Cory Chilton
UID: 605388291
CEE/MAE M20
December 12, 2020

Final Project Report: The Spread of a Virus

1 Main Component Analysis with PCA

1.1 Introduction

Principal Component Analysis (PCA) is a popular technique to reduce dimensionality, which is often used in machine learning in a variety of fields ("Final Project"). Although MATLAB has its own built in pca function, the goal in this problem is to create a new pca function that we are calling myPCA. We will use this function we created to analyze covid data from 27 different countries, which myPCA will do by finding the most important components of the data. This data analysis will then be shown on a biplot, and I will describe what this biplot means.

1.2 Model and Methods

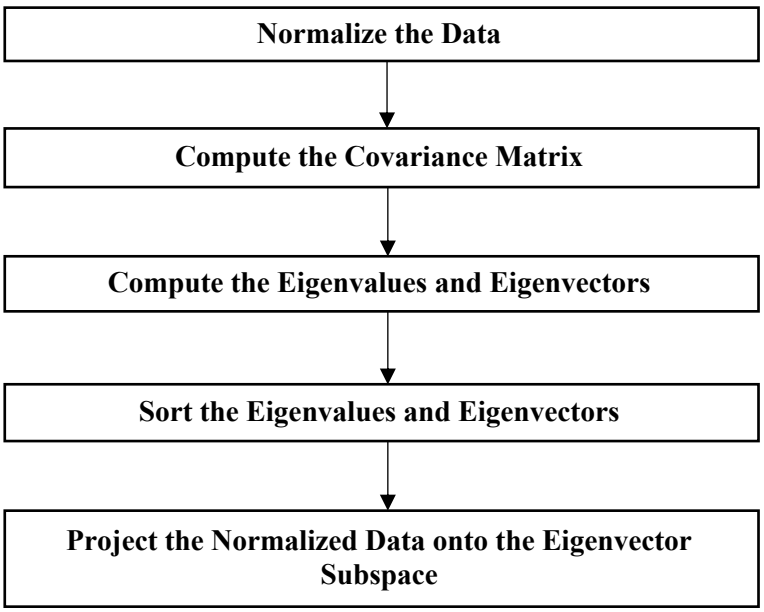
This problem is fairly straightforward as it only involves creating a single function: myPCA, calling this function in a main program, and creating a biplot based on the results from the myPCA call.

1.2.1 myPCA

The function myPCA only takes 1 input: "data", which is an nxp matrix representing the numerical parts of the dataset. It then has two outputs: "coeffOrth" and "pcaData". "coeffOrth" is a square pxp matrix that has "data's" eigenvectors as columns that have been sorted according to their corresponding eigenvalues. "pcaData" is also an nxp matrix that is a result of the "data" being normalized and projected onto its principal components. The function header looks as follows:

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

To create the myPCA function we need to follow these five steps:



("Final Project")

Step 1: Normalize the Data
This step is relatively simple and is just needed "to make the difference among the variables more comparable with each other" ("Final Project"). All you have to do is find the mean and standard deviation of each column. You then subtract the mean from its respective column and divide this column by its standard deviation. I then stored this new set of columns (matrix) into a variable normalizedData (same size as data) using a for loop, and the pseudocode is shown below.

```
Preallocate normalizedData (nxp)
For each column
   normalizedData ← (data column – mean of data column) / (standard deviation of data column)
end
```

Step 2: Compute the Covariance Matrix
In this step we just find the square covariance matrix from the normalizedData using MATLAB's built in cov function and store it in a new matrix C:

```
C = cov(normalizedData);
```

Step 3: Compute the Eigenvalues and Eigenvectors
Similar to the last step there is a built in MATLAB function called eig that can be used on the covariance matrix C, which then returns one matrix with the eigenvectors, and another matrix with the eigenvalues along the diagonal:

```
[eigenvectors,eigenvaluesDiag] = eig(C);
```

However, I wanted to have the eigen values in a single lined array so I used a for loop to put all of the values in a new array called eigenvalues, and here is the pseudocode:

```
Preallocate eigenvalues (1xp)
For each column j
   eigenvalues value j ← eigenvaluesDiag row j, col j
end
```

Step 4: Sort the Eigenvalues and Eigenvectors
Now we need to sort the eigenvectors according to their eigenvalues sorted in descending order to make the "coeffOrth" output. In order to do this, I used MATLAB's sort function, passing the absolute value of eigenvalues as the input, and telling it to sort in descending order. I then took two outputs from this function, the sorted eigenvalues, and the indexes of the sorted eigenvalues. However, the sorted eigenvalues are not actually needed for the rest of the problem so I replaced that part of the output with ~ to signify to MATLAB that I do not need it.

```
[~,indexes] = sort(abs(eigenvalues),'descend');
```

We then need to sort the eigenvectors based on these indexes, where the eigenvector with the largest eigenvalue comes first and the eigenvector with the smallest eigenvalue comes last. These sorted eigenvectors are then stored in "coeffOrth". The pseudocode of how to do this is shown below:

```
Preallocate coeffOrth (pxp)
For each column k
```

coeffOrth column k ← eigenvector column indexes(k)
end

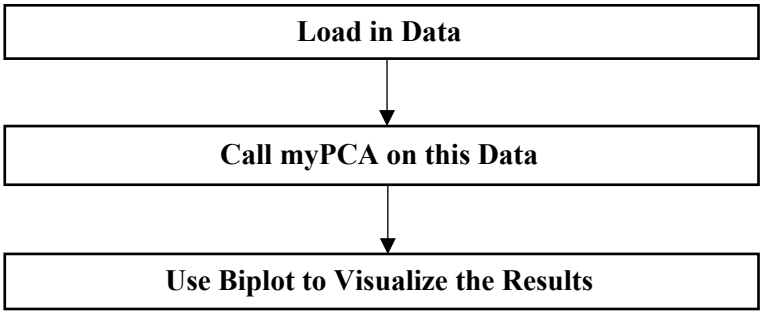
Step 5: Project the Normalized Data onto the Eigenvector Subspace
This step sounds complicated but is actually very straightforward. All we have to do to make the output "pcaData" is project the normalized data onto the "coeffOrth" matrix we made in the previous step. This is done by doing the matrix multiplication shown below.

```
pcaData = normalizedData * coeffOrth;
```

