M20 Final Report

Matthew Lacaire

December 9 2020

1 Introduction

The following report demonstrates several computational capabilities of MATLAB in modeling disease spread, animating results, and performing normally complex linear algebra operations such as PCA numerically. This report is broken up into two distinct parts, each using two different sets of data: The first is an implementation of the Principal Component Analysis (PCA) method to determine relationships between common disease-pertinent statistics, and the second utilizes higher-dimensional matrices and ODE solvers to visually model disease spread based on an idealized model. Throughout, there is extensive utilization of MATLAB's integrated matrix manipulation functions.

2 Principal Component Analysis (PCA) Implementation

Introduction

In this section, I use the Principal Component Analysis method to determine the general "directions" of data in two-dimensional space by identifying and sorting principal component vectors. The data I use is coronavirus data from 27 countries located in various regions around the world, and data measured is infections, deaths, cures, mortality rates, cure rate, and infection rate.

To analyze this data, I perform the PCA process and select two principal components so that my results can be viewed and graphed.

Walkthrough

I begin my actual implementation of this process by creating two files, a function `myPCA.m`, and a main script `project_205288439_p1.m`.

In `project_205288439_p1.m`, I import data from covid_countries.csv by using the `csvread()` function built in to MATLAB, and I assign this to the variable *M*. This is the 27x6 matrix we will use to perform calculations. Rows are given countries' six points of data, and columns are data of a given type for each country. In the line

```
[coeffOrth, pcaData] = myPCA(M);
```

I pass M into the myPCA function. In `myPCA.m`, I perform the actual computation required to provide a PCA computation with two principal vectors returned. Pseudocode is below:

```
$\Rightarrow Input\ matrix\ M$
$\Rightarrow Determine\ number\ of\ columns,\ rows$
$\Rightarrow Initialize\ normalized\ data\ array$
$\Rightarrow for\ each\ column:$
    $find\ mean\ of\ column,\ subtract\ mean\ from\ each\ cell,\ and\ divide\ by\ standard\ deviation\ of\ column$
$\Rightarrow populate\ normalized\ data\ array\ with\ these\ vectors$
$\Rightarrow use\ integrated\ cov()\ function\ to\ find\ covariance\ matrix\ of\ normalized\ data$
$\Rightarrow use\ eig()\ to\ find\ eigenvectors\ of\ cov\ matrix$
$\Rightarrow for\ num\ eigenvalues:$
```

```
    $find\, max\, of eigenvalue{-}storing\ vector,\ store\ associated\ eigenvector\ in\ new\ matrix,\ make\ this\ value\ zero$
$\Rightarrow Take\ first\ and\ second\ columns\ as\ principal\ components$
$\Rightarrow Project\ normalized\ data\ onto\ these\ components$
$\Rightarrow Return\ both$
```

Most of this is fairly straightforward, but I find it important to actually show code snippets of both of the *for* loops so that their exact methods for returning results is realized. Below is the first *for* loop, which populates the normalized data matrix:

```
for n = 1:ncol
   avgcol = mean(data(:,n));
   normalizedData(:,n) = (data(:,n) - avgcol) ./ std(data(:,n));
end
```

Basically, here I perform this iteration for each column; the loop runs for the number of columns. I compute the average of a given column using the `mean()` function. Additionally, I use the colon operator : to indicate that I wish to perform this operation on the entire column. I then populate the normalized data array in the second line within the `for` statement, and simultaneously subtract the average from each element, then divide using the dot operator . to divide each element by this scalar std value. After the end of this loop, the normalized loop is populated.

Skipping the rest of the code, which is fairly self-explanatory, I explain the second `for` loop below, considering it is somewhat technical:

```
for j = 1:length(eigVals)
   [A, B] = max(eigVals);
   coeffOrth(:,j) = eigVectors(:,B);
   eigVals(B) = 0;
end
```

Here, I have already computed eigenvalues and eigenvectors using the `eig` function. I now need to order the eigenvectors such that they are sorted left-to-right by descending eigenvectors. In the first part of the loop, I find the maximum of the eigVals vector. Output `A` is the value, and `B` is the important argument that finds the index of this max. I then populate the previously-created `coeffOrth` jth column with the `B`th eigenvector. In other words this `coeffOrth` is just a matrix that holds and Then, so that the next loop produces different results, I simply set the maximum to zero in the third line, meaning that the next iteration finds the second-largest eigenvalue and so on.

