. First we open the file 'output.dat' for writing if the user so specifies and then begin the do loop that handles the simulation. We have an if-condition that decides

whether we want to simulate a regular random walk or a SAW, which is also specified in the command argument. The third command argument determines the number of simulations.

For each new walk we set the position and time to zero so the sailor resets. In our normal random walk loop we have three main conditions: if the time is over 50 years, the sailor dies and we start over, if the position is 10 blocks or more, the sailor has reached the shore and the walk is a success and in any other case than the previous two the sailor walks on with a new randomized direction at each crossroad. We achieve the randomisation by the function update_position() which was discussed in the previous section. After position update we also write the position to 'output.dat'.

After each exit from the loop that walks the sailor around, we calculate the distance and time it took to either die or get to finish and add it to the list of distance or time which we will use later.

If we want to simulate SAW, we have another large loop that works mainly the same but with a few key differences. First: now we don't wait until the sailor dies – if she doesn't make it in time to the shore, we exit and consider the walk a failure. Second, we now have the possibility of a dead end since the walker avoids spots it has been to already. This means it can walk to a square that has no free squares around it. Third: we can't calculate the randomized direction the same way because now we may have less than four options.

In our movement loop, we first check if the value of the sum of the components of the available() function is 4. If it is, we've reached a dead end and the walk is a failure. Next, we check if the sum is 3: this mean that there is only one available direction and we can find it using find() function from module movement.mod. Then we use the update_position_saw() function to update position.

If the sum is 2, we find the first available direction using find() and then set that component to 1 after which we can use find() again to find the second component. Now we randomize between these directions. The case sum = 1 works similarly but with one more use of find().

After we have simulated the wanted walks, we close the file and calculate

the probability to die (in random walk), get to a dead end (SAW) and get to finish. We also calculate the average, minimum and maximum values for time and distance using our lists time() and distance() and Fortran's intrinsic functions.

4 Results

Running the code with 10000 normal random walks, we find that the average distance is 9876.27 km, the maximum distance is $2.10 \cdot 10^6$ km and the minimum distance is 1.3 km. The average time is 75 days, 9 hours and 37 minutes, the minimum time is 13 minutes and the maximum time is 50 years.

The percentage of successful walks was 55.93% and the percentage of deaths was 0.21%.

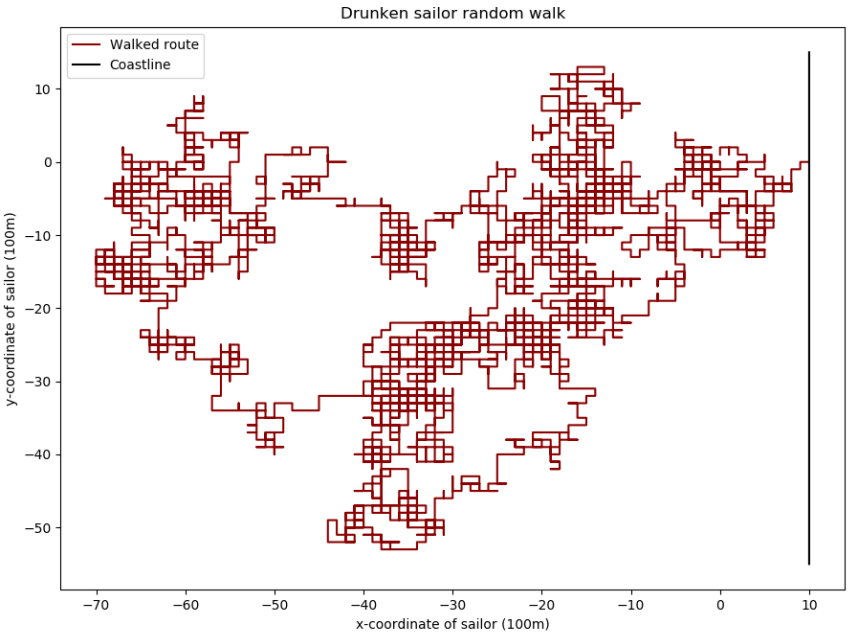

Figure 1: Normal random walk where the sailor dies of age

Running SAW we find the average distance to be 28265 km, the maximum distance to be 56868 km and the minimum distance to be 5.80 km. The average time is now 57 minutes, the minimum time is 7 minutes and the

maximum time is 8 hours and 14 minutes.

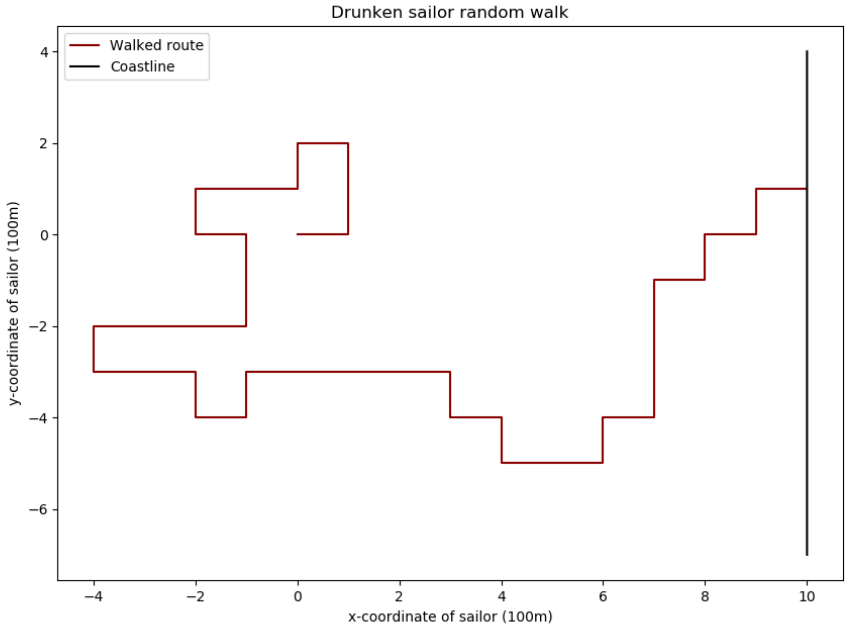

Figure 2: SAW simulation where the sailor gets to the shore in time

The percentage of successful walks was 25.91% and the percentage of a dead end was 74.09%.

5 Conclusions

Comparing the results between a normal random walk and a SAW, we can come to the conclusion that while the SAW might sound like a more optimal strategy to navigate to shore successfully, it actually makes it harder to get to finish since most walkers end up in a dead end. We can come to this conclusion even though we didn't simulate the full 50 years for SAW since of the 10000 walkers not even one made it to even 10 hours.