This makes sense because normalizedData is nxp and "coeffOrth" is pxp, and we want "pcaData" to be nxp, so it must be normalizedData * coeffOrth to obtain this size matrix and could not be the other way around. With **Step 1** through **Step 5** completed we have our two outputs "coffOrth" and "pcaData," are finished with myPCA, and ready to call it in our main function.

1.2.2 Main

Now that we have our myPCA successfully set up we can have it perform analysis for us. To make us do that there are 3 steps that need to be followed.



Step 1: Load in Data
In those problem we are going to be using the data from 'covid_countries.csv' that has been provided, but this could work for any data set. In order to load it in I used MATLAB's readtable function and stored it in a table called covid_countries_table. I also added a parameter to keep the variable names from the table loaded in the same, as can be seen below.

```
covid_countries_table = readtable('covid_countries.csv', ...
    'VariableNamingRule', 'preserve');
```

Step 2: Call myPCA on this Data
Now that we have the data loaded in, we can call myPCA on it. However, myPCA is only for numerical data so we cannot include things such as variable names, or country names that we loaded in from 'covid_countries.csv'. To do this we use {} to call data from the table because take only the data and not the variable headers, but this still includes the countries so just have to change the indices accordingly.

```
[coeffOrth,pcaData] = myPCA(covid_countries_table{:,3:end});
```

Step 3: Use Biplot to Visualize the Results

We now use MATLAB's biplot function in order to show all of our variables and data based on the principal components found using myPCA. Because myPCA actually returned all of the components we need to call biplot on only the first two, so we use the appropriate indices.

```
biplot(coeffOrth(:,1:2), 'Scores', pcaData(:,1:2), 'Varlabels', categories);
```

This biplot call visualizes "both the orthonormal principal component coefficients for each variable and the principal component scores for each observation in a single plot" ("Biplot"). In our situation the "orthonormal principal component coefficients for each variable" is our coeffOrth, and the "principal component scores for each observation" is our pcaData. We also added labels for our variables that is specified by 'Varlabels' and we inputted categories, which is just a cell array of all of the variable headers. A cell array was used so that categories could hold variable name strings of different lengths.

```
categories = {'Infections', 'Deaths', 'Cures', 'Mortality Rate', ...
    'Cure Rate', 'Infection Rate'};
```

1.3 Calculations and Results

When we run the main function for this problem the following plot is created:

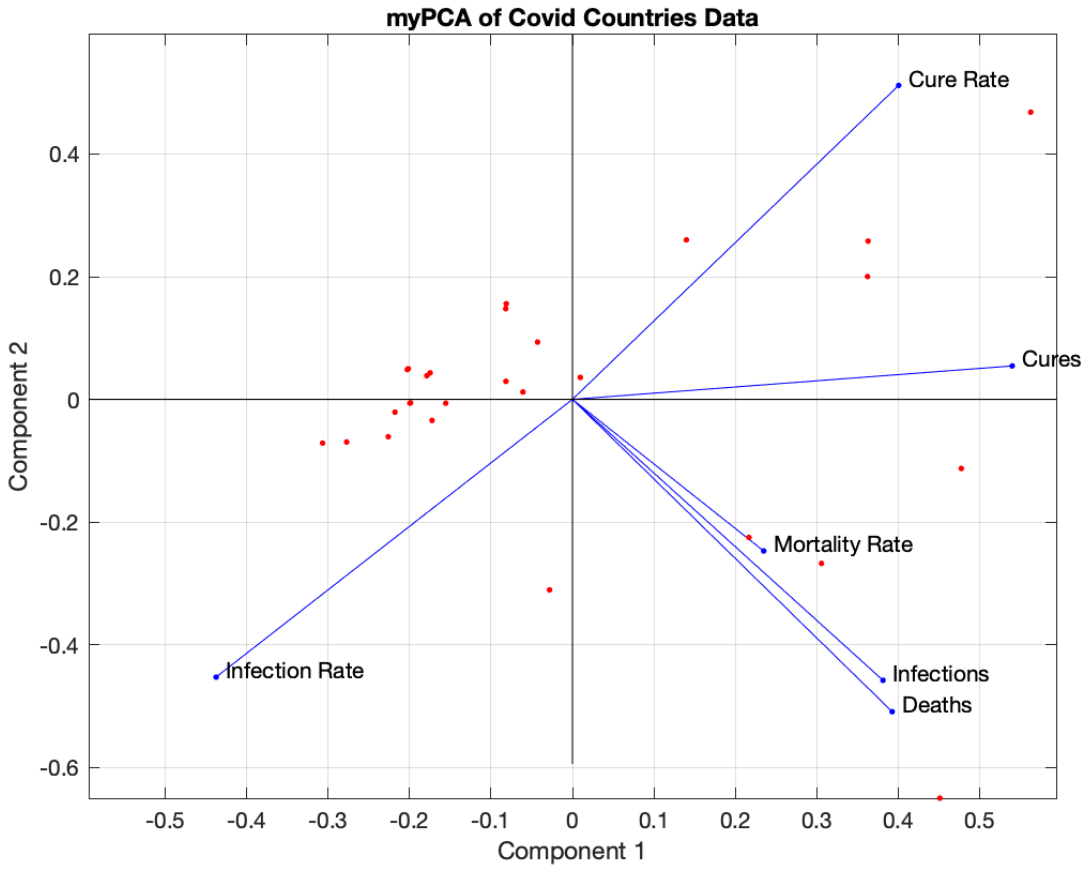


Figure 1: Biplot created from the result of running myPCA on 'covid_countries.csv'. The red dots are created from pcaData and the blue vectors are created from coeffOrth.

