Mauricio Deguchi
UID: 405582562
CEE/MAE 20
December 3, 2021

Final Project

1 Problem 1

1.1 Introduction

Here, we are introduced with the issue of dimensionality. We have been provided a surplus amount of variables which would inevitably lead to a more complex system. This problem seeks to reduce the dimensionality of the data by identifying the leading contributors from the set of variables. To achieve this, we will use Principal Component Analysis (PCA) as our method of reduction. We will create our own PCA function to break down each component of the method.

1.2 Models and Methods

For starters, we will need to import our data from the provided csv file. This can be done through use of the `readtable` function; once we have the table in Matlab, we must store only the numerical data portion of the table. This is done by eliminating the first two columns and first row which includes dates, titles, and names. From there, we can transform our table to an array which can be translated into our myPCA function. More details of the myPCA function will be included under calculations and results (section 1.3).

Once the myPCA function has processed the data, we can begin to create our figure. We begin by recalling the original data file and storing the variable names in a cell array. Again, we must truncate the variable names to eliminate the country names and dates. Next, we must truncate the two arrays returned by the myPCA function to solely the first 2 columns, which contain the information from the largest contribution variables. Once we have that data, we are ready to plot; using the `biplot` function and the extracted data, we can create Figure 1.1 shown below.

1.3 Calculations and Results

myPCA

Let's take a closer look at the specifics of the PCA analysis. For starters, we must take in the numerical input data. Once that is stored, we can begin standardizing the data. To standardize the data, we will need to calculate the mean and standard deviation of each column. Equation 1.1 and 1.2 outline these calculations.

$$\bar{x} = \frac{\sum data(:,j)}{\#\, of\ rows}$$

Equation 1.1 Calculates the mean of each row 'j'

$$\sigma = \frac{\sum(\bar{x} - data(:,j))}{\#\, rows\ -\ 1}$$

Equation 1.2 Calculates the standard deviation for each column 'j'

From these equations, we can standardize each column of data by subtracting each row entry in a column by the mean of that column and dividing it by the standard deviation of that column.

Now that our data has been standardized, we must compute the covariance matrix. The equation from the covariance matrix has been provided in the problem statement on page 3. To use this equation, we must take the transpose of that standardized data matrix we have just computed, which can be done by using the `matrix.'` operator to any matrix. From there, the calculation follows the equation provided. Then, we must compute our eigen values and vectors by applying the `eig` function to the covariance matrix. From there, we must sort our eigen values to determine the variable with the largest contribution; likewise, we will also need to sort our eigen vectors in their corresponding order. From there, we can output the array of sorted eigen vectors and lastly, we must multiply our standardized data by the eigen vectors to obtain the projected data onto the eigen space.

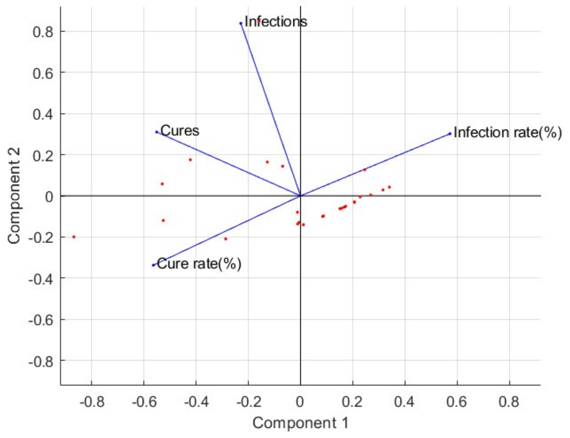

Figure 1.1 Biplot of the 4 variable data projected onto the principal eigen space

1.4 Discussion

As discussed previously, once the myPCA function returns the transformed data, we can create the plot shown in Figure 1.1. The axes of the biplot represent the leading eigen vectors of our data set. Each of the 4 variables we were initially provided are represented as vectors in the figure and the individual red points represent the countries we inspected. From the biplot, we can determine that ‘infection rate’ and ‘number of infections’ provide the largest contribution out of the variables provided. This can be determined by inspecting the length and direction of each vector; longer vectors close to a positive axis show that those variables have positive coefficients. Overall, this problem provided valuable insight into how to deconstruct a multivariable system into its most significant contributions. I can understand how overfitting a system would not only be computationally expensive but also highly complex to understand and it may even lose the generality of a system leading to minimized use cases.