After this, I take the first two columns only of the `coeffOrth` matrix to make results two-dimensional:

```
coeffOrth = coeffOrth(:, 1:2);
```

I also define a `pcaData` output, which is the projection of normalized data onto the 2D `coeffOrth` space:

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

I return `pcaData` and `coeffOrth`.

With all of this established, I run this function in the main script, then use the `biplot()` function with `coeffOrth` as a primary argument, and `pcaData` as a 'Scores' argument. At the end of the function arguments, I also append arguments referring to each column in the normalized data. Further elaboration is available in the next section.

Results

The `biplot` function does its job, and this graph is created. To interpret this data, consider what this PCA process actually does. Through computing eigenvalues and eigenvectors of the `cov` matrix, ordering, and selecting the two most significant vectors, it generates a 2D basis of two of the most important *directions* of data. Along component 1, we see that cure rate, cures, mortality rate, infections, and deaths are all positive, and infection rate is negative. This implies that along component 1, all factors except infection rate are positively correlated with one another, and that they are all negatively correlated with infection rate. Component 2 qualifies this data further, and along this axis, we see a different ”direction” of the data, in which cure rate and cures are positive, and all others negative. Likewise, Cure Rate and Cures correlate positively together here, as do all other vectors negative along component 2. Additionally, considering we're in 2D space and we've established that ”closeness” between vectors along both axes is a measure of their relation, it's also fair to interpret this data as saying that the smaller the angle is between two of the six vectors, the more closely they're related. For thoroughness, the red dots are 27 in number, one for each country measured. They represent the data of each country projected onto these components' space, where they're plotted.

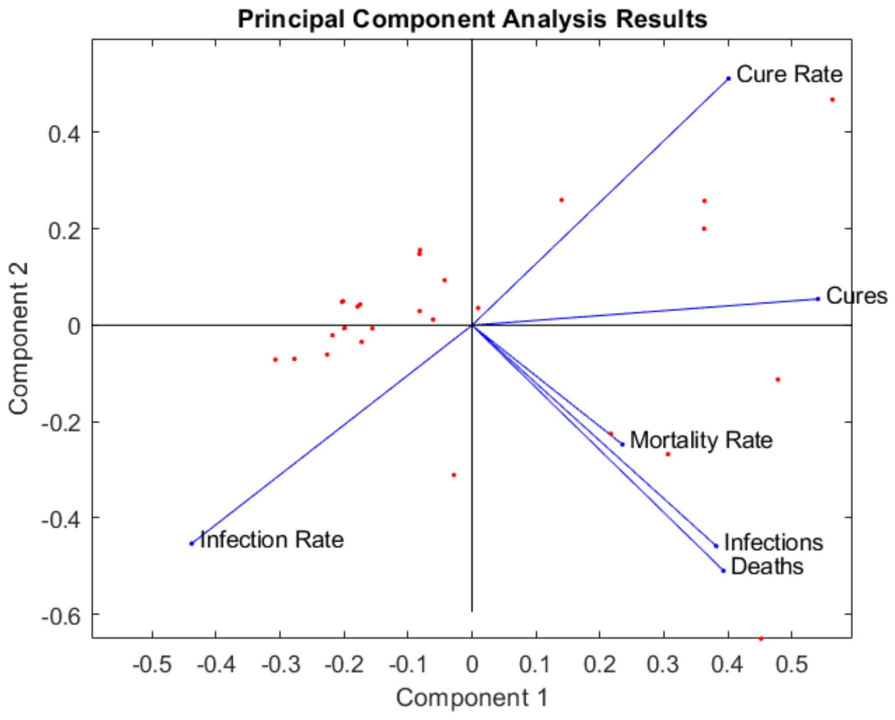

Figure 1: Output of `project_205288439_p1.m`

3 Spatial SIR Model Implementation

Introduction Second in this report is the SIR (sick, infected, recovered) model of disease used to model disease spread based on initial conditions in the provided `initialValues.mat` file. I construct my own RK solver to solve this problem, and make extensive use of MATLAB's `squeeze` and `reshape` matrix manipulation techniques to extend my analysis to an idealized grid of points with varying susceptibility. I make extensive use of abstraction throughout.