1.4 Discussion

It is not apparently obvious what the biplot shown in **Figure 1** is supposed represent so I will break it down. The axes of the biplot, Component 1 and Component 2, are the principal components, or the two most important components found from myPCA. Then, each blue vector (from "coeffOrth") represents how much weight each variable has on each principal component, based on its project value onto that principal component (Ngo). For example, cures has a fair amount of influence over Component 1, but has very little influence over Component 2. The angle between two vectors also tells you how much they correlate with one another (Ngo). For example, Infections and Deaths are positively correlated, Infection Rate and Cure Rate are negatively correlated, and Deaths and Cure Rate do not have much of a correlation at all. Also, if different variables have positive or negative coefficient for a certain component that can distinguish between them. For example, Component 1 distinguishes between a high values for Deaths and a low values for Infection Rate, and low values for Deaths and high values for Infection Rate. The red dots then show the data ("pcaData") which can be interpreted as what score each data point has for each of the principal components ("Biplot"). "For example, points near the left edge of this plot have the lowest score for the first principal component" ("Biplot"). Finally, only the relative location of each point matters because they are scaled based on the maximum score value and the maximum coefficient length ("Biplot").

2 Solving the Spatial S.I.R. Model

2.1 Introduction

A spatial SIR model looks at the spread of disease for a fixed population, where S(t) represents the number of susceptible individuals, I(t) represents the number of infected individuals, and R(t) represents the number of recovered individuals, where all three have to add up to a constant number over time ("Final Project"). In our case they will always add up to 1 because we are just looking at the ratio of the population that is susceptible, infected, and recovered. In this problem we will be looking at a bunch of smaller populations within one large population. Each smaller population is represented by a square on a large grid that represents the entire population. These smaller populations each change over time according to governing equations that take into account their neighboring small populations (squares), which can be seen below:

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

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

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

Equations 1-3: Governing equations for a spatial SIR model ("Final Project")

In these equations β represents the contact rate, α represents the spatial contact rate, γ represents the infectious rate, and W is a weighting function that weighs the neighbors of the square on the grid being looked at ("Final Project"). The weighting function only takes into account the square's direct neighbors and works as follows: anything touching the square diagonally is weighted as $\frac{1}{\sqrt{2}}$ and anything touching the square on the top, bottom, left or right is weighed as 1. If a neighbor is outside of the grid it is

assumed to count as zero in the weighting function. The diagram below shows how different squares are weighted based on their location, if the square being processed is in the middle.

W(-1,1)	W(0,1)	W(1,1)
W(-1,0)	W(0,0)	W(1,0)
W(-1,-1)	W(0,-1)	W(1,-1)

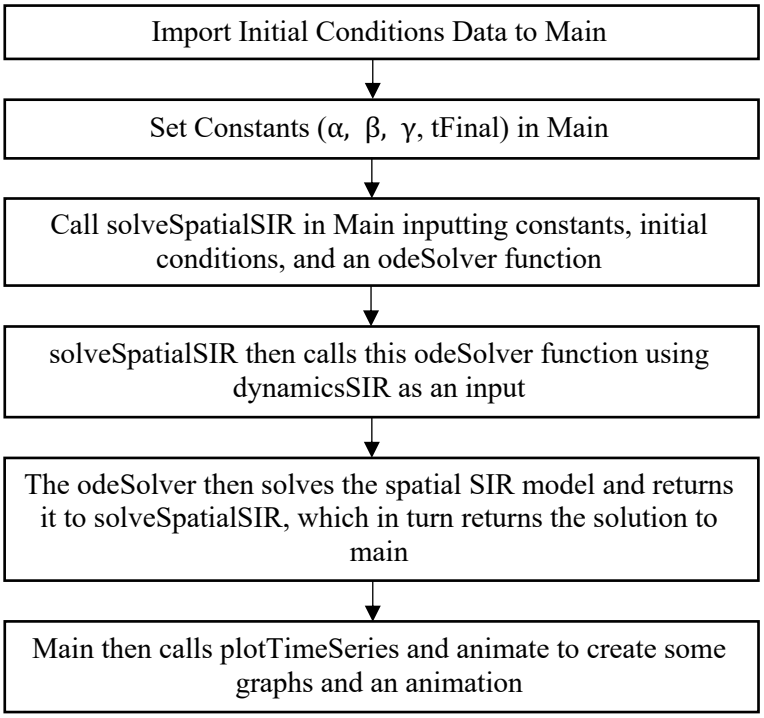
$\frac{1}{\sqrt{2}}$	1	$\frac{1}{\sqrt{2}}$
1	0	1
$\frac{1}{\sqrt{2}}$	1	$\frac{1}{\sqrt{2}}$

Figures 2 and 3: The locations of the coordinates on the left in **Figure 2,** and the values of W(i,j) on the right in **Figure 3** ("Final Project")**.**

The goal in this problem is, when given the initial conditions, to solve spatial SIR model over time and make an animation that shows how the number of individuals who are susceptible, infected, and recovered changes over time in different parts of the grid. We also want to be able to show how the number of susceptible, infected, and recovered individuals also changes for each small population if given the specific coordinates of that small population on the grid.