2 Problem 2

2.1 Introduction

This script will incorporate all of the remaining functions to generate various plots, graphics, and outputs. This script will call each of the functions, completing a different component of problem 2. Each of the components have been separated into sections with a small amount of commands to call the functions and generate any necessary results. The goal is to create a spherical mesh network where each node represents an SIR population. Each node's SIR population will influence the SIR status of the neighboring nodes; thus, when we place initial infections in our mesh, we hope to see the other coordinates gradually become infected and recovered.

2.2 Models and Methods

2.2.1 Loading Mesh

This section will load in the mesh structure which will be needed throughout the rest of the problem. We have been provided a modifiedSphereSTL.txt file which follows the format displayed in the problem statement, so we will use this pattern to our advantage later. Prior to calling the function, we begin a timer to clock the speed of the stlRead function we created. The mesh structure will consist of two types of data: (1) the coordinates of all the unique points on our grid and (2) the location of neighbors for each point. This mesh structure is created through the stlRead function which imports and interprets the given .txt file.

The stlRead function first imports the modifiedSphereSTL.txt file into Matlab through the `readcell` command. Once we have imported the file, the function then initializes the mesh structure and a separate array which will be used to aid in searching and indexing unique points.

We then move into the for loop where we search through the file for two key characteristics: 'outerLoop' and 'Corner'. The outerLoop search is used to find each set of 3 points. These three points can then fall into 8 different possible combinations of unique and previously seen points outlined in the following table:

Point 1	Point 2	Point 3
Unique	Unique	Unique
Seen	Seen	Seen
Unique	Seen	Seen
Seen	Unique	Seen
Seen	Seen	Unique
Seen	Unique	Unique
Unique	Seen	Unique
Unique	Unique	Seen

Table 2.1 Outlines all of the possible combinations of categories of points at each outerLoop.

Through the following combinations we write each case as if statements to add our unique points and various neighbors to the mesh structure. Next, we search for 'Corner' tiles which we then store in the points array if the point is unique. This array allows us to easily search for unique points and their index to build our mesh structure. The for loop goes on through the entire length of the txt file at which point, our mesh structure will contain all the necessary information. This mesh is then returned to the main script as the output of our function. We then end the timer, return the clock-speed and move on to the next section.

2.2.2 Loading SIR Parameters

We then move into loading in the SIR parameters. This is done mainly using the `load` function where we load in the SIRparameters.mat file. These parameters will mostly be used later in the code. One piece which is currently used however, is the initial infections cell. This provides the points where the infections first begin. We must then initialize the conditions for each point, so we begin by creating an array with size Nx3 where N is the number of unique points. This array contains value of 1 for each first column and zeros everywhere else. Each column represents one of the SIR letters in that order, so to begin, each point has all of their population as susceptible. We can then use our initial infections to reassign the corresponding points their [0 1 0] starting SIR condition. This is done by searching through the mesh structure for matching points; once a matching point is found, we reassign its value. In order to find whether or not a point matches, we find the distance between the point we are looking, and a given point and set a low tolerance of error to find whether or not the points are sufficiently close to be defined equal. Once we have completed this, we are ready to move on to the next section.

2.2.3 Solving the Spatial SIR

We then move into solving the spatial SIR. Here, we solve the SIR to output each point's change in SIR condition over time. So, for each point, we will need the 3 SIR states over however many points of time the ODE solver requires. This will create a Nx3xTime array. We will also need our corresponding time array which will be Time x 1 array containing all of time steps. We will conduct this using two different ODE solvers and compare the efficiency of our custom ODE Solver and Matlab's ode45.

Now that we understand what we need to produce, let's jump into piecing down the code. We will use two identical blocks to run our functions using ode45 and RK4. These blocks will start a timer, call the solveSpatialSIR function; then, output our 3-dimmensional array, the time array, end the timer, and return the clock-speed to the user.

For starters, the solveSpatialSIR function itself includes the dynamicsSIR function and the odeSolver of our choice. For this problem, we coded our own RK4 ODE solver and compared it to the performance of Matlab's built in ode45 solver. Within our solveSpatialSIR function, we first declare the dynamicsSIR as a function of x and t and it is used within our ode solver.

Dynamics SIR
The dynamicsSIR function takes in an array of the current state of the SIR populations at each point and returns the SIR rate of change at each coordinate. Using equations 5, 6 and 7 in the

problem statement, we can determine the rate of change of the S, I, and R populations at each point, given the current status. The equations rely on the current infected and susceptible populations at each given point and a neighbor contribution.

