Modeling COVID-19 with Principal Component Analysis and the Spatial S.I.R. Method

UCLA M20 FINAL PROJECT FALL 2020
DECEMBER 11, 2020

EDEN ZAFRAN

1 Main Component Analysis with PCA

1.1 Background

Principal Component Analysis (PCA) is a dimensionality reduction technique that is used to extract the most important data from a given dataset. The purpose of dimensionality reduction is to display the data in a way that is easily interpreted, at the cost of ignoring some parts of the dataset. PCA is a method that aims to find the most important variables (the principal components) and reduce information loss from the ignored variables.

This project takes data relating to the COVID-19 pandemic from 27 different countries and aims to determine the most relevant variables. The six variables that were considered were: infections, deaths, cures, mortality rate (measured as a percentage), cure rate (measured as a percentage), and infection rate (measured as a percentage).

The Principal Component Analysis method has 5 main steps:

- Normalizing data
- Computing covariance of the data
- Computing eigenvalues & eigenvectors of the covariance matrix
- Determining the principal components
- Projecting normalized data onto the subspace spanned by the principal components

The data is normalized so that if one variable is measured with large numerical values (e.g. populations being represented in the order of hundreds of thousands of people) in comparison to a different variable that is measured with small numerical values (e.g. percents being represented numerically in the order of tenths and hundredths), no one variable will overpower the other solely on the basis of numerical value. The goal is to find the relative relation of one variable to another and to find which variable has the greatest effect on the data.

1.2 Structure of the Main Script

In the main script, "`project_105344247_p1.m`", the principal component analysis is performed and displayed. First, the script loads the dataset ("`covid_countries.csv`") into MATLAB from the current folder using the `readmatrix` function (any similar function that performs this task is also fine). Since the first two columns of the provided csv file contain the date and the country name, it is necessary to remove these two columns, since they do not contain any information from the 6 different variables that are being compared. This leaves behind only the relevant data in a matrix, which can then be passed into the `myPCA` function for implementation of the principal component analysis method. The script then calls the custom `myPCA` function, which will perform the principal component analysis and return the full dataset projected onto the subspace spanned by the principal components. To reduce the data to only 2 dimensions, the script then takes the first two columns of the data that has been projected onto the subspace by the principal components and the two largest eigenvalues of the covariance

matrix (the principal components) and plots the two against each other using the `biplot` function in MATLAB. Any similar function in a different language that generates a biplot is suitable. The dimensionality of the data has now been reduced from 6 variables to only 2 variables.

1.3 Structure of `myPCA` function

The `myPCA` function takes in one input, which is the dataset we will be performing the principal component analysis on. Then, the function returns two outputs, the first being a matrix containing the eigenvalues of the covariance matrix sorted in order from largest to smallest, and the second being the dataset projected onto the principal components.

First, the `myPCA` function normalizes the data. To set up the normalization, the function computes the mean of the values in each column, then computes the standard deviation of the values of each column. To normalize, each element in a column is subtracted from the average of that column, then divided by the standard deviation of the column. The `myPCA` function utilizes a for loop that accesses each individual element in the matrix containing the un-normalized data. For clarity, the normalized data is stored in a new matrix that is the same size but separate from the matrix that contains the original data that was passed into the function. This separate matrix is pre-allocated before the data is stored for speed.

Next, the `myPCA` function computes the covariance matrix of the normalized dataset using MATLAB's built in `cov` function. Any other built-in function that computes the covariance is also sufficient (if using a language other than MATLAB). Then, the eigenvalues and eigenvectors of the covariance matrix are computed using the MATLAB's built in `eig` function (again, any other similar function works just as well in a language other than MATLAB). The eigenvalues and corresponding eigenvectors are sorted from largest magnitude to smallest. Then, the normalized data is projected onto the principal components (the two largest eigenvectors) by multiplying the matrix containing the normalized data by a matrix containing the eigenvectors sorted from largest to smallest corresponding eigenvector. Finally, the function passes out the normalized data projected onto the 2D subspace spanned by the principal components.

