COVID-19 Statistics with Principal Component Analysis and Spatial S.I.R. Model

Hunter Beebe - 505384217

December 2020

1 Introduction

The topic of this paper will be using COVID-19 statistics from multiple countries incorporated into a Principal Component Analysis and a Spatial S.I.R. Model to represent the spread of the illness in a systematic analysis involving the use of differential equation solvers, planes which represent larger areas (with the individual coordinate points being smaller areas within), and multiple plots and animations to characterize the events as time passes.

2 Principal Component Analysis

2.1 Theory

The technique of Principal Component Analysis (PCA) simplifies observation of data, reducing the elements looked at to only the key components, reducing the number of dimensions considered. This is done through this general process:

- Normalize the data.
- Find the covariance (correlation) matrix.
- Determine the eigenvalues and eigenvectors of the covariance matrix.
- Reorder the eigenvalues from largest to lowest in magnitude.
- Store the corresponding eigenvectors of the eigenvalues beginning with largest in magnitude, and stopping once desired dimensionality is reached (the number of vectors corresponding directly to the dimension of the projection).
- Multiply the normalized data by the matrix of desired number of vectors.

In our case, we will be projecting the data onto a 2D space, so two eigenvectors will be used for the projection. This will reduce the data from 6-dimensional to 2-dimensional through analyzing the two most important components.

2.2 Implementation

2.2.1 myPCA.m

A function is used to implement the PCA, which will output the eigenvectors ordered by magnitude of their corresponding eigenvalues, and the normalized data projected onto these vectors.

```
function [coeff0rth, pcaData] = myPCA(data)
```

The function begins by creating an array for the normalized data, and then running an algorithm that will fill that array. This is done by running through each column of the inputted data in a loop. The average of the column is taken, along with the standard deviation, and then for each value in the column the average is subtracted and the result is divided by the standard deviation to center the data.

```
normalized = zeros(27,6); %Create normalized array
for i = 1:6
    avg = mean(data(:,i)); %Find average
    stdev = std(data(:,i)); %Find standard deviation
    normalized(:,i) = (data(:,i) - avg)/stdev; %Center the data
end
```

After this is done, the covariance matrix is found by using MATLAB's built in cov() command. The eigenvalues and eigenvectors are found in a similar manner, using MATLAB's built in eig() command on the covariance matrix. The eigenvalues are then sorted in descending order, and the index is saved so that the vectors can be sorted in the same manner.

```
c = cov(normalized); %Generate the covariance matrix
[vectors,values] = eig(c, 'vector'); %Find the ...
    eigenvalues/vectors of c
[¬, index] = sortrows(values, 'descend'); %Sort the eigenvalues
coeff0rth = vectors(:, index); %Sort the vectors in the same way
```

From here, the normalized array is multiplied by the sorted eigenvectors. This creates the pcaData output, with the data being normalized along the principle eigenvectors.

```
pcaData = normalized * coeff0rth; %Create the pcaData
```

2.2.2 Problem 1 Main Script

The main script of this problem begins by loading the data given on COVID-19 statistics in multiple countries. An offset is used when reading the document to ensure that only the numeric elements of the spreadsheet are loaded. The myPCA function is then used to run a principal component analysis on the data.

```
data = csvread('covid_countries.csv', 1, 2); %Import data
[sortedvctrs, pcaData] = myPCA(data); %Run myPCA function
```

The titles of each column in the data are then assigned to a cell array, which is used in labeling the biplot that will be generated. The biplot is made by only using the first two columns of outputted eigenvectors and pcaData, since we desire a two dimension view of the correlations.

```
vbls = {'Infections', 'Deaths', 'Cures', 'Mortiality Rate', ...
    'Cure Rate', 'Infection Rate'}; %Create the data labels
biplot(sortedvctrs(:,1:2),'Scores',pcaData(:,1:2),'Varlabels',vbls);
title('myPCA of Covid Data')
```

2.3 Result and Discussion

When the main script is ran, this is the result:

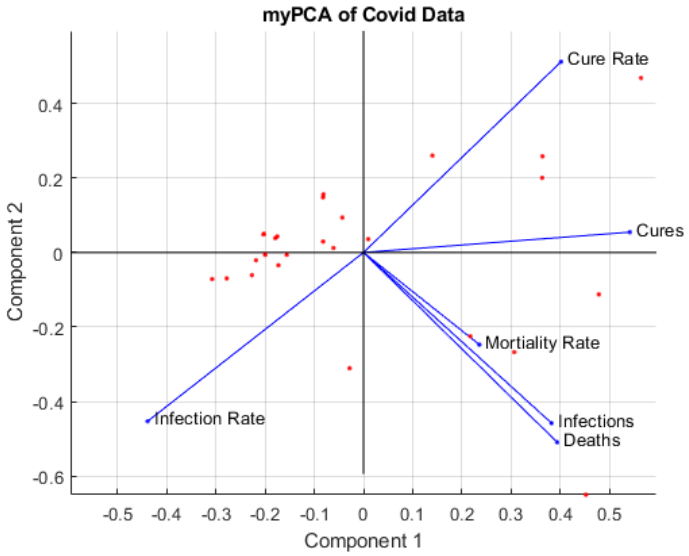

Figure 1: The output of the biplot command on the principal component analysis data, with the two most influential components being used.

The biplot of the PCA on the data can show us correlations between components of the dataset. The red dots represent the data points being used to determine the correlations. Blue lines which have a small angle between them show correlation, perpendicular lines show no correlation, and lines pointing in opposite directions show negative correlation. This allows us to make some conclusions:

- The amount of deaths and infections correlate with each other strongly. This should make sense because as more people become infected, more people will die from the virus.
- The cure rate is negatively correlated with the infection rate. This makes sense because as a larger amount of the population is cured of the virus, less of the population will be getting infected.
- The cure rate and infection rate have no effect on the mortality rate. This is sensible due to the fact that the virus will have the same mortality no matter how many people have the virus. Even if more or less people are dying as a whole, the proportion will generally remain the same.

Overall, reducing the data to two principal eigenvectors produced correlations which are realistic given the context of the situation, so our principal component analysis of the data is reasonable.

3 Spatial S.I.R. Model

3.1 Theory

3.1.1 Fourth-Order Runge-Kutta Approximation

This approximation is a method of approximating the solutions of differential equations. It operates by using weighted linear approximations of the next point based off of the derivative of the current point. The accuracy is dependent on the step size used, but a smaller step size requires more computations to be done, and thus is more intensive. The general method will be as follows:

- Load the initial conditions of the differential equation.
- Determine the number of steps that will need to be taken to reach the desired end time with the desired step size.
- Create a loop for each step which will find each Runge-Kutta approximation (there are four of them, all weighted differently). The equations are given below, where h is step size, k means the current value of a variable, and the function is the differential equation.
  - The first calculation $k_1 = hf(t_k, y_k)$
  - The second calculation $k_2 = hf(t_k + h/2, y_k + k_1/2)$
  - The third calculation $k_3 = hf(t_k + h/2, y_k + k_2/2)$
  - The fourth calculation $k_4 = hf(t_k + h, y_k + k_3$
  - And the weighted average: $y_k = y_k + (k_1 + 2k_2 + 2k_3 + k_4)/6$
- Store the calculated y value with the algorithm, along with its corresponding time based on the step size and step number.

3.1.2 Creating the Dynamics Model

The Dynamics Model is used to store the vectorized differential equations of the spatial S.I.R. model. The equations are derived from the Final Project reference document, and will be replicated here:

$$\frac{dS_{x,y}(t)}{dt} = -\left(\beta * I_{x,y}(t) + \alpha * \sum_{ij} W(i,j) I_{x+i,y+j}(t)\right) S_{x,y}(t) \tag{1}$$

$$\frac{dI_{x,y}(t)}{dt} = \left(\beta * I_{x,y}(t) + \alpha * \sum_{ij} W(i,j) I_{x+i,y+j}(t)\right) S_{x,y}(t) - \gamma * I_{x,y}(t) \tag{2}$$

$$\frac{dR_{x,y}(t)}{dt} = \gamma * I_{x,y}(t) \tag{3}$$

In this, $W(i,j)$ is definied by a weighting function which accounts the number of infected individuals increasing faster when surrounded by other areas with infected individuals. Orthogonally adjacent cells to the cell being considered have a weight of 1, while diagonally adjacent cells have a weight of $\frac{1}{\sqrt{2}}$. All other cells will have a weight of 0, including those beyond the boundary of the plane.

3.1.3 Solving the Dynamics Model

Solving the Dynamics Model will be done by passing the model through an ordinary differential equation solver. This will be done with MATLAB's built in ode45, along with our implemented Runge-Kutta solver. Initial conditions will need to be defined and passed in, along with a final time value for the solution to be generated until. The output of the solution will need to be reshaped to be easier for use, with the ideal shape in the end being a 4-Dimensional solution which has the first two dimensions as the coordinate point being examined, the third dimension being the variable (either susceptible, infected, or recovered), and the fourth dimension being time, so that there is a full solution for each time step.