Walkthrough

The full implementation of this process involves several files: `project_205288439_p2.m`, `RK4.m`, `solveSpatialSIR.m`, `animate.m`, `dynamicsSIR.m`, and `plotTimeSeries.m`. For ease of understanding, I will walk through all of the files in a procedural order, explaining the `plotTimeSeries.m` and `animate.m` files later; they're simply files tasked with visualizing data and aren't involved in any calculation of result matrices.

The objective of this project begins with a physical understanding of what's going on. Per the problem description, I'm to imagine a 2D grid M*N, with M as 50, and N as 70. More explicitly, these correspond to a grid in (x,y) coordinates of form (M,N). For each point on the graph at a given time step, I'm to compute what percentage (amount) of each cell is susceptible, infected, or recovered. For each cell, $S + I + R = 1$ holds due to the initial conditions provided in `initialValues.mat`.

`InitialValues.mat` is a 3D matrix of size $M * N * 3$. Essentially, we can imagine this as a 2D grid of $M * N$ with "thickness" of three. These three values ascribed to each cell are initial values corresponding to sick, infected, and recovered values, respectively. Thus, for example, taking (1,1,1) will find the proportion of susceptible individuals in the coordinate (1,1). Similarly, taking (45,19,3) will find the proportion of recovered individuals in cell (45,19).

With this understood, I'll move on to the main function that models this and forms the basis of this entire project: `dynamicsSIR.m`. This function has interface `function dxdt = dynamicsSIR(x, M, N, alpha, beta, gamma)`. x is simply an array that is passed in in $M*N*3$ format (this will be explained later- in `solveSpatialSIR.m`, I make the previously 3D matrix a column vector; I 'linearize' it), M and N are the length of the x and y axis (corresponding to initial conditions data) and alpha, beta, gamma are all constants that govern various factors about disease spread. This pseudocode basically explains the code:

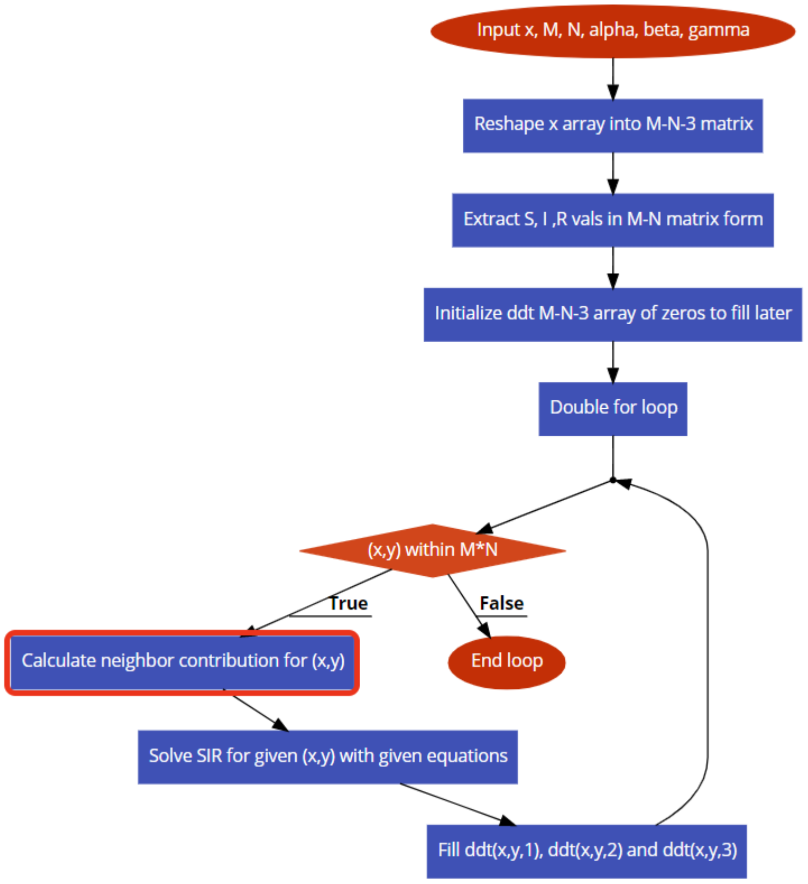
Input x, M, N, alpha, beta, gamma
Reshape x array into M-N-3 matrix
Extract S, I ,R vals in M-N matrix form
Initialize ddt M-N-3 array of zeros to fill later
Double for loop
(x,y) within M*N
True
False
Calculate neighbor contribution for (x,y)
End loop
Solve SIR for given (x,y) with given equations
Fill ddt(x,y,1), ddt(x,y,2) and ddt(x,y,3)

