Adrian Loekman
UID: 105785820
CEE/MAE M20
November 27, 2021

FINAL PROJECT

1. Main Component Analysis with PCA

Principal Component Analysis, more commonly known with the abbreviation PCA, is used widely in machine learning. In this emerging era of technology, many data have been known to have high dimensionality as more and more factors came to affect the data. Consequently, analyzing data with such dimensionality creates a difficulty since in exponentially increases time to solve them. One method of decreasing dimensionality is through computing the PCA.

1.1. Introduction

The goal of this problem is to minimize a 4-dimensional data (cure rate (%), cures, infection rate (%), and infections) into a 2-main-component and 2-dimensional data that could be presented in a biplot. The end product of this problem is a 2-dimensional biplot that visualizes the first two principal component eigenvectors projected onto the 2-D vector. In order to achieve this, there are a few steps to follow.

1. Data Standardization
2. Covariance Matrix Computation
3. Eigenvalues/Eigenvectors Calculation
4. Principal Components Extraction
5. Normalized Data Projection

These 5 methods are going to be implemented using a function and visualized in a biplot presented from the main script.

1.2. Model and Methods

A. Function `myPCA.m`

These steps that will be thoroughly describe below will be implemented in the function `myPCA.m`. There will also be 2 additional functions `myMean` and `myStd` inside `myPCA.m` to help organize a better implementation of the function.

1.2.1. Data Standardization

In this step, we will be computing and creating a normalized data. This step could be achieved using the data average and standard deviation. Since the data given is divided into 4 categories, we will need to compute the average and standard deviation for each category. Next, we can normalize each element of the data using the average and the standard. The formulas of average and standard deviation are shown below:

$$Mean(average) = \frac{sum\ of\ data\ points}{Number\ of\ data\ points}$$

$$Standard\ Deviation = \sqrt{\frac{\sum(X - mean)^2}{N}}$$

X = The element of each category
Mean = average of each category
N = Amount of data in each category

The implementation for this method in MATLAB is very easy. Note that our data for this problem is a matrix of 27 rows and 4 columns. Each column represents a distinct category. Thus, the mean and standard deviation must be created for each column.

The first step is to calculate the size of the matrix. This can be easily done using the function `size`. Next would be calculating the mean. This is done in a separate function to promote coding organization and generalization of the main function. The average function, `myMean`, has the data as its input and output a mean. Since this problem is a 4-column matrix, the mean would consist of 1x4 array.

```
dataMean = sum(data)./length(data);
```

Using the syntax above, the average of each category is calculated. `length(data)` is used to calculate the number of data (i.e., the number of rows) inside the matrix itself. Lastly, call this function into the main function.

Computing the standard deviation is a trickier task since it requires the single element from each column. This means that we need to isolate each column by itself and subtract it with the column's (category's) mean, then sum all those up and divide it with the total of rows (elements) and take the square root. This is a long process but can be done using a for loop. This will be done in another function called `myStd` with the data as the input.

```
for i = 1:n
    avg = dataMean(i); % find mean for each column
    for j = 1:m % for each column calculate the data - average
        dataSum(j,i) = (data(j,i) - avg)^2;
    end
end
```

The for loop above shows calculating the $(X - mean)^2$ part of the standard deviation. the iteration 'i' is used for the columns while 'j' for the rows. Since the mean is a 1x4 array, it should be called onto the correct column. dataSum is initialized as zeros before the for loop with the same elements as the original data. Start calculating the data for each column and swap the mean values every time the column changes.

`dataSum` should be a m x n matrix and we need to calculate the sum of each column. Simply use the `sum` function and a 1x4 matrix containing the sum of each data

subtracted by the average squared is gotten. Lastly, divide that sum of data with the number of data in each column and take the square root. That is the standard deviation, a 1x4 array. Call this function into the main function `myPCA.m`

Now that we have the mean and standard deviation, we can normalize the data. Each data will be normalized by subtracting each data by the average of its column and dividing by the standard deviation. This can be done using a for loop similar to how the standard deviation was computed. The size of this normalized data, `dataNorm`, is identical to the original data.