1.4 Results

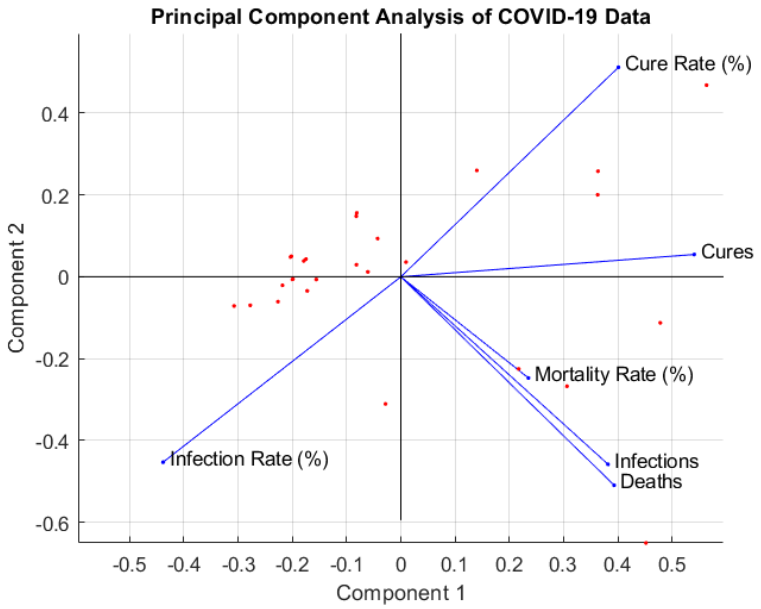

Figure 1: Principal component analysis on COVID-19 data from 27 countries. Data has been normalized and is projected onto the principal components.

The axes of the plot represent the principal components of the data, where "the principal components are the linear combinations of the original variables that account for the variance in the data" (Interpret all statistics and graphs for Principal Components Analysis.).

From this graph, we can see that the cure rate and the infection rate are negatively correlated, whereas the mortality rate, (number of) infections, and (number of) deaths are all positively correlated. There is little/no correlation between mortality rate, infections, deaths, and infection rate and cure rate. Cures, mortality rate, infections, and deaths all have positive loadings along Component 1 (are larger along Component 1), so we can infer that Component 1 is focused on the deaths in a population. From looking at how infection rate and infections both have large negative loadings along Component 2, we can also infer that Component 2 is focused with the number of infections among the population.

2 Solving the Spatial S.I.R Model

2.1 Introduction

The S.I.R model is a method for modeling the spread of disease among a fixed population. It splits a population into 3 distinct classes: susceptible (S), infected (I), and recovered (R). Some limitations of this model are that it does not take into account changes in population size from births or deaths (from both natural deaths or epidemic-related deaths), nor does it take into account the effect of the spread of a population over an area. The spatial S.I.R. model is an extension of the S.I.R. model that takes into account the effect of the spread of a population over an area by splitting the total population into a grid. Each element of the grid corresponds to a local S.I.R model.

The three equations that govern the spatial S.I.R. model are given below:

$$\frac{dI_{m,n}(t)}{dt} = \left(\beta I_{m,n}(t) + \alpha \sum_{ij} W(m,n) I_{m+i,n+j}(t)\right) S_{m,n}(t)$$

$$\frac{dI_{m,n}(t)}{dt} = \left(\beta I_{m,n}(t) + \alpha \sum_{ij} W(m,n) I_{m+i,n+j}(t)\right) S_{m,n}(t) - \gamma I_{m,n}(t)$$

$$\frac{dR_{m,n}(t)}{dt} = \gamma I_{m,n}(t)$$

Equation 1: Three equations governing the spatial S.I.R. model. “m” refers to the row number of the local S.I.R. model, and “n” refers to the column number of the local S.I.R. model. “i” and “j” refer to the location of a neighboring local S.I.R. model in relation to the specified local model.