Figure 2: Flowchart describing `dynamicsSIR.m`

Additionally, the code linearizes results to return a $M * N * 3$ matrix in the final line, returning linearized dxdt: `dxdt = ddt(:);`

To elaborate on the flowchart a bit, I need to explain the `for` loop and how it works. This is the bulk of the computation involved in this entire problem, and solving this with ODE solvers is what causes the code to take several seconds to run.

In the double `for` loop, I calculate the weight of neighbors according to distance. Neighbors immediately above, below, left, or right of the cell have weight 1, and those at the nearest diagonal (four total) have weight $\frac{1}{\sqrt{2}}$. All other x,y points are assumed to have no relation. Thus, we have a matrix of nine elements that can contribute to the results at a given cell (x,y). Thus, before the `for` loop, I instantiate a 3*3 W matrix, which I fill with contributions in the for loop. I assume W(2,2) to be the position of the (x,y) graph being calculated. *What is very important* is that the grid does not "wrap around"- thus, there are coordinates along the edges, in the central region, and in corners. For each of these cases, the values of the W matrix look a lot different. For example, corner pieces can only be affected by three neighbors. Per the equations that govern SIR modeling, which are:

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

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

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

I get each W(x,y) value and multiply it by the I(x,y) value at that point. Thus, for example, the nonzero W coordinates of a bottom left corner are:

```
W(3,2) = 1*Ivals(i+1, j);
W(2,3) = 1*Ivals(i, j+1);
W(3,3) = (1/sqrt(2))*Ivals(i+1, j+1);
```

With (3,2) being to the left of the coordinate (x,y), (2,3) being above, and (3,3) being to the right and up. After this process, I necessarily sum the W matrix per the equations listed above. Then, I input everything into these three differential equations, and get data stored in `ddt`. This is illustrated below; equations are put into MATLAB format:

```
ddt(i,j,1) = -(beta * Ivals(i,j) + alpha*Wsum)*Svals(i,j);
ddt(i,j,2) = (beta*Ivals(i,j) + alpha*Wsum)*Svals(i,j) - gamma*Ivals(i,j);
ddt(i,j,3) = gamma * Ivals(i,j);
```

This happens for each coordinate, or M*N times. As mentioned before, the function ends with results being linearized such that output is a M*N*3 column vector.

With this established, the main machinery of the program is implemented. The rest is building a solver that can solve this. Keep in mind that the function just mentioned takes in a given set of initial conditions and returns a *derivative*. Basically, what we did was just model a sophisticated differential equation. Now, we need to solve it for explicit S, I, and R. This is done in `RK4.m`.

The RK4 function does what it promises from its title- it implements a fourth-order Runge-Kutta numerical ODE solver. The exact derivation of this process can be found in *Final Project*. Essentially, by making several estimates about the value of the next step, we can weight estimates in a given way and find a close estimate for the differential equation solution at a next step *t*. This is similar to the Euler method, but far more accurate.