2.2 Model and Methods

Solving the spatial SIR model is no small feat and is going to take multiple functions to pull off. These functions include RK4, dynamicsSIR, solveSpatialSIR, plotTimeSeries, animate, and main. We are going to dive into each of these functions individually and take a look at how they work, but first it's important to see how we want the completed program to work because that is what we are heading towards. Below is a flowchart of how the program works:



2.2.1 RK4

RK4 implements the Fourth-Order Runge-Kutta Algorithm and is an ODE solver, meaning that it can be used to solve ordinary differential equations, such as our governing equations that were introduced above. It takes 3 inputs: f, tspan, and y0; f is the function handle of the differential equation we are solving and takes t (time) and y (values) as inputs; tspan is the time period for the simulation, which is a 1x2 array that has the start and end times; y0 is a matrix of the initial conditions for the differential equation. R4 then has 2 outputs: t and y; y is the solution of the differential equation and is a Txn matrix, with T being the number of time steps, and n is the number of values in y0; t is the corresponding time sequence to y and is a Tx1 vector ("Final Project"). To start RK4 all of the variables are initialized, constants are set, and matrices are preallocated.

```
h = 0.1; % Step size
nSteps = (tspan(2) - tspan(1)) / h + 1; % Total number of steps
```

Our step size h is set to 0.1, which could be set to anything, a smaller h giving you better resolution. The total number of steps is then determined from the tspan input. We preallocate the t output array to be an array of zeros of size Tx1, and we preallocate the y output matrix to be an array of zeros of size Txn.

```
t = zeros(nSteps,1); % Preallocating the time array
y = zeros(nSteps, n); % Preallocating the y array
t(1) = t0; % Setting the first value of the time array to starting time
y(1,:) = y0(:)'; % Setting the first row of y to y0
```

The first value of t is then set to the starting time which is the first value of tspan. The first row of y is then set to y0, but since y0 is a matrix, we need to vectorize it by using (:) which turns y0 into a column, so it is then turned into a row by using '. Now that we have everything preallocated and initialized it is time to start actually solving the differential equation. The differential equation is solved using the fourth order Runge-Kutta Method which uses four different points to solve the differential equation for each time step, and whose pseudocode can be seen below.

$k \leftarrow 2$
while $k \leq$ nSteps do
  $k_1 \leftarrow hf(t_{k-1}, y_{k-1})$
  $k_2 \leftarrow hf(t_{k-1} + h/2, y_{k-1} + \text{row } k_1/2)$
  $k_3 \leftarrow hf(t_{k-1} + h/2, y_{k-1} + \text{row } k_2/2)$
  $k_4 \leftarrow hf(t_{k-1} + h, y_{k-1} + \text{row of } k_3)$
  $t_k \leftarrow t_{k-1} + h$
  $y_k \leftarrow y_{k-1} + (\text{row } k_1 + \text{row } 2k_2 + \text{row } 2k_3 + \text{row } k_4) / 6$
  $k \leftarrow k + 1$
end ("Final Project")

Once this code has finished the t vector and the y matrix will be completed filled. For each step we use row k_i instead of just k_i because we do not know what shape f is going to return, but in order for it to be added to y we need it to be a row vector. This can actually be done in the code by doing k(:)', which first vectorizes k into a column and then transposes it to make a row.

2.2.2 dynamicsSIR

The function dynamicsSIR computes the rate of change of the model, and is the input function f that is given to RK4 in this problem. It takes 5 inputs: x, M, N, alpha, beta, and gamma; M is the number of rows in the grid of populations; N is the number of columns in the grid of populations; x is the vectorized state of all these populations and has length 3MN; alpha, beta, and gamma the model parameters that we previously defined. It makes sense that x has length 3MN because there are MN smaller populations on the grid, and each smaller population has 3 values assigned to it: the ratio of susceptible, infected, and recovered individuals. dynamicsSIR then has a single output: dxdt; dxdt is the vectorized time derivative of the state. We will calculate dxdt according to **Equations 1-3.**
We start by remaking the grid using MATLAB's reshape function so that it is easier to work with. We also need to initialize our variables and preallocate our matrices.

```
grid = reshape(x,M,N,3)
```

We can think of this grid as three 2D matrices stacked on top of each other, each having a size of MxN. The front 2D matrix of the grid is the susceptible ratios, the middle matrix of the grid is the infected ratios, and the back matrix of the grid is the recovered ratios, so these are assigned accordingly.

```
S = grid(:,:,1); % Susceptible: initalized to front plane of grid
```

We then preallocate three MxN matrices full of zeros to store values for the rate of change of the susceptible ratios, the rate of change of the infected ratios, and the rate of change of the recovered ratios, which will eventually become dxdt after we have filled these matrices.

```
dSdt = zeros(M,N);
```

We can now work to fill these three matrices (dSdt, dIdt, and dRdt) by running through every square in the grid, which can be done by using stacked for loops. Inside of these for loops we start by finding $\sum_{i,j} W(i,j) I_{x+i,y+j}(t)$ for the current square, which I call sumWI. Then once we have found sumWI for the square, dSdt, dIdt, and dRdt are calculated for that point according to **Equations 1-3** and the value is stored in the appropriate place in each matrix. In this case we are only looking at a single time, so we are only calculating dSdt, dIdt, and dRdt for a single time. The pseudocode for this code segment is shown below:

For all columns xx
  For all rows yy
    Calculate sumWI
    $dSdt_{xx,yy} \leftarrow -(\beta I_{xx,yy} + \alpha \text{sumWI}) S_{xx,yy}$
    $dIdt_{xx,yy} \leftarrow (\beta I_{xx,yy} + \alpha \text{sumWI}) S_{xx,yy} - \gamma I_{xx,yy}$
    $dRdt_{xx,yy} \leftarrow \gamma I_{xx,yy}$
  End
End

Calculating sumWI can be quite tricky because you have to remember that the weighting function will take different squares into account depending on whether the square we are currently looking at is on the edge, in a corner, or in the middle of the grid. I overcame this problem by using a series of if statements that will only take a neighboring square into account if that square exists based on where the current square is located. For example, if a square is on the top edge of the grid the weighting function will not take into account any squares above it, but will take into account the other 5 squares. A pseudocode example of this is shown below.