Each equation corresponds to the rate of change of the number of susceptible, infected, and recovered individuals in a population, respectively. In addition, a weighting function describes the effect of space on the infection rate. The weighting function used in this instance assigns a weight of 1 to grid cells immediately above, below, and beside a given cell. Cells directly diagonal, or with one corner touching, a given cell are given a weight of $\frac{1}{\sqrt{2}}$. All other cells have a weight of 0. This weight is placed on the number of infected people in the local model. A diagram of this weighting is shown below in Figure 2.

$\frac{1}{\sqrt{2}}$	1	$\frac{1}{\sqrt{2}}$
1	0	1
$\frac{1}{\sqrt{2}}$	1	$\frac{1}{\sqrt{2}}$

Figure 2: Weight of spatial contact rate among local S.I.R. models in the spatial S.I.R model. The center (denoted with a weight of 0) is the local S.I.R. model being chosen at any given time.

To numerically solve the spatial S.I.R model, it is necessary to use an ordinary differential equations (ODE) solver. This script utilizes one of MATLAB's built in ODE solvers, `ode45`, as well as a custom fourth order Runge-Kutta method solver.

2.2 Structure of the `dynamicsSIR` function & the `solveSpatialSIR` function

The `dynamicsSIR` function is the function that contains the equations governing the spatial S.I.R model. The `solveSpatialSIR` function is used so that one can easily change what type of ODE solver they want to utilize to solve the spatial S.I.R. model. In this case, the model is solved using one of MATLAB's built in ODE solvers, `ode45`, and a custom fourth order Runge-Kutta method solver, `RK4`.

Since ODE solvers can only pass row/column vectors in and out, any data that is not in vector form must be vectorized before being passed into the solver. Since the spatial SIR model contains 4 dimensions (susceptible, infected, and recovered people all distributed over a space, and time being the 4^{th} dimension), it is necessary to vectorize the 4D matrix into a single column or row vector before beginning to solve the differential equation. Then, after solving the ODE, it is helpful to take the row/column vector containing the numerical solution to the differential equation and reshape it into the original matrices it came in. This is to keep the data organized and improve readability. To shape a column or row vector back into a matrix, use the `reshape` function (in MATLAB). The `reshape` function takes in the vector that will be reshaped, as well as the dimensions of the desired matrix.

2.3 Structure of `RK4` function

The fourth order Runge-Kutta method ODE function takes in 3 arguments: the function handle of the differential equation it will be solving, the time range over which it will be solving, and the initial conditions. Since the differential equation can only take in numerical values in a row vector, and the initial conditions in this case were given as a column vector, it is necessary to transpose the current value of y that were are taking each time before passing it to the differential equation (in this case, `dynamicsSIR`) and then re-transpose the numerical solution back into a column vector for storing in the solution matrix, y. In this project, step size was fixed at 0.1 unit of time, meaning there would be a total of 601 time steps (since the range model went from 0 to 60 units of time). Pseudocode of the RK4 function is given below (CEE/M20 Introduction to Computer Programming with MATLAB Fall 2020 Final Project).

RK4 Pseudocode

```
h ← set fixed step size (0.1)
y ← pre-allocate array for numerical solution
Set initial conditions
t ← pre-allocate array for time steps
numSteps = length of time matrix – 1
while n < = numSteps
     y_n ← get value of y from current time step and store
     t_n ← get value of current time step and store
     k_1 ← h* f(t_n, y_n)
     k_2 ← h* f(t_n + h/2, y_n +k_1/2)
     k_3 ← h* f(t_n + h/2, y_n +k_2/2)
     k_4 ← h* f(t_n + h, y_n + k_3)
     y_n+1 ← y_n + (k_1 + 2*k_2 + 2*k_3 + k_4)/6
end
```