In my RK4 implementation, I have three inputs: function handle, timespan (labeled as 'tspan', and y0 (initial conditions). The RK solver has a fixed step size 0.1s. The RK solver is also generalized to take in columns of initial conditions. Each time step, it creates a *column* of new values. To actually compute: I do the following, as written in pseudocode:

```
$\Rightarrow Take\ in\ f,\ tspan,\ y0$
$\Rightarrow Calculate\ number\ of\ steps\ and\ create\ array\ of\ timestep\ values$
$\Rightarrow Initialize\ 2D\ zeros\ array\ of\ length(y0) * length(tarray)$
$\Rightarrow Set\ first\ column\ equal\ to\ ICs$
$\Rightarrow for\ number\ of\ steps$
```

Calculate next column using RK method

Below is also an incomplete snippet of the RK "machinery":

```
k1 = h*f(t_n,y_n);
k2 = h*f(t_n + 0.5*h, y_n + 0.5*h*k1);
k3 = h*f(t_n + 0.5*h, y_n + 0.5*h*k2);
k4 = h*f(t_n+h,y_n+h*k3);

   y_nplus1 = y_n + ((k1 + 2*k2 + 2*k3 + k4)/6);
y(:,k+1) = y_nplus1;
```

To clean up after myself, I also `permute` the returned t array such that it is in the `nsteps*1` format. This mimics the output of ODE45 and it's just for consistency. To conclude, this t array is returned, and so is the y matrix. With all of this established, the rest is mainly just increased abstraction, visualization, and providing a script for the functions to actually run from.

In `solveSpatialSIR`, a fairly simply, 26-line script, I take in an odeSolver handle, conditions alpha beta and gamma; final t, and the initial condition matrix. I linearize the initial conditions by using the colon operator:

```
   linearized = initialCondition(:);
```

Also, I implicitly take t(0) to be 0 in the following line: `times = [0 tFinal];`

I also infer the size of the grid (M and N values) by using the `size` command in MATLAB for the first and second dimensions of input, considering it's in M*N*3 format. I then establish the following anonymous function, which is the one that is actually evaluted. It calls to `dynamicsSIR.m`:

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

This is then passed into `odeSolver`, which actually was an input in the line:

```
   [t, x] = odeSolver(dSIRdt, times, linearized);
```

This line alone is what allows the SIR model to be solved. By calling to the machinery in `dynamicsSIR` and using a differential equation approximation method, we actually yield S(t), I(t), and R(t) values. However, they're still linearized. We need to shape them into a M*N*3*T matrix. I do that here:

```
   x = reshape(x, [M N 3 length(t)]);
```

To summarize the role of this function, I'd describe it as unnecessary but useful. We could have simply called an ODE solver in the main script, but doing so for multiple solving (which I do with RK and ODE45) would require repetition. By creating a function that takes in a function handle and does the formatting on the output itself, we save ourselves from redundant code. Column array t of times from the solver and the M*N*3*T 4-D array are returned from this function.

In `plotTimeSeries.m`, I create subplots displaying S, I, and R values for a given cell across all times *t*, respectively. The function takes in 4-D results array in M*N*3*T format, x and y coordinates, and a time array (column time array output from ODE solvers). To get the S, I, R values for one cell for all of time t, I do the following:

```
Svals = squeeze(X(x,y,1,:));
Ivals = squeeze(X(x,y,2,:));
Rvals = squeeze(X(x,y,3,:));
```

In other words, I take the x,y components of the first two dimensions, get the S, I, or R value (1,2,3 in 3rd dimension), and get all times in the fourth with column operator :. Notice that I used the `squeeze` function, which deletes the singleton dimensions, making the results a graphable 1D matrix. Using `subplot()`, I plot the S, I, and R over time for given cells. I make the titles dynamic by using the `sprintf` command as follows:

```
formatSpec1 = 'Susceptibility in (%.0f,%.0f)';
S_str = sprintf(formatSpec1,x,y);
title(S_str)
```

I save files of graphs each time this function is called with the following two lines as well:

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

The 'gcf' is 'get current figure', which the program interprets as the current image displayed.

