Simple Epidemic Simulator

Hermanni Rajamäki

18. December 2020

1 Introduction

A simple epidemic simulator can give a grasp on how diseases spread and what can be done to limit that. Especially the characteristics of the disease and population, such as contagiousness and population density, can have a huge effect on pandemics. The epidemic simulator works by getting certain starting values from the user and simulating the spread of the disease with some randomness involved. The simulator sets a grid with people scattered around it randomly. Then depending on the starting values the disease starts to spread through the population by moving every person step by step and looking through every interaction (same coordinates) between sick and healthy person for infection. In the following text we go through the code in detail and look at the results.

2 Methods and Implementation

The method of this simulator is rather simple; it is purely based on simulating movement of individuals and using simple probability calculations to determine infections. Every probability calculation uses uniform distribution to get a value, which determines the outcome of the situation. For example if there is a 75% change of a getting infection, we will determine a value between 0-1 and if that value is below 0.75 the infection happens. The movement of every individual follows the same idea, so that every individual moves randomly. There is also a periodic boundary implemented in the code. When a walker goes over the allowed area (grid) the simulation moves the walker to the other side. This way we actually approximate a larger system (population) than the simulation simulates. More details and implementation of the method can be read in the description of the modules.

The source code consists of multiple functions and subroutines. All of these subprograms have been arranged to fitting modules: functionality module, operation module and filehandling module. There is also a type module and a random number generation module. The random number generation module (mtdefs/mtfort90.f90) has several functions to get a random value. We mostly use the one which gives a uniformly distributed random value

between 0-1 and one which gives a value between 0-n (n being a given value to the function). All of the variables in the source code use the default kind.

2.1 Typemodule

Type module defines the type population. Population consists of people with coordinates and their status (healthy, sick, immune). These coordinates are set in two dimensional array called xy. Every column consists of one person. The first row represents his/her x-coordinate and second row y-coordinate. Of course these coordinates can be other way around. Population type also consists of vector named status. The n:th index of this vector tells the status of the n:th person (column) of the xy array. The status is coded as an integer. One represents healthy, two is sick, and three is immune.

2.2 Filehandling Module

The filehandling module consists of subroutines that write in a file or read from a file. The subroutine readfiles takes values from the file input.dat. Those values are then assigned to given variables which are later on used to run the simulation. These values are the gridsize, population size, probability to get sick, probability to get immune, amount of timesteps, sick people at start and immune people at start.

The subroutine writefilexyz takes the coordinates and status of every person in given population and writes an xyz-format file (output.xyz), where the coordinate and status of a person is written on a same line and every person is separated with a new line. The subroutine writefiledat takes only the status of the population and writes the amount of healthy, sick and immune people in a csv-file format (output.dat). The first column is the amount of healthy people, second the amount of sick and third the amount of immune. Every new line represents a timestep.

2.3 Functionality Module

2.3.1 Subprograms to Set the Scene

The functionality module consists of multiple functions and subroutines. Let's start with the methods needed in setting the starting state.

The subroutine setcoordinates simply creates a grid with people scattered around it randomly. The subroutine sets random values to the population's xy-array between 0-n using the given random number generator module (mtdefs). That way we have a population in random positions inside a (n+1)*(n+1) grid (x and y can have 0 as value, meaning one more row and column for the grid). Every person in the population then gets an status healthy.

The setsick subroutine sets given amount of population sick. If the given amount is larger than the population, then all of the population is sick and if the given amount is negative then nobody is set sick. The setimmune subroutine works almost the same way. Now the difference is that the sick people are given priority: If the given amount of immune people is larger than the healthy people, then the rest of the healthy people are set immune so that no sick people is set immune with the subroutine. If the given amount of immune people is negative naturally no-one is set immune.

2.3.2 Subprograms Needed in Running the Simulation

On every timestep in the simulation, every person has to move. If there is an encounter between a healthy person and a sick person then there always has to be a chance for the healthy person to get sick. There also has to be a chance for the sick person to get immune.

The subroutine move moves every person in the grid one step. The movement is governed by probability. There are four directions the person can move with equal chance. Using the random number generator module we get a value between 0-1 (uniform distribution) for every person. if that value is between 0-0.25 then we add one to the person's x-coordinate, if between 0.25-0.5 then substract one etc. Ultimately every person's eiher x or y value changes by one meaning that every person moves one step to

some direction. In the case that the person moves out of the grid the subroutine uses where function to find every coordinate where the values is less than 0 or more than n. If the value is less than zero the value is changed to n and if the value is more than n then it is changed to 0. That way when the person goes out of bounds they hop to the other side and we can simulate a periodic boundary.