2.4 Structure of Weighting Function

The weighting function assigns a weight to the number of infected people in a local SIR model, which determines the rate at which susceptible individuals in a neighboring local model get infected. As previously mentioned, the weighting function assigns a weight of 1 to grid cells immediately above, below, and beside a given cell (see Figure 2 for reference). Cells directly

diagonal, or with one corner touching, a given cell are given a weight of $\frac{1}{\sqrt{2}}$. To do this, the function determines what type of cell the given local S.I.R. model is, and then assigns a weight to the rest of the surrounding local S.I.R. models according to the instructions above. The function makes use of an if-else statement to determine the weighting. A flowchart of different weights the weighting function can assign based on what type of cell the given local S.I.R model is shown below in Figure 3.

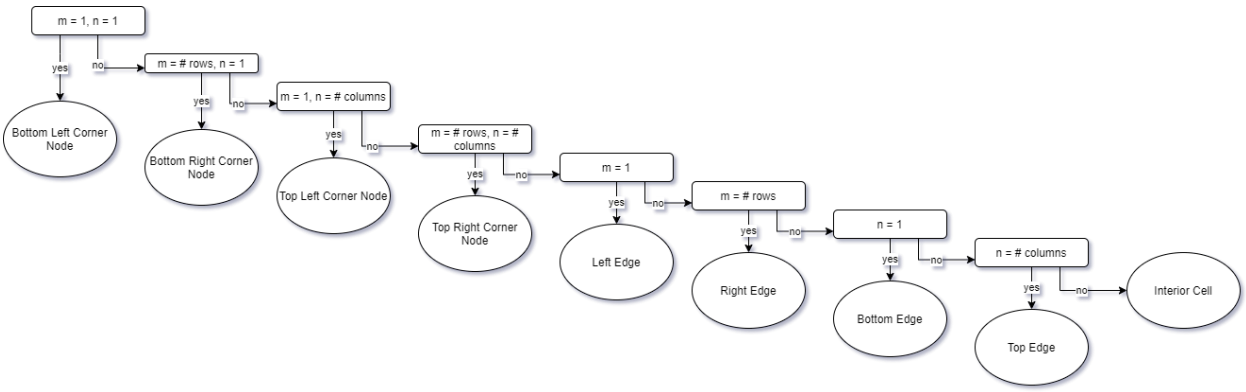

Figure 3: Flowchart of if-else statement used to determine what type of cell the passed in S.I.R. local model is in the weighting function. “m” refers to the row number of the local S.I.R. model, and “n” refers to the column number of the local S.I.R. model. Weights are assigned using these cell type designations.

2.5 Results

Below are 3 plots that hone in on 3 different local S.I.R. models made with the `plotTimeSeries` function. Figures 4 and 6 both show local S.I.R. models where the entire population started out as susceptible to the disease, and then gradually progressed through the infected stage to become fully recovered at the end of the model time frame. However, in Figure 6, we can see the spike in infections happened at an earlier time than in Figure 4. Thus, we can expect to see the population in the local S.I.R model in Figure 6 to become fully recovered before the population in the local S.I.R. model represented in Figure 4. From the graphs, this is indeed true. In Figure 6, the recovered population (shown in green) reaches 100% before the model time frame is over; however, in Figure 4, the recovered population is slowly approaching 100% recovered, but does not quite reach it before the end of the time frame the model runs over.