If square is not on the top edge

sumWI ← sumWI + Infected ratio of square above it
End
If square is not on the left edge
sumWI ← sumWI + Infected ratio of square to the left of it
End
If square is not on the top edge and not on the left edge
sumWI ← sumWI + $\frac{1}{\sqrt{2}}$ * infected ratio of square diagonally up and to the left of it
End

This is repeated for the 5 other possibilities so that we have a complete sumWI. I think it would also be helpful to see how this actually works in the code, so you can see the code for adding the square above it to sumWI below.

```
if yy < M % Top box
    sumWI = sumWI + I(yy+1,xx);
end
```

We can see that the if statement checks that the y coordinate is less than the maximum y coordinate (M) so we can add the infected ratio of the square above the square we are looking at to sumWI confident that it will not be out of bounds. Also, we can see that the y coordinate comes first in I, and this is because when calling things from a matrix the first index indicates the row and the second index indicates the column.

2.2.3 solveSpatialSIR

Now that we have done all of the grunt work in RK4 and dynamicsSIR, this function is just bringing those together, and is the function that actually solves the spatial SIR model for our given initial conditions. It also helps us in comparing how fast RK4 can solve the equations in comparison to how fast ode45 can solve the equations, something we want to look at later. It has 6 inputs: tFinal, initialCondition, alpha, beta, gamma, and odeSolver; tFinal is the end time for the simulation; initialCondition is an MxNx3 matrix that sums to 1 in the third dimension; alpha, beta, and gamma are the model parameters previously defined; odeSolver is a function handle for an ode45-compatible solver ("Final Project"). It also has 2 outputs: t and x; t is a vector of all of the time steps and x is an MxNx3xlength(t) matrix that represents the state vs. time ("Final Project").
All this program does is run the odeSolver (ode45 of RK4) using dynamicsSIR as the f input. In order to use dynamicsSIR we need to find M and N, which we can easily do from initialCondition.

```
M = numel(initialCondition(:,1,1));
N = numel(initialCondition(1,:,1));
```

Then we need dynamicsSIR in a form so that it only takes inputs (t,y), because that is what RK4 is going to input when calling f (f is dynamicsSIR in this case). To do so we create a new function dSIRdt that is the same as dynamicsSIR but only takes t and y.

```
dSIRdt = @(t,y) dynamicsSIR(y, M, N, alpha, beta, gamma);
```

Even though dynamicsSIR does not actually have a t input this does not matter as long as the function we are putting into the odeSolver takes two inputs. Now that we have dSIRdt the only thing we need to have all of the inputs for an odeSolver is the tspan, which can easily be created by making a 1x2 matrix with the values 0 and tFinal. Now that we have everything, we can solve the spatial sir model using the odeSolver.

```
tspan = [0 tFinal];
[t,x] = odeSolver(dSIRdt,tspan,initialCondition);
```

The variable t is now complete as it is a vector of the time steps, but x is not because it is not in the shape MxNx3xlengh(t). To get it into this form I used MATLAB's reshape function but had to use the transpose of x instead of just x, because reshape does things column wise.

```
x = reshape(x',M,N,3,length(t));
```

2.2.4 plotTimeSeries

The function plotTimeSeries plots and saves the ratio of susceptible, infected, and recovered in a smaller population at a designated spatial coordinate (x,y). It takes inputs t and X; t is a vector of the time steps and X is the solved spatial SIR model from solveSpatialSIR. X is going to be a "MxNx3xlength(t) matrix, where each point in the MxN space corresponds to a local S.I.R. model with states whose values are between 0 and 1. This 3D matrix repeated for each time step, making it a 4D matrix" ("Final Project"). plotTimeSeries has no outputs, it just creates plots and shows them on the screen. Since we want to plot S, I, and R over time for a specific square, I stored that part of X in each variable S, I, and R for all of the time steps, and then vectorized each one of them.

```
S = X(x,y,1,:); % x,y in the front plane
S = S(:);
```

I then created a new figure with a custom pointer just for fun, which I created using a 16x16 matrix where NaN is a transparent pixel, 1 is a black pixel, and 2 is a white pixel. Then each of S, I, and R are plotted on the same figure using MATLAB's subplot function. I changed the x axis of each subplot to only be from 0 to 60 because our time is from zero to 60 and added aesthetic adjustments including title and labels. I also added a title over the entire figure using MATLAB's sgtitle function.

```
subplot(3,1,1)
plot(t, S, 'b', 'Linewidth', 2)
sgtitle(sprintf('SIR Model at Position (%d,%d)',x,y))
```

Finally, the figure is saved programmatically using saveas, and is saved with the specific x and y inputs using sprintf.

```
saveas(h1,sprintf('time_series_%d_%d.png', x, y));
```