In the final script, `animate.m`, I actually animate the results, and show disease spreading. The animation function takes in 4-D solution vector and uses the `image` function to display a figure. Because the function isn't very long, I elect to not use pseudocode and to rather explain with snippets.

First, as required in the project description, I want infected to be displayed in red, susceptible in blue, and recovered in green. However, when the image function takes in a 4D array, the 1st dimension index is in red, the second in green, and the third in blue (RGB). I can perform the following trick to swap the colors:

```
Z = X(:,:,1,:);
X(:,:,1,:)  = X(:,:,2,:);
X(:,:,2,:)  = X(:,:,3,:);
X(:,:,3,:)  = Z;
```

Here, I create the intermediary array Z. Then, I make the new first index infected values, the second value recovered values, and the third the Z values (susceptible). Doing so, I can effectively display results in BRG for SIR instead of RGB, respectively.

Next is a `for` loop where I make an iteration approximately every tenth step, using the rule:

```
k = 1:floor(size(X,4) / 10
```

For 601 time points, this makes exactly 60 iterations. In the loop, I label the graph and actually display results at each tenth t value:

```
z = X(:,:,:,10*k);
image(z)
```

This is essentially all that's behind this function. I also add a 0.1s pause with `pause()`

In `project_205288439_p2.m`, I consolidate everything, and make calls that allow everything to work. Here, I load in initial values and constants, and make calls to `solveSpatialSIR` for ODE45 and for my own RK solver. I also make calls to `animate` and `plotTimeSeries` to visualize results. Images of results and analysis are available in the next section.

Results

As required in the project description, the following are specific deliverables:

Tic-Toc Analysis: In the main project file of part II, `project_205288439_p2.m`, I use the commands `tic` and `toc` to measure the runtime of ODE45 and the RK solver. I display these results, which looks like:

```
RK runtime is 0.7413 seconds
ODE45 runtime is 0.1152 seconds
```

As expected, the RK runtime is a lot longer. This is mainly because it has to do many more steps than the ODE45;

for tfinal = 60, the RK solver will always have 601 time points (600 times it needs to solve), while ODE45 will try to "economize" and take larger step sizes when accuracy losses are minimal. This explains why ODE45 is faster- it's simply got less to calculate. From the variables returned, I can see that ODE45 takes only 161 steps to get a solution.

Required `plotTimeSeries` graphs can be seen in the appendix section.

4 Conclusion

What follows prior is, to the best of my knowledge, a correct implementation of PCA analysis and SIR implementation in a simplified model. This report and the code it is based on represents a culmination of what I've learned in M20, and I extend gratitude to both my professor Khalid Jawed and TAs for their assistance in helping me learn this software this quarter.

