Drunken sailor

Miro Valorinta,
014942387

Introduction

In the Drunken Sailor problem, a drunken sailor leaves the bar and has to get to the shore to his ship in ten hours. The bar is located at (10,10) ,where one unit represents 100m, and the shore is on the y-axis (0,y). The sailor can live up to 50 years before dying. The chance of the sailor getting to the ship, shore and dying are all things that are wanted to know, if the walk type is chosen to be the normal walk (i.e. in the command line parameter ‘walk’). However if the walk type SAW (same name as the parameter) is chosen the only interesting probabilities are the probability of the sailor getting stuck and the probability of them getting to the ship on time.

The Drunken Sailor problem is essentially a variation of random walk (SAW is random walk with knowledge of past positions), which is used in Monte Carlo simulations. These kinds of simulations have many applications in physics, for example they can be used in calculating distributions of mobile ions[1].

The background for this is relatively old, and I couldn’t find a definite date for when it first appeared, but it’s been a classic text book case for explaining Monte Carlo simulations and random walk. In the original version however the drunken sailor is on the edge of an abyss, and he either takes a step to the abyss and dies, or he can walk away from it. Then from this the probability of him living after taking n steps is calculated by doing many runs and calculaiting the probabilities using the information from them.

Methods

The program goes through n amounts of iterations given by the user as an command line argument. In each iteration a sailor leaves the bar and starts walking, taking random directions until they either reach the shore or die. If SAW is chosen until 10 hours has passed, or in other words the sailor has walked 600 units (60km). The results are then saved, and when all iterations have been finished the program tells the important information using the results.

The time is calculated using a semi-iterative method, first the time is given in minutes, then the amount in years is calculated flooring the result (using 365 days in year, so no leap years), then the days are calculated by substracting the years in minutes from the time, and then the same is done for hours and minutes.

The average is calculated by taking the sum of the array’s elements and dividing that by the size of the array.

The fractions are calculated by dividing the amount of sailors achieving one condition by the amount of all runs.

1 https://books.google.fi/books?id=hdeODhjp1bUC&pg=PA327&redir_esc=y#v=onepage&q=Monte&f=false

Implementation of the methods

To compile the program first go to the /src file in terminal than you just have to write and run make to the terminal. This should create an executable called sailor. There might be an error message saying that certain modules don't exist; if this happens you have to compile all modules using your fortran compiler (e.g. gfortran). To do this write and execute commands gfortran arrayTools.f90, gfortran getParams.f90, gfortran TDTools.f90, gfortran mtmod.f90. This explanation all also found in the README.txt files.

For the random walk to be random, and not using the same seed like fortrans own subobject random_seed() and random_number(), the module mtmod from mtfort90.f90 is used. It has the ability to generate random seeds for each run, so that all the runs are not the same.

TDTools contains two functions, minute2time, which takes time in minutes and returns an array containing in the indices [1] years [2]days [3]hours [4]minutes. This is calculated using the semi-iterative method described in the 'Methods' section above. The other function dist2km changes meters to km, but it is easier to just dividing the distance by 10 to get the result, so it is not used.

getParams is used to get parameters from the command line for the program to use. It contians two subobjects and the three command line arguments as public objects that the program can use. The correctTypes subobject checks if the 2nd (saving trajectories to output.dat) and 3rd (walk type) command line arguments are of the correct type, i.e. Y/N and walk/SAW. The getVals tries to get each command one at a time, if the first command is not found, or it's not an integer, the program tells the user about the problem and tells what to do. Then if the first argument exists and is an integer, the second argument is checked, and then if it exists the third. Then if they exist the correctTypes is called. These arguments are saved to the public objects.

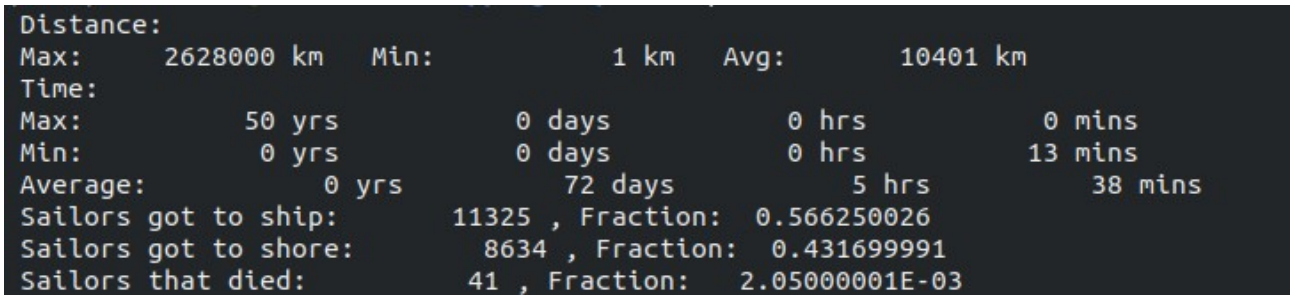
arrayTools contains many different functions to do with arrays. The function randValList returns a random value from a given array using the grnd() from mtf90.f90. The removeZeros removes all the zeros from a given array. The containsLocation checks if an array contains a given location. The availableDir takes an array and the x and y positions as arguments. It then checks what directions are available from all the directions using the containsLocation function and then gives impossible directions 0 as value. Finally it returns all the directions in the form of an array.

drunkenSailor module contains all the functions and subobjects needed to do the runs. The function randWalk takes the current position and the history of locations as arguments, the history is only used if SAW is chosen. The function gets a random value from the possible directions, which is given as 1-4 for walk, and SAW uses the function availableDir, and then changes 1-4 for the movement using if statements. The leaveBarWalk goes through one iteration of runs, and stops if the shore or 50 years is reached (Notice this can be slow, so choosing walk can take minutes to run if many thousands of runs is chosen) and adds one to the appropriate result, or in the case of time and distance, it gives the time as result. The leaveBarSAW functions similarily to the leaveBarWalk, except it saves the locations, and the maximum time is 10 hours instead of 50 years. The subroutine printing does all the printing of results.