The subroutine getsick spreads the disease. It uses another function, sickpeople, that lists every sick person's columnindex to an array. Then the subroutine goes through every sick person in that array and finds every healthy person in the same coordinates with where function. After that the healthy person's status changes to 4, which simply means pending in this case. After every healthy individual who encountered a sick person has value 4 we loop through all of them and use the given probability (p_c) to decide if they get sick. Again using the generator module we get an uniformly distributed value between 0-1. If the value is less than the given probability the person's status changes to 2 (sick). If the value is more than the probability then the person's status changes back to 1 (healthy). Now the probability to get sick doesn't depend on how many sick people are on the same site as the healthy person because the status 4 is always set if there is one or more sick person sharing the same coordinates.

Lastly the subroutine getimmune is very simple. It goes through all the sick people using the function sickpeople and sets their status to 3 (immune) with a probability (p_h) given by the user. The probability works the same way as in the subroutine getsick.

2.4 Operation Module

The operation module combines all the methods in the functionality module. First it has a function setvalues, which uses the functions setcoordinates, setsick and setimmune to set the grid and the population in it. That way we have the starting scene ready.

Then there is the subroutine timestep. The subroutine first calls the getsick subroutine to spread the disease. Then it calls the subroutine getimmune to "cure"the sick people. It is important to note that the simulation now prioritises the getsick subroutine. Even if the sick person gets immune in the

same timestep it can still spreads the disease because we call the subroutine getsick first. Lastly it calls the subroutine move to move every person. Now when we call the timestep method again we spread the disease and cure people and then move again.

2.5 Main Program

The main program now uses these modules to make the simulation work. First it reads the input using filehandling module, then creates population variable and sets starting values to it (setvalues function). After that it loops the subroutine timestep to make the simulation work. Every timestep it also writes the output data, which we can later on plot.

After running the simulation the program calls python program to plot the amount of healthy, sick and immune people every timestep. That plot is given as the default output from the whole program.

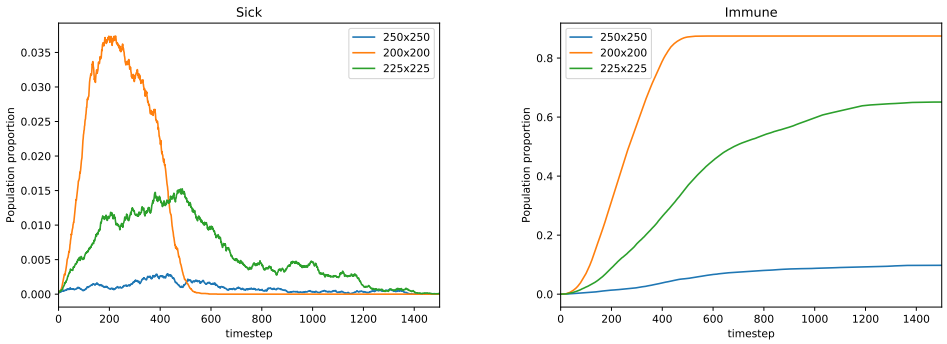
To compile the program use command gfortran typemod.f90 filehandling.f90 mtfort90.f90 functionalitymod.f90 operationmod.f90 main.f90 in path /rajamäki_ hermanni_ sc2_ project6/src/. To run the program just type ./a.out in the same path. To change the inputs go to the path /rajamäki_ hermanni_ sc2_ project6/run/ and change the values in input.dat file.

3 Results

Now let's see what type of results we get from the simulation. Instead of looking at the absolute value of population size in the graphs we look at the population proportion because the simulation can be used to approximate larger populations (periodic boundary). It is also important to note that the graphs in this section haven't been done using the default output from the program.

Now using the values in the table we can compare the effect of population density ($\frac{population}{gridsize}$). Comparing the pictures we see that with higher population density (orange graph) the maximum of sick people is higher. The disease goes through the population much faster. On the other hand we see

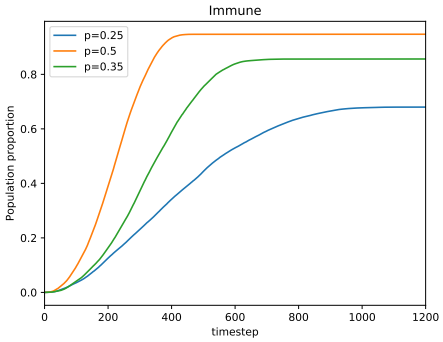
	1. Scenario	2. Scenario	3. Scenario
Length of square's side	200	225	250
population size	50000	50000	50000
probability to get sick	0.25	0.25	0.25
probability to get immune	0.075	0.075	0.075
people sick at start	10	10	10
people immune at start	0	0	0
timesteps	2000	2000	2000

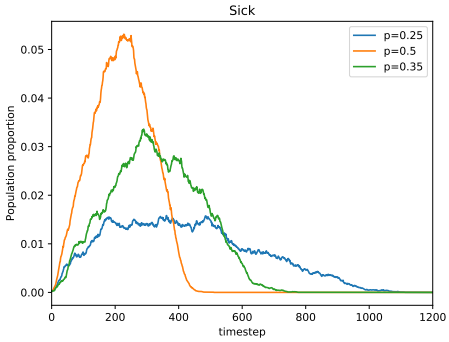


that the disease also vanishes faster. With lower population density (green and blue graphs) the disease lingers longer. Especially with the green graph we can see in the immunity graph how the disease is still spreading through the population while in the 1. scenario (200x200) the epidemic is essentially over. This is because in 1. Scenario the population get immunity much faster and we can see the effect of herd immunity: with every encounter with a sick person there is higher change it is between immune person who can't get the disease and spread it further. In the 2. and 3. scenario the population doesn't have good herd immunity so early. Although the disease is over faster in 1. scenario, substantially more people have been sick compared to 2. and 3. scenario.