3.1.4 Time Series Plotting

The Time Series Plotting requires three plots for a specific coordinate point, displaying the ratios of susceptible, infected, and recovered individuals in the local area. This will be made easy by how we formatted the solution from the spatial S.I.R. system, since the first two dimensions will be the coordinates we want to graph the ratios at, the third dimension will cycle through all three variables, and the fourth variable will help us create the time axis of the graphs. The process will simply be a loop that graphs each variable on their own subplot for each time step.

3.1.5 2D Animation

The animation will use color based on proportions of categories the population fits into, with blue representing susceptible, red representing infected, and green representing recovered. An image for each time step will be generated, and the animation will simply cycle through the images. The way we organized our solution to the spatial S.I.R. model is also helpful here, since we simply need to have a for loop that runs through the time steps, grabbing the ratio for each variable at every point, and using the image function so that each variable's proportion combines to make a color on every grid point, where a larger proportion of a given variable will lead to it weighting more towards that color.

3.2 Implementation

3.2.1 RK4.m

This function RK4.m will implement the Fourth-Order Runge-Kutta algorithm described earlier. The inputs will be a function handle that is being solved, the span of time (start and end) over which it will be solved, and the initial conditions in column form. The outputs will be a vector of the time steps, along with a vector of the solutions.

```
function [t, y] = RK4(f, tspan, y0)
```

The step-size is hardcoded as $h = 0.1$ since that will not be changed during this process, and the amount of timesteps required is found by checking the amount of steps it takes to move from the initial time to final time given the step-size. Arrays are created for the solutions to be put into, and then a while loop is used simply implementing the equations defined earlier.

```
while i < nSteps %While within the total number of steps
    k1 = h*f(tk,yk); %Find the Runge-Kutta values
    k2 = h*f(tk + .5*h, yk + .5*h*k1);
    k3 = h*f(tk + .5*h, yk + .5*h*k2);
    k4 = h*f(tk + h, yk + h*k3);
```

After these calculations are done, y_k is found using the weighted average we defined earlier, along the t_k being found by simply adding the step size to the current time value. The values are then stored in their corresponding arrays which were created.

```
    tk = tk + h; %Find the new tk
    yk = yk + (k1 + 2*k2 + 2*k3 + k4)/6; %Find the new yk
    t(i + 2) = tk; %Store the new values
    y(i + 2,:) = yk;
```

The row location is $i+2$ due to the fact that the code uses a counting variable which begins as $i = 0$, and the first element of the array will be the initial conditions, and we are finding the next value. The count is then incremented, and will continue until $i = nSteps$ since at this point we have ran through $nSteps$ calculations. This is most clearly illustrated through a smaller example: if 5 steps are required of calculation, we will have ran through 5 steps after $i = 0, 1, 2, 3, 4$ which means that once $i = 5 = nSteps$ we should not run through the loop again. The function is now complete, and will output the t and y arrays when it is called.

3.2.2 dynamicsSIR.m

The function dynamicsSIR.m will create the differential equations that the RK4.m will solve. It will take the data in a vectorized state, the amount of rows and columns, and the model parameters as inputs. It will output a single column of the solutions of all 3 equations.

```
function dxdt = dynamicsSIR(x, M, N, alpha, beta, gamma)
```

This functions begins by reshaping the data so that it is more intuitive to work with. It will convert the data into a three dimensional set, where the first two dimensions are the coordinate position and the third dimension consists of susceptible, infected, and recovered (accordingly). A results array is also created to preallocate room for the solutions.

```
newX = reshape(x, [M, N, 3]); %Reshape the vectorized data to ...
    work with
finalresults = zeros(M,N,3); %Create a results array
```

Since the previously defined differential equations require a method of checking neighboring cells, another array is created which is the size of the 50x75 plane of the actual data, surrounded by 0's on every side, giving it two extra rows and columns. 1's are placed where the data would be. This will mean potential neighbors can be checked in a circular manner around a given data point by referring to this array, if it has a 1 then there is a neighbor in the checked location and the calculation can proceed, and if there is a 0 then no such neighbor exists and the calculation does not continue. This array is created by making an array with two extra rows and columns, and running a loop through the inner squares to change them to 1's.

```
weighteasy = zeros(M+2,N+2);
for i = 1:M
    for j = 1:N
        weighteasy(i+1,j+1) = 1;
    end
end
```