1.2.2. Covariance Matrix Computation

The covariance matrix can be calculated using our standardized data, `dataNorm`, using the formula:

$$Covariance\ Matrix, C(M) = \frac{M^T * M}{m - 1}$$

M is the matrix we want to evaluate, and M^T is the transpose of that matrix (i.e., swapping the rows and columns of the matrix). m is the number of rows, so in this problem m is equal to 27. In MATLAB, the syntax would look like this:

```
covMat = (dataNorm' * dataNorm)/(m-1);
```

The notation `dataNorm'` is used to transpose the matrix.

1.2.3. Eigenvalues/Eigenvectors Calculation

Eigenvalues are very important components of the computation of PCA because it provides a data that correlates to the eigenvectors. Eigenvalues and eigenvectors will be presented in a diagonal n x n matrix, n in this problem is 4. Since each eigenvalue will be in a different column, it also represents the eigenvectors. For example, the first eigenvalue represents the first column of eigenvectors, and so on. The code of obtaining the eigenvector and eigenvalue is simple in MATLAB.

```
[V,D] = eig(covMat);
```

From the code above, V outputs a n x n matrix containing the eigenvectors from the covariance matrix and D outputs a n x n diagonal matrix containing the eigenvalues from the covariance matrix.

1.2.4. Principal Components Extraction

In PCA, we want to rank our eigenvalues in descending order because we get the principal components in order of significance this way. Thus, we will need to sort the eigenvalues and its corresponding eigenvectors from the highest eigenvalue to the lowest eigenvalue. From the function in 1.2.3, the eigenvalue is ranked from lowest to highest. In this case, we can sort them by switching the diagonals of the

matrix. This also applies to the eigenvectors. The eigenvectors in the 4th column will be switched into the 1st column, the 2nd to the 3rd, and so on. Now, we have eigenvectors that ranks the most important components of the data. This will be one of the outputs of the function under the variable `coeffOrth`.

1.2.5. Normalized Data Projection

This step is to compute one of the main outputs of the function, `pcaData`.

```
pcaData = dataNorm * coeffOrth;
```

From the MATLAB syntax above, pcaData is simply gotten from the dot product of the normalized data and the sorted eigenvectors from highest to lowest.

B. Main Script `project_105785820_p1.m`

The main script firstly should load the data from the file `covid_countries.csv`. Since this file is a table, we can use the function `readtable` to get the data from the table in an organized manner. In other words, MATLAB will be able to distinguish from different data types and knows whether it is the data or a variable for the data.

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

The code above allows the .csv file be translated to MATLAB with the preserved naming rule. This means that the variable names from the .csv file will be identical even after it is being translated to matlab.

```
dataDuct = table2array(data(:,3:end));
```

The code above creates an array from the table given in the data. We will only be considering the 3rd column through the 6th column since those data have numbers that we will analyze. Those data will be translated into an array creating a matrix. The size of the matrix is 27 x 4.

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

The next step is to call `myPCA.m` to the main script and get the outputs. We will only need the first two columns from each output because we are only interested in the 2-dimensional data of the PCA.

```
pcaData2D = pcaData(:,1:2); % 2 first columns of pcaData
E2D = coeffOrth(:,1:2);
```

This code above creates a new matrix that only contains the 2-dimensional data from the p-dimensional data.

The next few steps will be the biplot

```
vbls = data.Properties.VariableNames;

% Call the 4 last variables
vbls = vbls(3:end);

% use biplot to plot the results
biplot(E2D,'Scores',pcaData2D,'Varlabels',vbls);
```

The variable vbls is used to create a detailed variable for each corresponding line. This way, it will be easier to visualize the plot using the known variables. These variables are those from the data. Since the only data we will be considering for this are from the 3rd column through the last column, we won't need to include the first two. The last step will be to create a biplot using the 2-dimensional sorted eigenvectors and the PCA from the normalized data and eigenvectors.

1.3. Calculations and Results

The output from this part will be a biplot containing the 4 categories plotted into a 2-dimensional biplot.