2.2.5 animate

The function animate shows a changing image of the grid over time, with the ratios of susceptible, infected, and recovered individuals for each small square in the grid changing. It has one input, X, which is the same as it was in plotTimeSeries. Similar to plotTimeSeries it also has no outputs, but instead shows the animation on the screen. In this problem each small square has a color that is dependent on the ratios of susceptible, infected, and recovered individuals. Red represents infected, green represents recovered, and blue represents susceptible. To start I need to find the total number of time steps, which can be done from taking the fourth output of the size the input variable X.

```
[~,~,~,T] = size(X);
```

The next step is to make a matrix with all of the colors in it. We start by preallocating a colors matrix full of zeros that is the same size as X. Then each of the 2D matrices over time within the color array is set to susceptible, infected, or recovered. The front is set to infected, the middle is set to recovered, and the back is set to susceptible. This is because the image function we are about to use takes in an MxNx3 matrix, where "the coordinates (:,;,1) of the 3-D array contain the red components, (:,:,2) contains the green components, and (:,:,3) contains the blue components," so we need the infected front to be represented by red, the recovered middle to be represented by green, and the susceptible back to be represented by blue ("Final Project"). Also, we do not need to change the values at all for the colors matrix because "the input of image is the normalized intensity, between 0 and 1," and our colors matrix is already normalized ("Final Project").

```
[~,~,~,T] = size(X); % Finds total number of time steps
colors = zeros(size(X)); % Preallocates colors array
colors(:,:,1,:) = X(:,:,2,:); % red for infected
colors(1,1,1,:) = X(1,1,1,:); % time steps added to colors
```

Now we use a for loop over all of the time steps to animate our solved spatial SIR model. We only want to display the model every 10 time steps and to pause for .1 seconds between each time it displays the model. The pseudocode for this for loop is shown below:

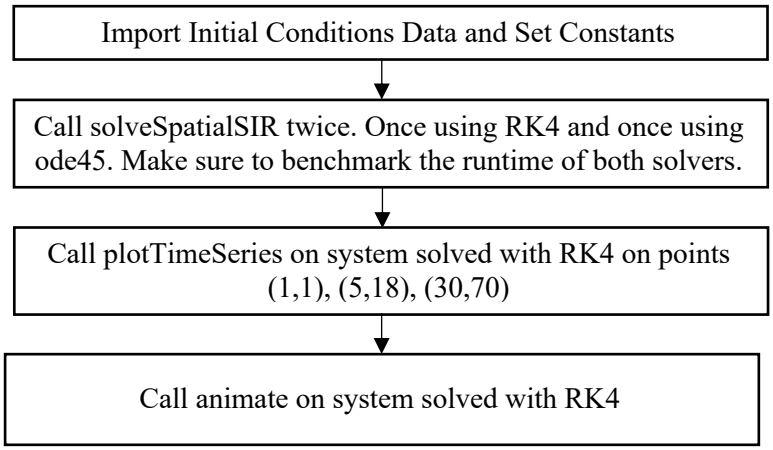
```
For each time step t
   If it is a 10th time step
      Display model at time t
      Pause for .1 seconds
   End
End
```

The display model part of the pseudocode is done using MATLAB's image function and inputting the colors array at time t. Pause is then done using MATLAB's pause function.

```
image(colors(:,:,:,t)); % Display the colors array
pause(.1); % Pause for .1 seconds
```

2.2.6 Main

Now that all of the other functions are completed, we can use main to create plots and an animation. To do so we will follow the 4 steps shown in the flowchart below.



Step 1: Import Initial Conditions Data and Set Constants
I first import initialConditions (given) using MATLAB's load function. Then I set the constants alpha, beta, gamma, and tFinal as given. These constants are dependent on the situation we are looking at the spatial SIR for.

```
load('initialValues.mat')
alpha = .1;
```

Step 2: Run solveSpatialSIR for ode45 and RK4
We want to compare the run time for ode45 against the runtime for RK4 so we call solveSpatialSIR twice. In order to find the run time, I used the built in tic toc commands before and after running solveSpatialSIR for each ode solver, which then displays the elapsed time to the Command Window.

```
tic
[tRK4,xRK4] = solveSpatialSIR(tFinal,initialConditions,alpha,beta, ...
    gamma,@RK4); % uses RK4
toc
```

Step 3: Plot using plotTimeSeries
We then create plots for three different points on the system solved with RK4. These points are (1,1), (5,18), (30,70).

```
plotTimeSeries(tRK4, xRK4, 1 , 1 )
```

Step 4: Animate
In this step we then create a new figure and run the animate function on the system solved with RK4. I also added a custom pointer again for fun like I did in plotTimeSeries, but this time it is 32x32 and is meant to look like the DJ Marshmello.

```
figure('Pointer', 'custom', 'PointerShapeCData', marshmello);
animate(xRK4)
```

2.3 Calculations and Results

When main is run, we get the following graphs.

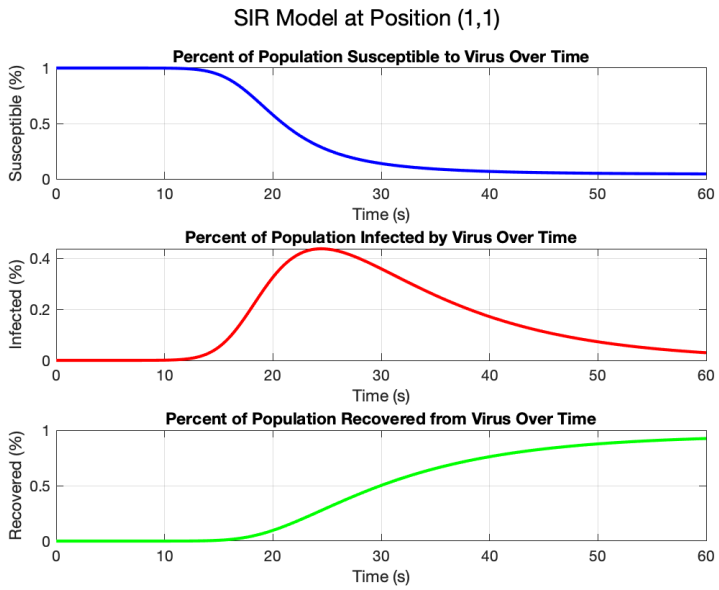


Figure 4: Shows the result of plotTimeSeries being called at the point (1,1) for the data from the system solved by RK4.