References

K. Jawed. *Final Project.* Fall 2020
n.a. *covid_countries.csv* n.d.
n.a. *initialValues.mat* n.d.

5 Appendix

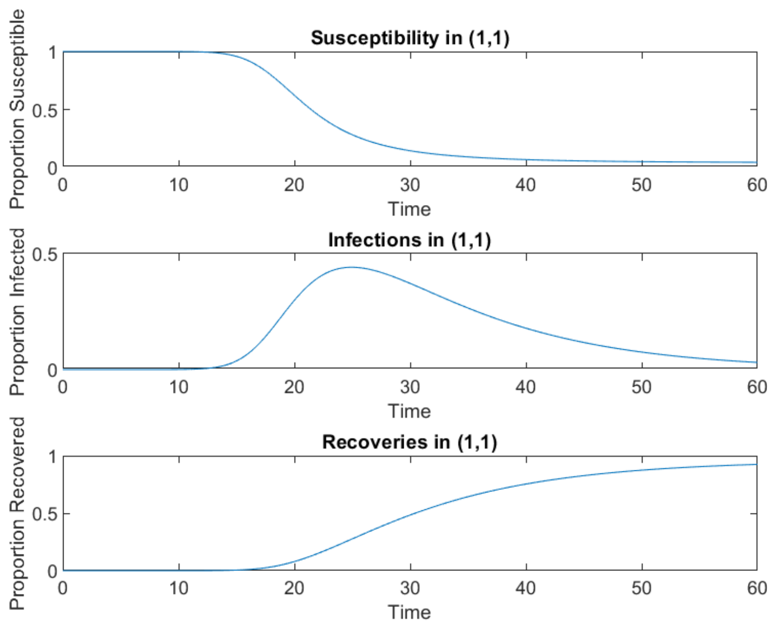

Figure 3: Output for (1,1)

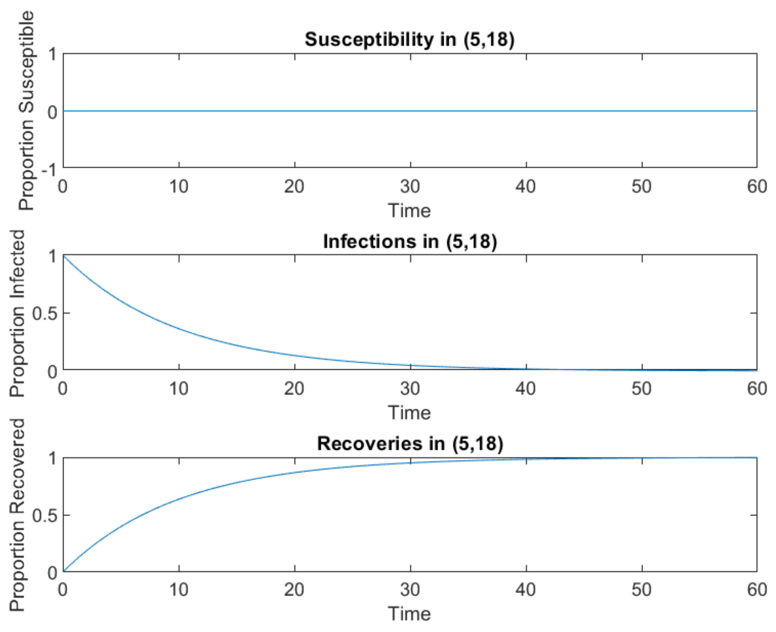

Figure 4: Output for (5,18)

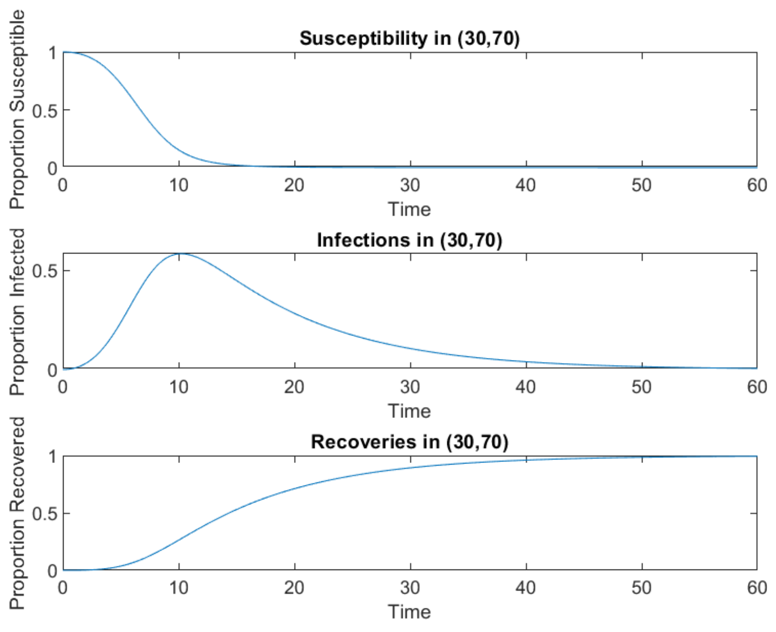

Figure 5: Output for (30,70)