After this is created, a loop is done for every value which will calculate the weighted sum of infected neighbors, and then carry out the calculations of susceptible, infected, and recovered differentials in the cell being looked at. The loop begins by assigning W (the weighted sum) a value of 0.

```
for i = 1:M
    for j = 1:N
  W = 0;
```

From here, each potential neighbor is individually checked with if statements, starting at the upper left diagonal, and moving counter-clockwise around the target cell. The general process is as follows:

- For a given target cell (i, j)
- Check the value in the weighteasy array for the neighbor wanting to be checked (remembering that the data itself would be at $(i + 1, j + 1)$ in the weighteasy array due to the surrounding zeros. This means that the upper left diagonal neighbor of a target cell will actually be at (i, j) in the weighteasy array.
- If the neighbor is equal to 0, madd nothing to the sum.
- If the neighbor is equal to 1, take the value of the infected at that square in the newX array (remembering the shift) and multiply it by the corresponding weighting factor (1 for orthogonally adjacent, $\frac{1}{\sqrt{2}}$ for diagonally adjacent). Add this to the total W sum.
- Repeat the process for every neighbor.

This is done through a sequence of if loops. One example is given below, of checking for a neighbor directly below the target cell, and adding to the sum if one exists, or otherwise adding nothing.

```
if weighteasy(i+2,j+1) ≠ 0
    W = W + newX(i+1,j,2); %Where newX(:,:,2) is Infected
end
```

After this process is completed, and a total weighted sum is determined, then the derivative of each variable at the given location can be found by simply plugging our values into the differential equations defined in the theory. These are within the same loop, and the third dimensions of the finalresults array correspond to susceptible, infected, and recovered in that order.

```
finalresults(i,j,1) = -(beta*newX(i,j,2) + alpha*W)*newX(i,j,1);
finalresults(i,j,2) = (beta*newX(i,j,2) + alpha*W)*newX(i,j,1) ...
    - gamma*newX(i,j,2);
finalresults(i,j,3) = gamma*newX(i,j,2);
```

After this is done, the finalresults array is converted into a column vector. This is to make it compatible with our implemented solver. The output dxdt is set equal to the finalresults in column form, and the function is complete.

3.2.3 solveSpatialSIR.m

Our solveSpatialSIR.m function will serve as an incorporation of both RK4.m and dynamicsSIR.m, to ease in changing variables and allow for the output to be in an easier to interpret form. It will output a vector of the time steps, along with the solution of the system in a 4-dimensional matrix, in the previously preferred row, column, state variable, and time form. The inputs will be the time value at which the calculations should stop, the initial conditions of the system (in a three dimensional form), the model parameters (alpha, beta, and gamma), and the function handle that will be used to solve the differential equation.

```
function [t, x] = solveSpatialSIR(tFinal, initialConditions, ...
    alpha, beta, gamma, odeSolver)
```

The function begins by determining the number of rows and columns in the data by checking the size of the first two dimensions of initialConditions. A time span is then defined by starting at 0, and using tFinal to define the endpoint. The last adjustment that needs to be made is converting the initialConditions into a column so that it can be plugged into our solver.

```
[M,N,¬] = size(initialConditions);
tspan = [0 tFinal];
initialC = initialConditions(:);
```

The function we defined as dynamicsSIR.m is then used to create a function handle that will be used to model the system. This, along with the time span and our column of initial conditions are then plugged into the solver we passed into this function.

```
dSIRdt = @(t,x) dynamicsSIR(x, M, N, alpha, beta, gamma);
[t,sol] = odeSolver(dSIRdt, tspan, initialC);
```

The solution that comes out of this will be two dimensional, and we would prefer to analyze the data ina four dimensional form. To do this, we use the reshape command, allowing us to convert the form of the results. We then permute this adjusted result to order the state variables as Susceptible, then Infected, then Recovered, since that is the intuitive order (considering it is called an S.I.R. model).

```
solInt = reshape(sol, [length(t), M, N, 3]);
x = permute(solInt, [2, 3, 4, 1]);
```

This completes the function, with the outputs being the time array from the odeSolver, and the permuted 4-dimensional data we just adjusted.