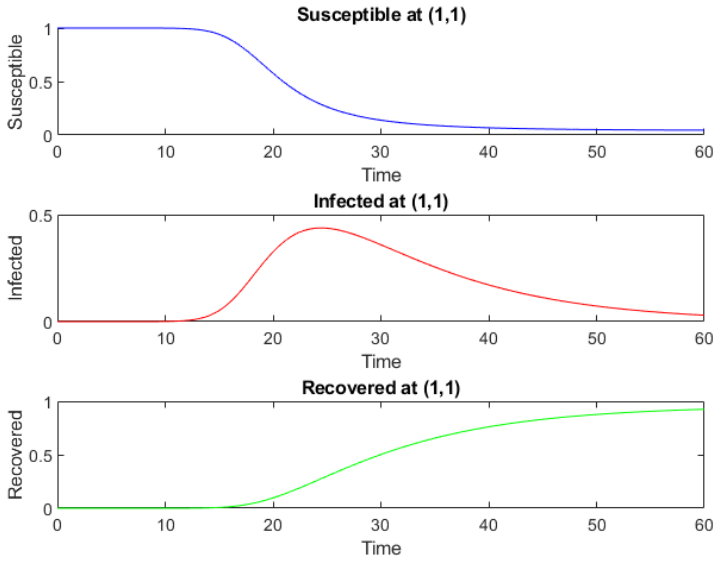

Figure 4: Local S.I.R model displaying the percentage of the population that is susceptible, infected, and recovered as a function of time at the location (1,1) on the spatial grid.

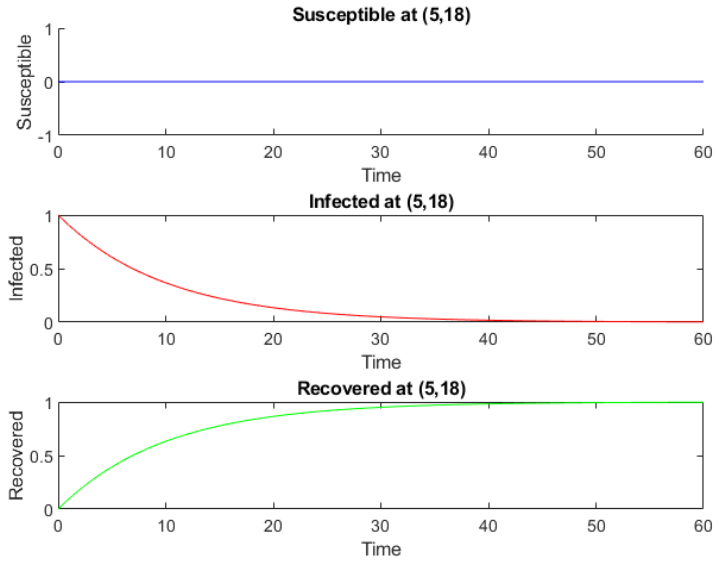

Figure 5: Local S.I.R model displaying the percentage of the population that is susceptible, infected, and recovered as a function of time at the location (5,18) on the spatial grid.

Figure 5 is an example of a local S.I.R model that started with 100% of the population infected at time 0. As time progresses, we can observe the percent of people recovered steadily increasing from 0 to 30 units of time, and by 40 units of time, the entire population is recovered. The

percent of susceptible people is kept at 0 because once a person is infected, they cannot move back to the susceptible fraction of the population, and can only move forward into the recovered fraction of the population. Since the entire population started out as infected, the population will never go back to the susceptible region, thus leaving it to stay at 0 for the duration of the model time frame.