Let's see how the probability to get sick affects the epidemic. Now the values are in the second table (next page).

	1. Scenario	2. Scenario	3. Scenario
Length of square's side	225	225	225
population size	50000	50000	225
probability to get sick	0.5	0.25	0.35
probability to get immune	0.075	0.075	0.075
people sick at start	10	10	10
people immune at start	0	0	0
timesteps	2000	2000	2000

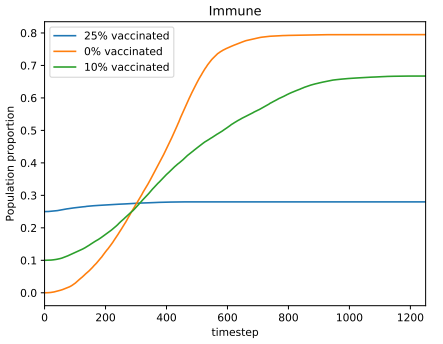


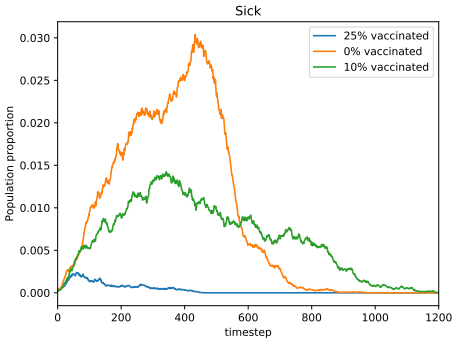


The scenarios behave very similarly to the population density change: The higher the probability to get the disease the higher peak we have in the amount of sick people. On the other hand the epidemic ends sooner because the population gains the herd immunity sooner. Also overall more people got sick in 1. scenario than in 2. and 3. scenario.

Let's look how vaccination (people immune at start) influences the simulation with the following values used.

	1. Scenario	2. Scenario	3. Scenario
Length of square's side	225	225	225
population size	50000	50000	50000
probability to get sick	0.3	0.3	0.3
probability to get immune	0.075	0.075	0.075
people sick at start	10	10	10
people immune at start	0	5000	12500
timesteps	1250	1250	1250





We can see a drastic effect of vaccination. With only 10% of the population vaccinated we can already see that the disease affect's less people. With 0% vaccinated around 80% of the population gets sick. With 10% vaccinated only around 50% of the population gets sick (0.6 is immune but 0.1 were

already set at start). With 25% vaccinated the population seems to already have a herd immuity and the disease dies off quickly. We can also see that with 10% vaccinated the disease lingers longer in population compared to the orange graph, but the peak of sick people is lower.

There is also a video (Ovito.mp4) attached to the folder which uses ovito to demonstrate visually how the disease spreads through a population using the first table 2. scenario's values. the blue dots are the healthy people, white are sick and red are immune. In the video you can see how the white color leaves red behind it, meaning that the frontier of the disease moves forward and lefts immune people behind. You can also see how the periodic boundary works.

It is good to remember that all the individual graphs were the result of one run of the simulation. Because there is randomness involved in the simulation different runs using the same values can give quite a different output. To get more concrete results one should use the average of multiple runs with same values and plot that.

The approximation for larger populations seems to be quite good, although the simulator expects that the population density stays around constant in the population (walkers uniformly distributed to the grid). However if the population size is set much smaller the approximation gets worse because small fluctuations due to randomness in the simulator begin to have bigger impact. These fluctuations don't represent what happens on a larger population. In the next section we will discuss the efficiency and realism of the program and how to improve it.

4 Conclusions

The epidemic simulator gives a pretty good idea of how diseases spread through a population although there are certain things making the simulation unrealistic. For example the simulation doesn't take into account how many sick people are in the same coordinates as a healthy person. Realistically the probability to get sick should change the more sick people are in the healthy person's vicinity. Also the effect of vaccination seems to be higher than one would expect. This might be related to the fact that a sick person gets immune relatively quickly so he/she doesn't have a lot

of encounters before becoming immune. Therefore encountering a immune person has a bigger impact on the epidemic. This can be simply solved by changing the probability to get immune smaller.

The code itself could also be optimized better. For example in the method getsick, always following the sick people's encounters can become quite heavy when the amount of sick people get high. One way to improve this is to make the code follow the healthy people's encounters when the amount of sick people is higher than the healthy people. Also every timestep getting rid of the immune people, who don't contribute to the spread of the disease, would also make the code lighter.