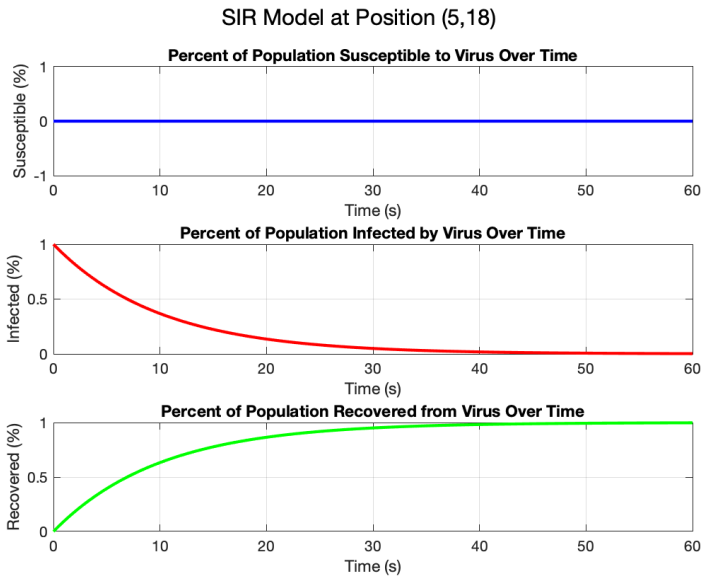


Figure 5: Shows the result of plotTimeSeries being called at the point (5,18) for the data from the system solved by RK4.

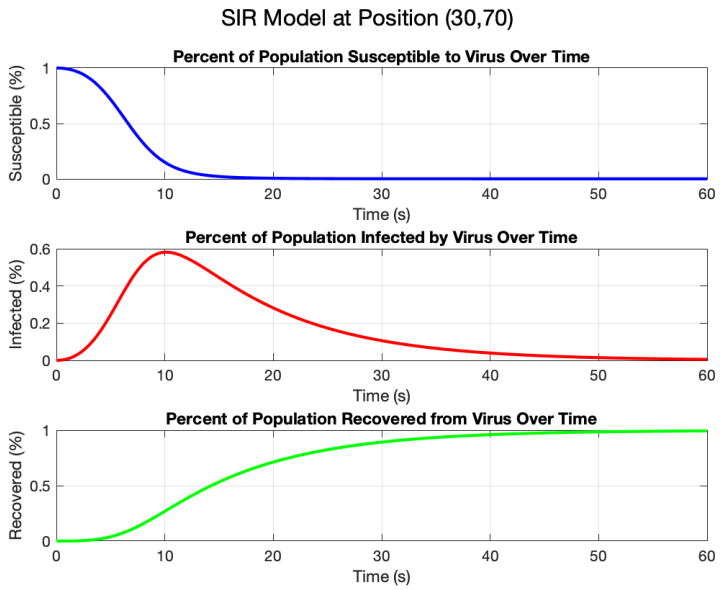


Figure 6: Shows the result of plotTimeSeries being called at the point (30,70) for the data from the system solved by RK4.

In **Figure 4** we can see that the ratio of susceptible individuals starts at 1 and then drops to zero over time, the ratio of infected individuals starts at zero, goes to .4, and then goes back down towards zero, and the ratio of recovered individuals starts at zero and goes to 1 over time. In **Figure 5** we can see that the ratio of susceptible individuals starts at zero and stays there, the ratio of infected individuals starts at one and goes to zero over time, and the ratio of recovered individuals starts at zero and goes to 1 over time. In **Figure 6** we can see that the ratio of susceptible individuals starts at one quickly drops to zero, the ratio of infected individuals starts at zero, goes up to .6, and then goes to zero over time, and the ratio of recovered individuals starts at zero and goes to 1 over time.
When main is run we also get an animation, of which some frames are shown below.