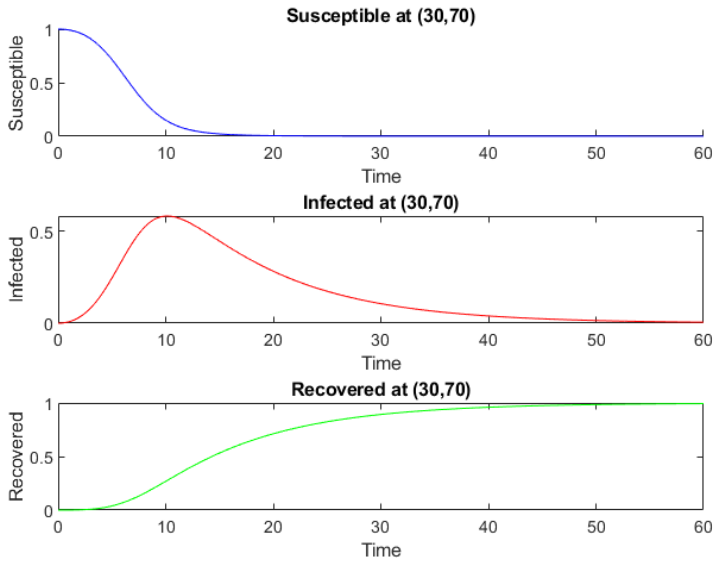

Figure 6: Local S.I.R model displaying the percentage of the population that is susceptible, infected, and recovered as a function of time at the location (30,70) on the spatial grid.

The following 5 figures (Figures 7 – 11) are snapshots of the animation that the `animation` function produces. Figure 6 shows the initial conditions of the spatial S.I.R model, where a few local S.I.R. models have a greater percentage of infected people (shown in red) than susceptible people (shown in blue). At this point in time, the population only exists in two of the three states of the S.I.R. model — no local model has reached the recovered stage yet.

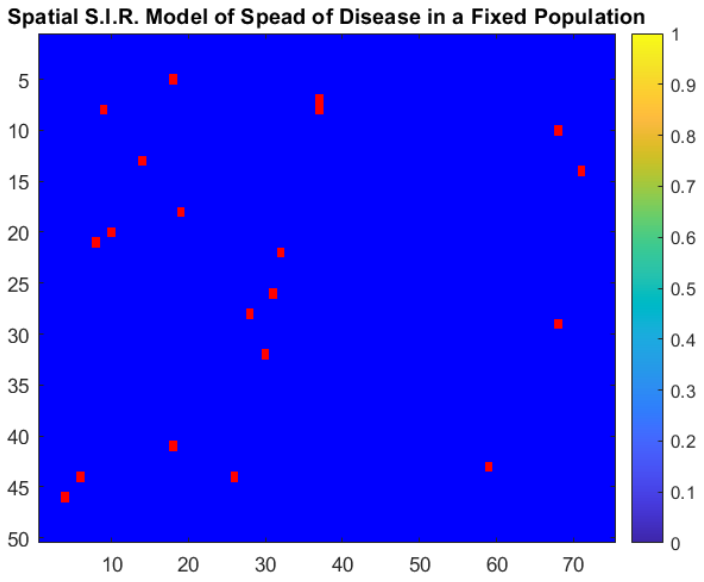

Figure 7: Frame of spatial S.I.R. model at t = 0. Displays initial conditions. The red dots correspond to the local S.I.R. models that begin with infected people.

Figure 8 shows the spatial S.I.R. model at 150 time steps from the start. We can see how the local model that were initially infected have moved to the recovered stage, whereas the surrounding local models are becoming infected from contact with the disease. However, a large swath of the population stays untouched by the disease at this point in time.

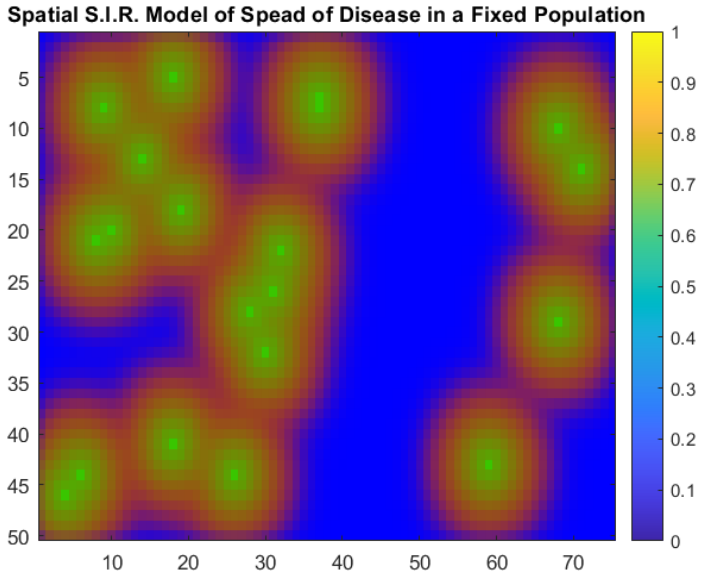

Figure 8: Frame of spatial S.I.R. model at t = 15. The local models that started initially infected have moved on to the recovered stage, while infections spread to nearby susceptible populations.