3.2.4 plotTimeSeries.m

This function will take in the vector of time steps, the 4-dimensional solution generated by solveSpatialSIR.m, along with a coordinate position to generate a figure with three subplots, one representing the ratio of susceptible individuals in the population, the next representing the infected ratio, and the last being the recovered ratio.

```
function plotTimeSeries (t, X, x, y)
```

The code will repeat the same process three times, since three plots need to be generated in a very similar manner. First the plot is created, and hold is turned on. After this, a loop is used which iterates from 1 to the length of the time vector we passed in, which will allow us to mark the state variables value at every point in time. This is done by simply taking the value of the solution at the inputted x and y coordinates, for the desired state variable, at the point in time the loop is currently at. It is then scatter plotted, with the time variable divided by 10 to adjust it properly for our step size of 0.1.

```
for k = 1:length(t)
    S = X(x,y,1,k);
    scatter(k/10, S, 'b', '.');
end
```

The function then properly labels the graph, using the sprintf command to allow it to change the title depending on the coordinates chosen. This process is then repeated two more times, only changing the state variable examined. After this, the full plot is saved through the program and titled accordingly, using the sprintf command along with saveas.

```
imagefile = sprintf('time_series_%d_%d.png', x, y);
saveas(gcf, imagefile, 'png')
```

3.2.5 animate.m

This function will create an animation of how the ratios of each segment of the population (Susceptible, Infected, and Recovered) are changing through the use of RGB assignments to each condition. It only needs the full 4-dimensional solution from the solveSpatialSIR.m to be inputted, and will output the animation.

```
function animate(X)
```

The function begins by creating a size array of the solution input, and uses the first two values of this to find the rows and columns in the graph.

```
S = size(X); %Variable to find rows and columns
M = S(1); %Rows
N = S(2); %Columns
```

After this, a loop is ran which runs through every 10th time step between 1 and the final step. This is done by taking the size of the time array and defining a variable which will take every 10th element up to this size.

```
for t= 1:10:size(X,4) %Take steps by 10 to the end time
```

The loop begins by creating a color array, which is three dimensional and filled with zeros initially. The full susceptible, infected, and recovered values for every coordinate point are then defined, and the color array is filled using these values. They need to be filled in the dimensional order of R-G-B to ensure the colors correspond to the correct variables. This means the first value will be infected, the second recovered, and third susceptible.

```
    St = (X(:,:,1,t)); %Find Susceptible at that point
    It = (X(:,:,2,t)); %Find Infected at that point
    Rt = (X(:,:,3,t)); %Find recovered at that point
    color(:,:,3) = St; %Define St as blue
    color(:,:,1) = It; %Define It as red
    color(:,:,2) = Rt; %Define Rt as green
```

Once this is done, the image command can be used on the color array to display the state of the system at a certain time. The function will then continue to loop for each time step, creating a new image on top of the previous one, which will essentially be an animation.

3.2.6 Problem 2 Main Script

This is the script that will run all of the defined functions. It begins by loading the InitialValues.mat given in the problem statement. It also defines the values of alpha, beta, gamma, and tFinal as given in the reference document. The script will also make use of the tic, toc commands to determine the runtime required to solve the differential equations with ode45 and RK4.m. The script begins by running solveSpatialSIR.m with the RK4 function handle, and saves and prints the runtime as well.

```
tic
[t_rk, x_rk] = solveSpatialSIR(tFinal, initialConditions, alpha, ...
    beta, gamma, @RK4);
rkRuntime = toc;
fprintf('RK4 Runtime: %fs\n', rkRuntime);
```

The same process is repeated but using ode45 as the function handle:

```
tic
[t_ode, x_ode] = solveSpatialSIR(tFinal, initialConditions, ...
    alpha, beta, gamma, @ode45);
odeRuntime = toc;
fprintf('ode45 Runtime: %fs\n', odeRuntime);
```

After this, the plotTimeSeries.m function is used to find the plots of the ratios of susceptible, infected, and recovered individuals with the RK4 solution at the three given points: (1,1), (5,18), and (30,70). This requires the function to be called three separate times, and for three figures to be created. One example of the calling format is as follows (which is the figure for (1,1):

```
figure(1)
plotTimeSeries(t_rk, x_rk, 1, 1);
```

After repeating this two more times for the other points, a now just the animation needs to be created. This is done by creating a new figure, and calling the animate.m function with the RK4 solution as the sole input:

```
animate(x_rk);
```

After this, the main script is complete, and all desired outputs have been created.

3.3 Results and Discussion

The first result to examine from the main script is the runtime of the ode45.m and RK4.m solvers. Since they vary, two examples of the outputs from the script being ran will be given:

```
RK4 Runtime: 3.017161s
ode45 Runtime: 0.177900s
```

and another example:

```
RK4 Runtime: 2.748679s
ode45 Runtime: 0.168076s
```

While the exact times vary, it is clearly consistent that the ode45 solver is significantly faster than the RK4 solver. This is likely explained by two main causes: the built in ode45.m is likely more efficient in its implementation, in contrast to our RK4.m which involves quite a few statements that could likely extend the runtime, especially the multiple calculations that must be taken to find the weighted average. Another explanation for the difference in runtime is the number of steps that each solution takes. The RK4.m uses a consistent step size of 0.1s, resulting in 601 time values as it goes from a time of 0 to a time

of 60. The ode45.m model more intelligently determines the most efficient step size after every step, meaning it has a varying step size and only takes a total of 161 time values, approximately a quarter of the amount RK4.m needs to take. Both of these explanations paint a clear picture as to why ode45.m would have a faster runtime: it is doing less calculations, and the calculations it does are more efficient.