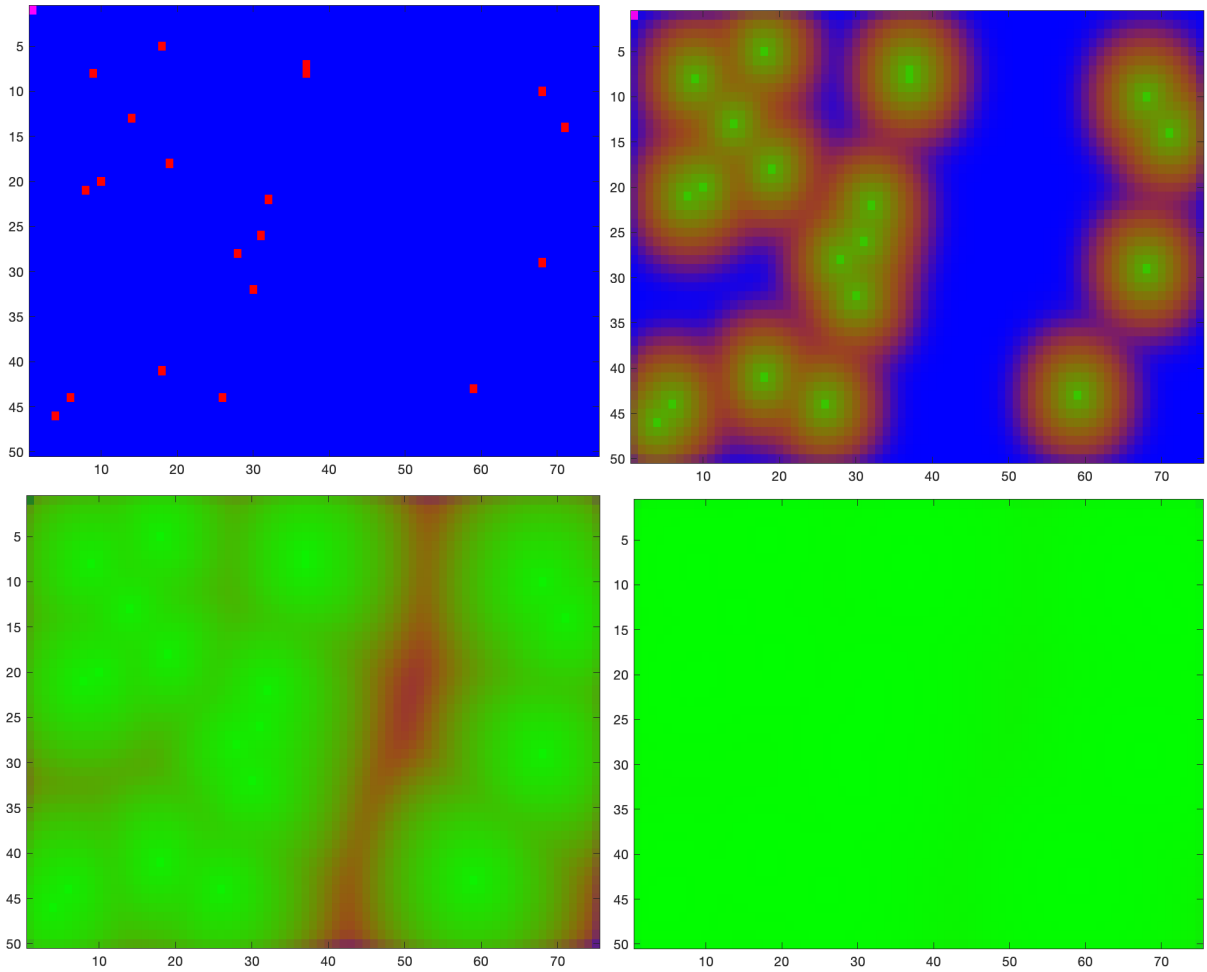


Figure 7: Four different frames from when the animate function is called on the system solved with RK4. The top left frame is from t=0, the top right frame is from t = 150, the bottom left frame is from t = 300, and the bottom right frame is from t = 600.

We can see that at t=0 the grid is mostly blue with a few red dots. Then at t=150 we can see that the amount of red is increasing, the amount of blue is decreasing, and green is starting to appear at the middle of the growing red circles. At t=300 the blue is completely gone and there is mostly green with some red left. Finally, at t=600 the grid is completely green.

When main is run there is also a message printed to the Command Window:

```
ode45: Elapsed time is 0.277260 seconds.
RK4:   Elapsed time is 0.975867 seconds.
```

2.4 Discussion

We can see in **Figure 7** that we start with most of the smaller populations being full of susceptible individuals with a few of the populations being full of infected individuals. Then as time goes on the number of infectious individuals in each population spreads in a circular fashion, which makes sense given how the spatial weighting system works. Then as populations have been infected for a while they start to recover. That is why as the red circles are growing outward, the middle of the circles starts to

become green, and then that green spreads more and more until every population on the grid is almost completely recovered as seen at t=600.

Figure 4 is shown at a coordinate that starts with all susceptible individuals. Then once a red "infectious circle" reaches them they start to quickly become more and more infected, so they must less and less susceptible. Then once there are infected people, they start to recover so the amount of recovered people increases causing the number of infected people to decrease until all of the people are recovered. **Figure 5** is shown at a coordinate that starts with all infected individuals. The number of susceptible individuals stays at zeros the whole time because individuals in this population are only ever going to be infected or recovered and will never become susceptible again. **Figure 6** is also shown at a coordinate that starts with all susceptible individuals; however, it starts closer to an infected point than **Figure 4.** I can tell this because this because the ratio of infected individuals spikes much sooner in **Figure 6** than it does in **Figure 4,** which means that in **Figure 6** the individuals recover sooner than in **Figure 4,** but individuals in **Figure 5** recover the soonest out of all of them because all of the individuals started infected in **Figure 5.**

We can also see in the Command Window that it took RK4 approximately 3.5 times longer to solve the spatial SIR system than ode45. This is because ode45 has been highly optimized to run as fast as possible. For example, ode45 uses fewer timesteps than RK4 because it determines its own timesteps based on some default tolerances. In fact, in this case ode45 uses 161 time steps, when RK4 uses 601 timesteps, about 3.7 times more than ode45. Although this is not necessarily the only cause, it does seem to correlate well.

Overall, in this problem we created an interesting and relevant way to look at the way a disease spreads through a larger population by spreading through smaller populations full of individuals.

References

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

“Biplot.” *Analyze Quality of Life in U.S. Cities Using PCA - MATLAB & Simulink*, www.mathworks.com/help/stats/quality-of-life-in-u-s-cities.html.

Ngo, Linh. “How to Read PCA Biplots and Scree Plots.” *BioTuring's Blog*, 21 Sept. 2020, blog.bioturing.com/2018/06/18/how-to-read-pca-biplots-and-scree-plots/.