The start() subroutine first gets the parameters from the command line and then gets a random seed for the mtmod. Then it allocates space for arrays and gives values to min and max. If the trajectory saving is chosen, it creates and opens output.dat (the previous output.dat is deleted) to the src file. In an if statement the walk type is chosen using the command line argument. Then it does the runs and finally prints the results and closes the possible output.dat file.

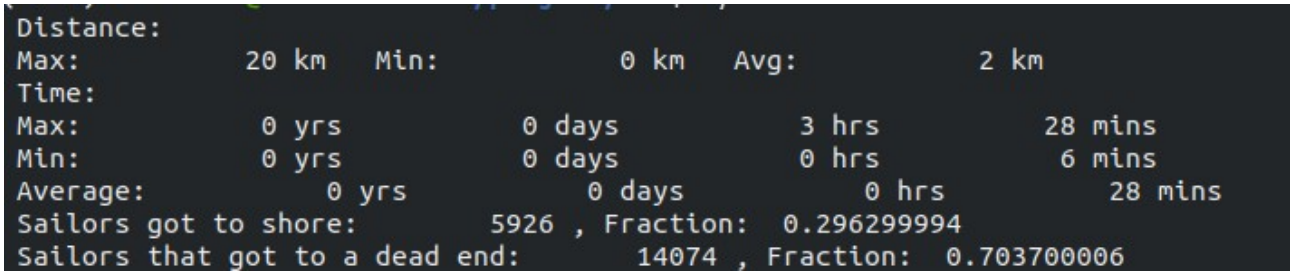
Results

The results are by definition random, but the higer the number of runs the more reliable the fractions (i.e. results) get. Here's an example result for the execution of ./sailor 20000 N walk:

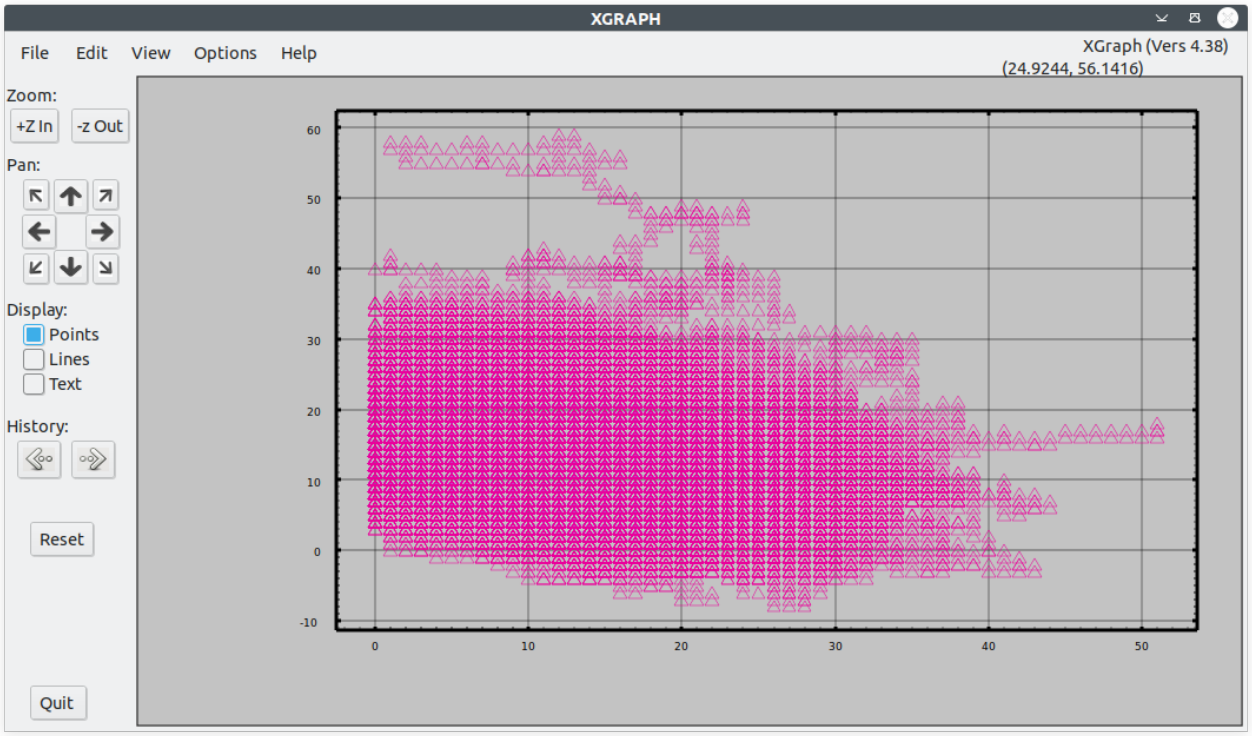


Here the distance is rounded to smallest km.

Here's an example result for the execution of ./sailor 20000 Y SAW:



Here's an example execution of xgraph output.dat:



Conclusions

There's much room for improvement, for example, could try to save the positions as a 2-dimensonal array instead of a 1-dimensional array that it is right now. Also the program could try to run xgraph directly after the output.dat is closed, but I couldn't get it working right. Also the output.dat should be opened to /run instead of /src where it is saved right now, but the problematic part is the location of the current directory. The data representation could also be more refined, for example a histogram of the distances and times could be made.