To begin working with the input data, its easiest to first reshape our data into a Nx3 array where N is the number of points and each column contains the respective S, I and R data for all given indices. This means that each row in our array contains all SIR data for a single coordinate. Now that we have reshaped our data, we will also need to preallocate our memory for the derivatives. This is done by establishing an equally sized Nx3 array of zeros. Now, we can begin calculating our derivatives.

First, let's look at the population's own effect on itself. The current state is linearly dependent on itself and parameters alpha, beta, and gamma which stand in for variables like the infection rate of the disease. This is implemented into the code through a for loop, looping through all of the points. Next, we must incorporate the neighbor contribution for each point and its effect on the derivatives. We first establish our neighbor contribution as 0 and then modify its value. In order to calculate the neighbor contribution, we must implement the following summation:

$$\sum_{j \in N(i)} \frac{I_j(t)}{d(i,j)}$$

Equation 2.1 The neighbor contribution in the dynamic SIR calculations

To do so, let's implement a for loop. The for loop will loop through the neighbors of a given coordinate, take the infected population at the neighbor, divide it by the distance from the coordinate we are inspecting, and add it to the total neighbor contribution. We reset our neighbor contribution for each coordinate and run the loop again. This is then implemented into the previous calculations in the outermost for loop. Once the loop is complete over all points, we vectorize our derivatives into a column array and return it to the function call.

RK4

The dynamicsSIR is used as a function input within our ODE solver. Our RK4 ODE solver takes in our dynamicsSIR function, an initial and final time, and the initial conditions. The pseudocode is provided in page 6 of the problem statement. The function f in the pseudocode is our dynamicsSIR function, the initial conditions are our start and end time, and an initial array of our SIR conditions. The RK4 function, runs a while loop from the provided start time to the given end time. The while loop is dynamic, meaning that the time steps vary at each point. The time step is determined by the h value in our code, which itself is determined by comparing the next order approximation to our current approximation; finding the distance between them and multiplying by a constant completes our calculation for our time step. The RK4 function solves the derivatives provided from the dynamicsSIR function and returns a finalized array of time values and y array which contains the SIR data over all the time points for each coordinate.

Spatial SIR

Once our ODE solver returns the array of solved SIR conditions and time array to the solveSpatialSIR function, we must reform it prior to returning it to the main script. The ode solver returns a time x (3*N) array and a time array to the solveSpatialSIR function. This data can be interpreted as the following: each row represents a single time step with SIR data for each coordinate in the format of [N Susceptible information, N Infected information, N Recovered information].

Knowing this, we can begin reformatting our data. We start by finding the length of our time array and preallocating memory for our final array as size Nx3xTime. So, our final array will have three columns of SIR data for each point 1-N over each time step. We can reassign our values using a for loop, which will loop through time. We can extract one row at a time form our y-array (output by the RK4 function) which will contain our SIR data. We can then reformat the data into an Nx3 array to match our final array using the `reshape` function. Lastly, we replace this array with the given time step in our final array; this is repeated for each time step until we have finalized our array which can be output to the main script.

RK4	ode45
10.052	3.804
10.159	3.558
10.096	3.507
9.947	3.460
10.116	3.512

Table 2.2 Time Required to run solveSpatialSIR using different ODE solvers (seconds)

Our various trials of the solveSpatialSIR function using ode45 and RK4 resulted in the data provided in table 2.2. As we can see, RK4 is substantially slower than ode45; this could be due to a number of factors, some of which could include more efficient coding of the solving method, ode45 being a more efficient solver than the Bogacki-Shampine 3rd order Runge-Kutta method, higher efficiency uses since it is built into Matlab, etc. One other reason could be that, on average, ode45 used less time steps than RK4, meaning that it had to run less calculations. While, the two methods differ greatly in terms of run time, their numerical outputs are similar when used in the subsequent functions.

2.2.4 Calling Plot Time Series

The plotTimeSeries function creates a figure including a subplot for the SIR values at a single point over the time array. We provide the function with the mesh structure, time array, y array output from the Spatial SIR, and the coordinates of the point we want to inspect. We begin by converting the mesh structure into a coordinates array of size Nx3, where N is the number of unique points in our mesh structure, using a for loop which saves each location's x, y, and z coordinate. This enables us to easily search through the array to find the data of the point we want to plot using the `find` function. Once we have the index of our point, we can extract the S, I and R data form the y array for that given coordinate and store them as three separate arrays. We can then plot these S, I and R arrays versus the time array in three subplots. We were given 3 points to analyze, and the results are displayed in figures 2.1-2.3 below.