Below, in Figure 9, we can see how the disease has spread to essentially the entire population in the space, and the initial infection sites are 100% recovered from the disease, as evidenced by the strong green color in the same location as the red dots representing active infections in local S.I.R. models that were present in the initial conditions (see Figure 7).

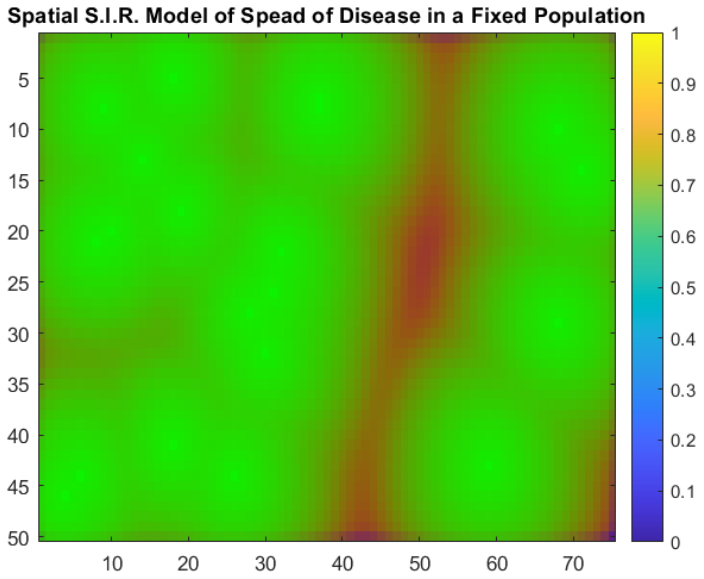

Figure 9: Frame of spatial S.I.R. model at t = 30. Displays initial conditions. The majority of the population is either recovered or infected. No local models show a majority of susceptible individuals.

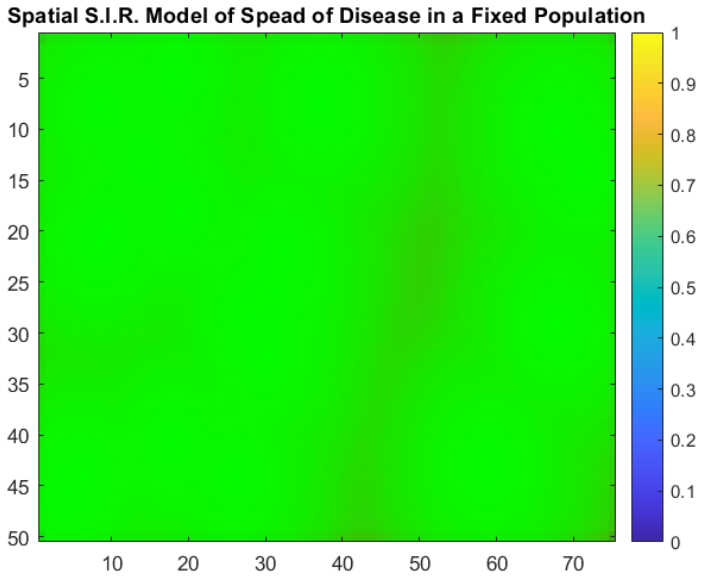

Figure 10: Frame of spatial S.I.R. model at t = 45. Almost all of the population is recovered from the disease. A few locations (where the disease took a while to spread from the initial infection sites) still have active infections.