The next results to examine come from the calls to plotTimeSeries.m, which come in the form of three plots, each consisting of three subplots. They are as follows (beginning on the next page):

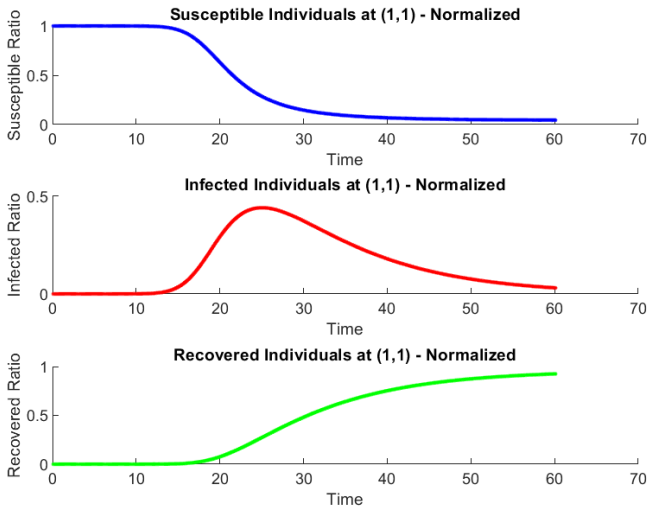

Figure 2: This graph displays the ratios of susceptible, infected, and recovered individuals at (1,1) over the full time span from 0 to 60.

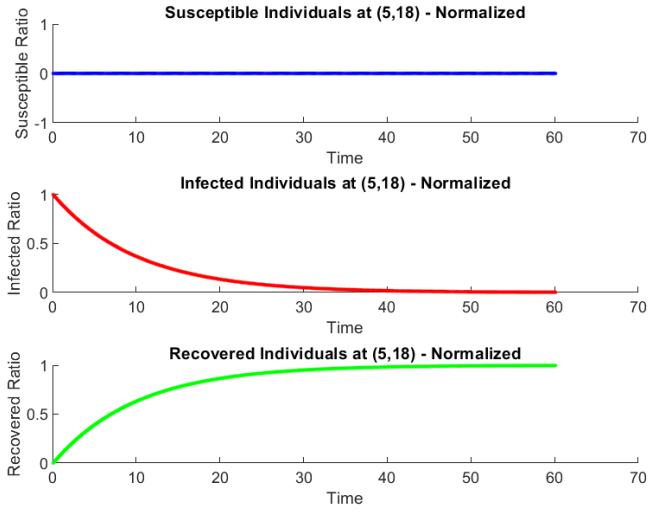

Figure 3: This graph displays the ratios of susceptible, infected, and recovered individuals at (5,18) over the full time span from 0 to 60.

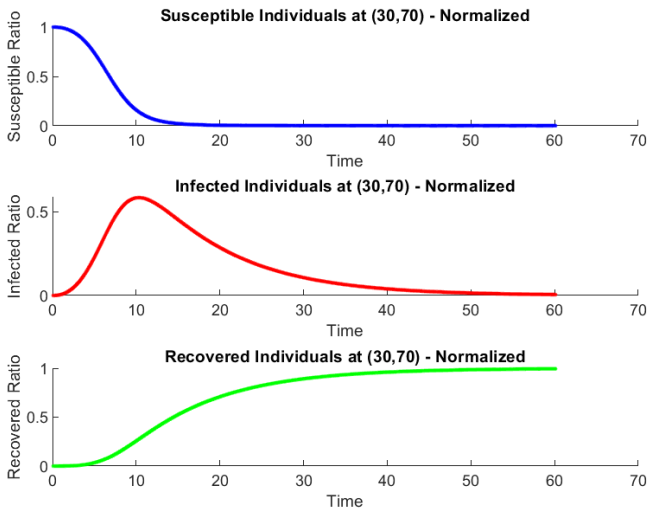

Figure 4: This graph displays the ratios of susceptible, infected, and recovered individuals at (30,70) over the full time span from 0 to 60.