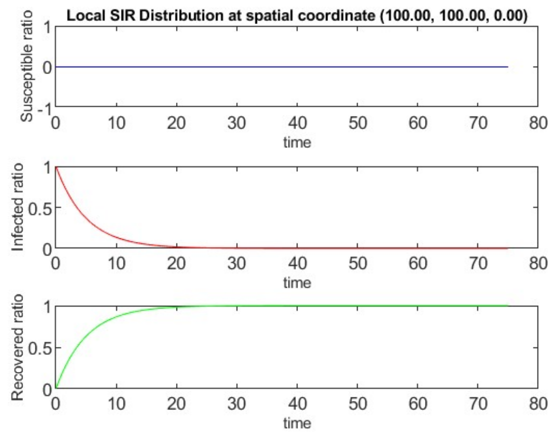

Figure 2.1 SIR Distribution with initial infection

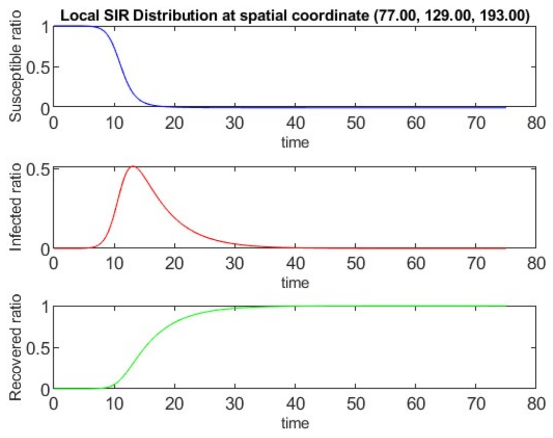

Figure 2.2 SIR Distribution near initial infection

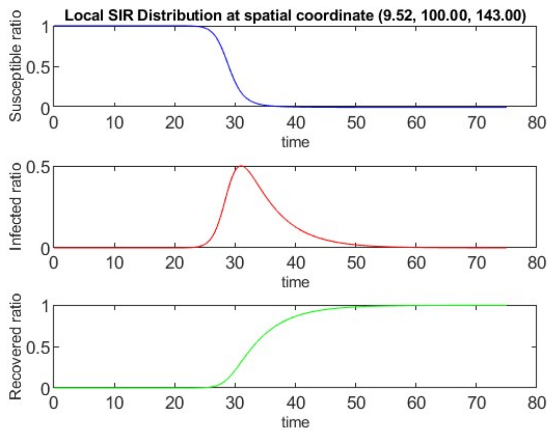

Figure 2.3 SIR Distribution far from initial infection

In Figures 2.1-2.3 we see points with varying distances from an initial infection. In Figure 2.1 we have the plot of a point where we begin with an infection; thus, or susceptibles remain constant at 0 and the recovered ratio gradually increases just as the infected ratio decreases. Next, in Figure 2.2, the point is close to an initial infection, leading to a spike in the infected ratio shortly after the time begins. Lastly, Figure 2.3 shows a population far from the initial infection where it takes some time for the infections to begin. One key feature across all of these figures is that the ratios at any point in time always sum to 1. This is because the ratios are fractions of the whole population, thus the total population must be 1, assuming everyone recovered and there were no deaths.

2.2.5 Calling Animation

The animate function takes in the full array of SIR conditions over time, the time array, and the mesh structure. Once it has obtained the inputs, the animate function establishes a color array where it contains all the SIR data for each point over time as a Nx3xTime array. The ‘x3’ allows us to use the standard color notation in Matlab where a 1x3 array can indicate a specific color. We can then use this and the pcshow command in a for loop, looping through time. However, as outlined in the problem statement, we only want to show some frames with the condition that the next frame’s time must be greater than 2 seconds past the previous frame. We can incorporate that as an if statement in the for loop and nest our `pcshow` and `drawnow` commands in the if statement. We will also want to hold the frame on the figure window, and this can be achieved through the `pause` command. The culmination of the for loop generates the animation and select frames have been extracted in figures 2.4-2.9 below to portray the overarching idea of what the animation looks like.

The animation figures display various time steps of the animation; what cannot be captured in these time steps is the speed at which the spread increases. While at the beginning there are only a few infected populations, the more time that passes, the more neighbors become infected, leading to a larger neighbor contribution from our Spatial SIR calculations. This larger contribution leads to faster spread as more nodes get infected.