Figure 10 shows the progression of the population's recovery. None of the population is left as susceptible, and many are moving towards the recovery stage. Figure 11 shows us the spatial S.I.R model in its last time step. The entire population, and every local S.I.R. model has moved to recovery. Some regions in the middle may have a few active infections left, but the entire population has experienced the disease and is recovered or recovering from it.

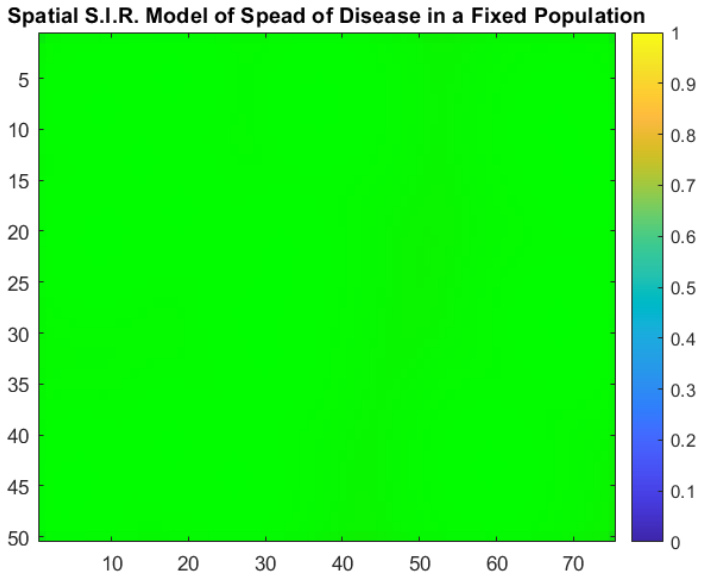

Figure 11: Frame of spatial S.I.R. model at t = 60. Displays initial conditions. The majority of the population is recovered. The entire population has been exposed to the disease.

This main script also compares the runtime of both MATLAB's built-in ODE solver, `ode45`, and the custom fourth order Runge-Kutta method solver, `RK4`, to solve the spatial S.I.R model. The script utilizes the `tic` and `toc` functions in MATLAB and displays each solver's runtime to the command window. From one specific trial, `ode45` had a runtime of 0.2335 seconds, whereas the RK4 solver had a runtime of 1.7094 seconds. This makes `ode45` about 7 times faster than the custom RK4 solver. `ode45` may be faster than the custom fourth order Runge-Kutta method solver because `ode45` uses variable timesteps. Just from looking at the outputs of the two ODE solvers compared to each other, `ode45` uses 161 timesteps, whereas the RK4 solver uses 601 timesteps. The RK4 solver requires the timesteps to be hardcoded in to the function, and is not able to change the size or amount of the timesteps while the function is being run. On the other hand, `ode45` changes the size and amount of timesteps to determine the optimal timestep within a set of tolerances defined in the function. `ode45` is able to find a balance between efficiency and accuracy, whereas the RK4 solver is only concerned with accuracy, at the cost of excess computing resources and time. The increased accuracy of the RK4

solver may not be necessary in some situations, and the increased accuracy may be so small it becomes essentially negligible. Thus, `ode45` is a more suitable, multi-use ODE solver that maintains a good balance between speed and accuracy that is applicable for many situations. Whenever one solely wants accuracy without regard for time or computing resources, then RK4 would also be a viable option.

References

CEE/M20 Introduction to Computer Programming with MATLAB Fall 2020 Final Project, University of California, Los Angeles, 2020.

Interpret all statistics and graphs for Principal Components Analysis. (n.d.). Retrieved December 13, 2020, from https://support.minitab.com/en-us/minitab/18/help-and-how-to/modeling-statistics/multivariate/how-to/principal-components/interpret-the-results/all-statistics-and-graphs/.

Joliffe Ian T and Cadima Jorge. 2016 Principal component analysis: a review and recent developments. *Phil. Trans. R. Soc. A.* 374:20150202. http://doi.org/10.1098/rsta.2015.0202.