These graphs all show the same general property: there is a limit to how many people can be infected within the same population at the same time. Even the population in Figure 3 that began fully infected very quickly started to decrease from that point. Both Figure 2 and Figure 4 display a population that began with no cases of COVID-19, and then rapidly began increasing in cases, but the number of infected only peaked at about half of the population before it began to decrease.

Another interesting property that is universally shared is the fact that all three populations in the end were nearly full of recovered individuals. The initial conditions only had a few populations infected (one of them being Figure 3), but even Figure 2 and Figure 4 by Time = 60 had been fully infected and fully recovered. This model likely does not account for herd immunity, or perhaps the spread is too strong within the same population, since typically an entire population would not get infected by an illness.

As a consequence of the previous property, it can be noted that there are essentially no more susceptible individuals in any of these locations. Since everyone has recovered, there are no more people able to catch the illness, and therefore it will no longer be able to spread, and all the lines on the Figures will continue to travel in a straight line (although Figure 2 may need a bit more time for the infected and recovered individuals lines to straighten out, as it took longer for the illness to spread there).

Finally, only examining Figure 2 and Figure 4, it is interesting to see the differences in their infected individuals ratios. Figure 4 at (30,70), with 8 neigh-

bors, shows a thin infected individuals ratio peak, and peaks much sooner in the process with slightly past half of the population being infected. Figure 2 at (1,1), in the corner of the plane with only 3 neighbors, shows a much wider infected individuals ratio peak, peaks much later in the process, and peaks to a lower value of slightly below half of the population.

The final result to analyze is the animation of the spread of COVID-19, included on the next pages of this document will be snapshots of the colored plane, at appropriate timesteps:

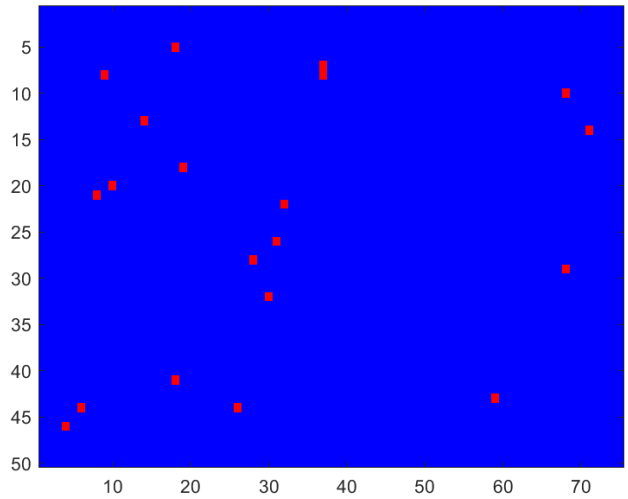

Figure 5: The image before any steps have been taken, the red squares representing the initially fully infected regions, while the blue squares are fully susceptible and have had no exposure to the virus.

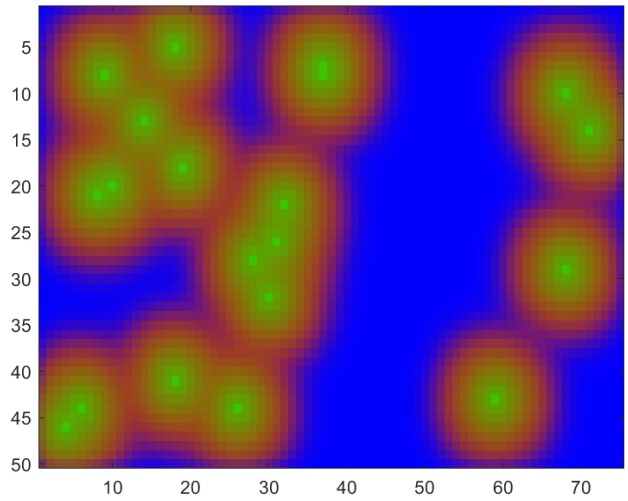

Figure 6: The image after 150 steps have been taken. There are still some regions fully blue with no exposure to the virus, but the places initially infected have begun bubbling out and infecting nearby areas. The spots initially fully infected are very green, likely near fully recovered.