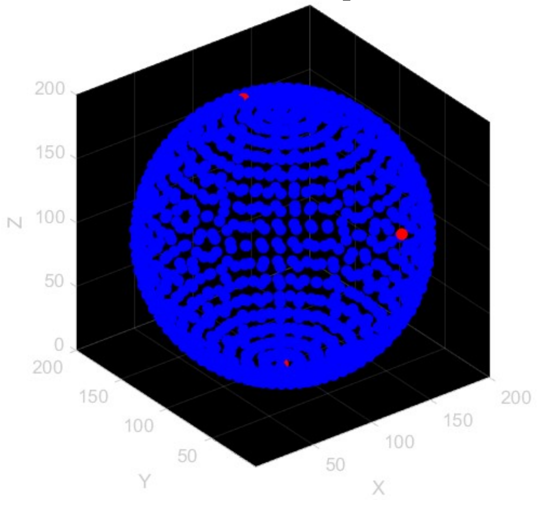

Figure 2.4 Animation at initial conditions

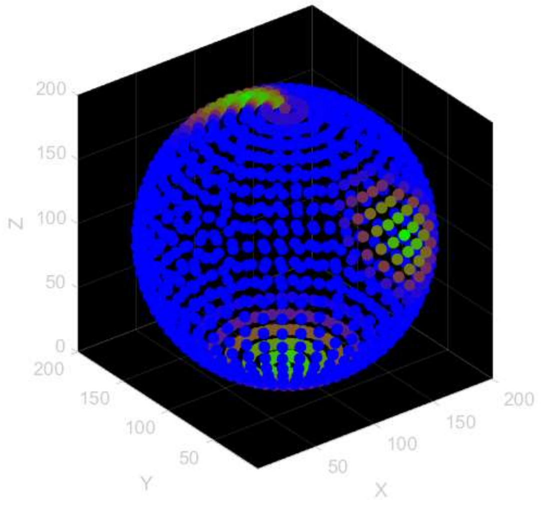

Figure 2.5 Animation at slow initial spread

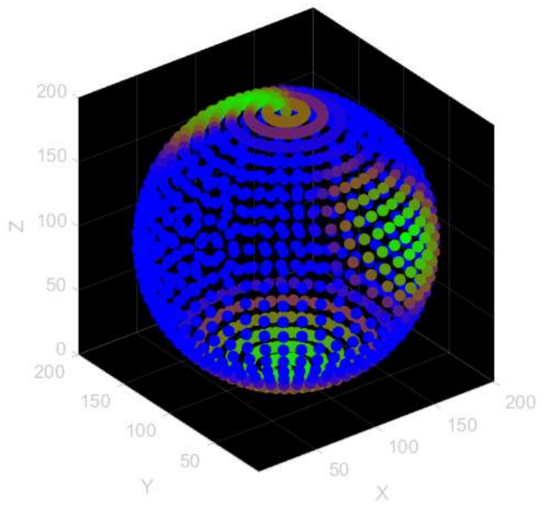

Figure 2.6 Animation as disease spread faster, top node has been infected

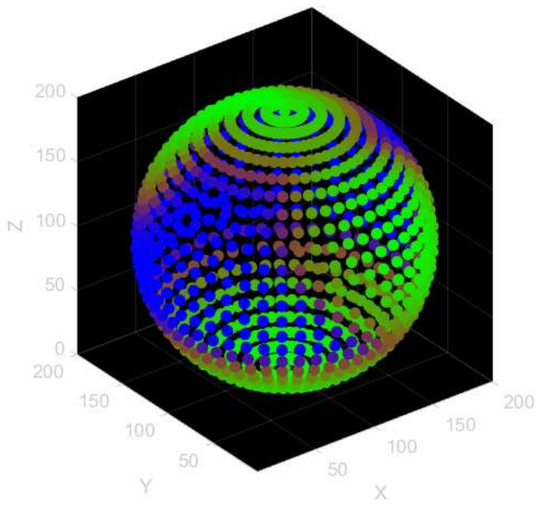

Figure 2.7 Animation as most populations encounter the disease

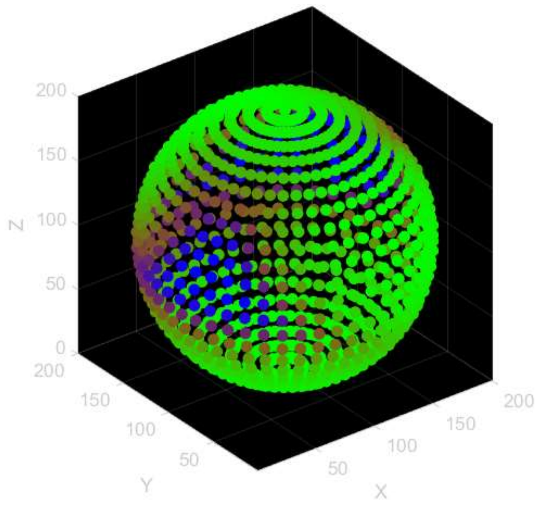

Figure 2.8 Animation as we approach the climax

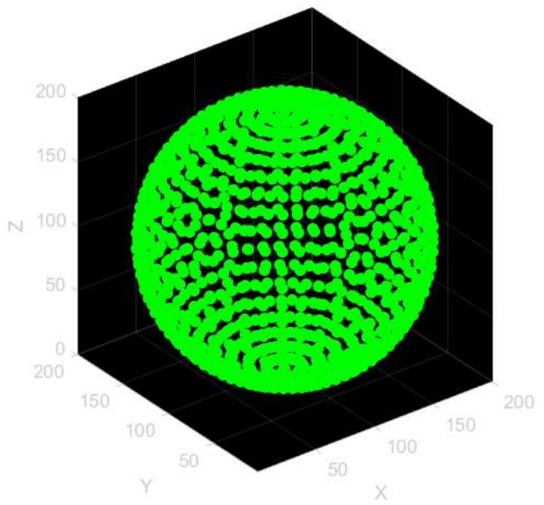

Figure 2.9 Animation at the final state

2.2.6 Calling write to Excel

Our last function is the write2Excel function. Here, we want to take our SIR data and transfer it to an excel file with the corresponding coordinates of each point. To do so, we begin by taking our mesh structure and converting it into a table. We then get rid of the ‘neighbors’ section of the table by setting it equal to empty grid spaces, leaving only the coordinates. Then, we can turn our table into an array which can be easily indexed. From this array, we can extract the 3 columns as the x, y, and z coordinates of each node. Before going into our loop, we must also preallocate memory for our SIR data; this memory is saved by creating an S, I, and R, array of zeros with size Nx1. Lastly, we must declare the names of the variable we will be storing in our sheet.

We can then move into the for loop where we want to create sheets of data from points in time that are at least 15 seconds apart. To achieve this, we have an outer for loop and a nested if statement where we create our sheet. The for-loop loops through all the time points; inside, the if statement checks if our current time is at least 15 seconds since the last sheet was created or if we are at the end of our indices. If so, we extract our S, I and R data from the input array at that given time step. We can then create a table from our x, y, z coordinates and S, I, and R data. Before writing the table to Excel, we must also name the sheet we will be writing on; then, we can use the `writetable` command to write our data and labels to the Excel sheet. Lastly, we must update our next time sheet by 15 seconds to continue our search in the next iteration. The result is an Excel file with various sheets including the points and SIR data at those points for time steps 15 seconds apart.