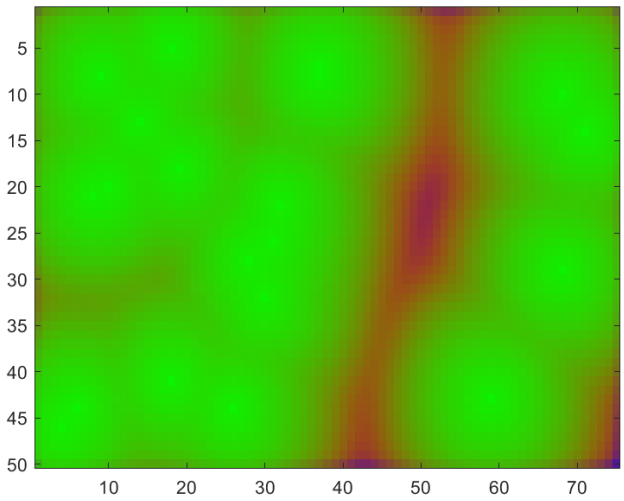

Figure 7: The image after 300 steps have been taken. A large majority of the regions are fully recovered at this point after having been fully infected. There is a narrow strip that is still majority infected.

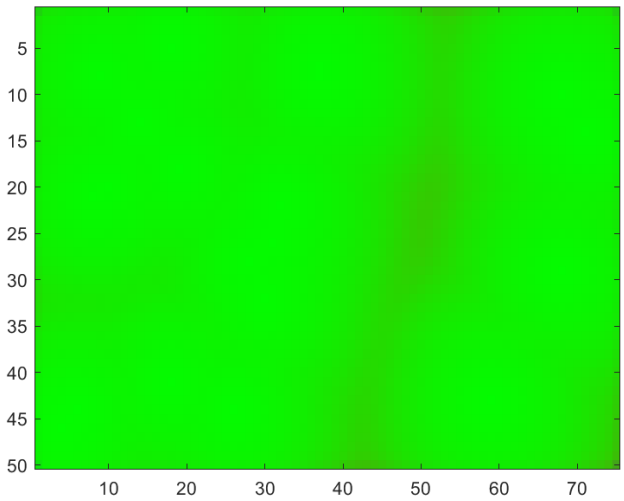

Figure 8: The image after 450 steps have been taken. Essentially all of the regions are now fully recovered, there is still a faint red tint to the narrow strip that was previously fully infected, but it is still majority green.

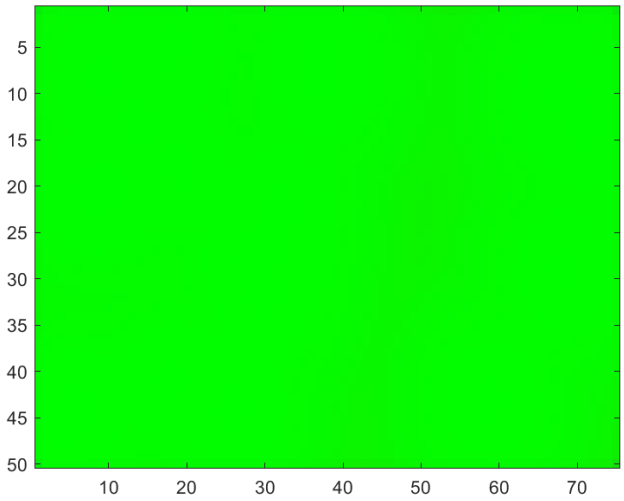

Figure 9: The image at the end after all 600 steps. It is fully green, meaning everyone has caught the virus, and everyone has recovered.

The animation as a whole shows that with only a few squares fully infected, the illness managed to spread to nearly every single member of the population, and it did not die out until nearly everyone has already recovered from their sickness. This shows the power of neighboring cells impact on the infection rates within other cells, since that is solely what caused this to spiral out of control.

The use of different colors representing the different state variables, and each square being an R-G-B composite of those colors serves as a valuable image of the situation in each position. It is clear which squares are in danger and likely about to rapidly increase in infection ratio, and it also makes it clear which squares are done with the worst of it, and will be improving in condition from that point on.

It is surprising how quickly the situation moved from Figure 5, with hardly any cases, to Figure 7, with most people having recovered from the virus already. It does make sense with the idea of exponential growth, though, and the larger and larger influence the weighting function will have.

4 Conclusion

The use of the weighting function served as an accurate tool to characterize the spread of an illness, and using normalized ratios to represent the state variables served as an insightful way to see the state of each coordinate point. Overall, the combination of created functions to solve and represent the spatial S.I.R. model was successful, allowing a view on how COVID-19 can spread in the real

world. This also simply explains how the COVID-19 pandemic came to be, since the use of these differential equations on this plane demonstrates the power of very few cases still being able to lead to a ton of cases in the end.

5 References

- "Final Project Assignment of M20 Introduction to Computer Programming", University of California, Los Angeles, 2020