2.4 Discussion

The culmination of this problem was extraordinarily satisfying. Seeing all the functions come together and work in harmony at the end was fascinating. While the process was burdensome, when broken down into smaller components, the problem became a lot more bearable. Having had no prior experience with structures made the initial part of this problem very difficult to understand conceptually. However, as I became more comfortable with the concept, completing that section became quite intuitive. The other major struggle came when solving the spatial SIR. That function itself was quite simple but the components from which it was composed of were quite challenging. Similar to structures, where the conceptual understanding was more difficult than the actual code, the dynamicSIR function being redeclared as a function and then using it in the RK4 function which itself was in the solveSpatialSIR function seemed overwhelming to understand. After moving through the code line by line, I was able to grasp a fundamental understanding of how the code transitioned from one piece to another. Aside from those two conceptual boundaries, the rest of the problem progressed smoothly. The animate function was a major milestone and relief as I saw the combination of each function in a visual representation working to its fullest extent. The write2excel and plotTimeSeries functions were relatively simple and provided useful outputs but didn't quite have them visual appeal of the animate function. One final challenge I encountered was the preallocation of memory; while some functions clearly needed preallocated memory and it could be implemented straight forward, others, namely the stlRead function, were very difficult to preallocate memory in a structure format. I preallocated where I could and saw necessary but in that instance, I wasn't able to come up with a more efficient method and was supported by the TAs in my decision.