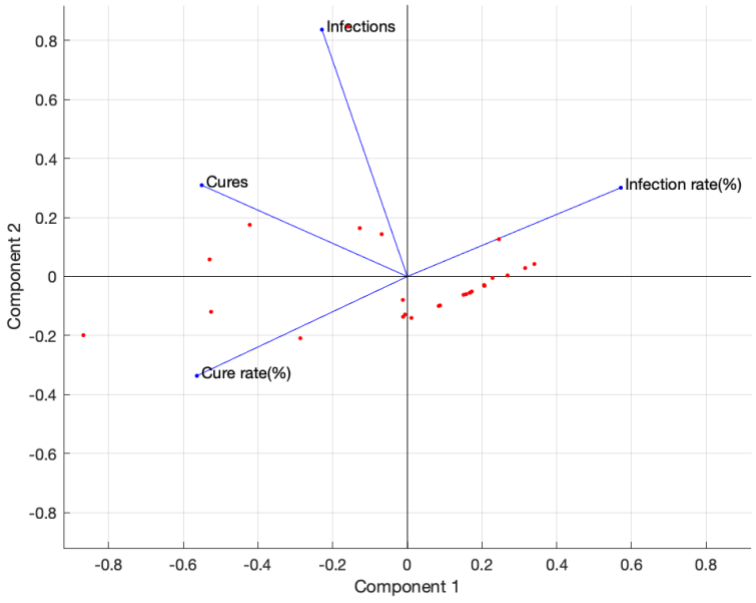

Figure 1 Biplot from Problem 1.

This biplot will be interpreted in the section below. We can see that the biplot has the variables desired from the selected data.

1.4. Discussion

Inside this data, we want to calculate how the distribution of the data from the given table. Since there are essentially four variables which we could evaluate, there will be exactly

four vectors. In the plot 1.3, these are visualized using the blue lines with each variable as its title. We can see that every variable are varying for the first principal component. We can see that the variables cures, cure rate(%), and infections are to the left of the first component and only infection rate(%) is pointing to the right hand side. This means that a majority of the countries from the data has a negative score for the first component.

The second principle component is positive for cures, infections, and infection rate(%). Meanwhile, cure rate(%) is negative. This helps us read the countries that are currently have high values for the first set variables, but lower in the other set. These values are highly determined by what has been found in the eigenvalues, eigenvectors, and the normalized data.

We can also see the angle associated with the data. Specifically looking at the infection rate(%) and cure rate(%), they seem to be at an 180 degree angle. This is due to the fact that these rates are inversely related. Say that the infection rate goes up, then most likely the cure rate goes down. Nonetheless, the infection and cures also has angles in between them that tells readers about the magnitude of those variables from the biplot.

The dots, more commonly known as "scores", shows how each country is doing in terms of the variable. We can see from the biplot that only a few countries have a positive value for both first and second principal component. Most of the dots shown in the biplot fall around the 4^{th} quadrant, where the first component is positive but the second is negative. This means that the country has high cure rate.

The biplot is an important part of study when a bunch of variables are involved. It can evaluate components that are principal in regard to the variables and solve it into a 2-d plot instead of a n-dimensional plot where n are the number of variables. This helps us interpret the data in a better way. The vector and the scores exactly tells readers how the COVID virus has affected each country's cure rate(%), cures, infections, and infection rate(%).

2. Solving the Spatial S.I.R Model

SIR is a model used commonly throughout the world to simulate disease outbreaks. The model will be used on a spherical, triangulated surface to trace the spread of a disease. Each point on the structure represents a location that has a susceptible rate, infection rate, and recovery rate. The spatial SIR model, which is a system of differential equations, can be solved using iterative solvers such as the Runge-Kutta method. The solution to the system of differential equations will be plotted onto a series of plots and animated through the spherical simulation. Finally, data are exported to Microsoft Excel.

2.1. Introduction

The main goal of this problem is to create a functioning code that will simulate a S.I.R model in a triangulated sphere using Bogacki-Shampine 3^{rd} order Runge-Kutta Method to solve the Spatial S.I.R which is a system of differential equations. The solution to the

system of differential equations will be plotted specifically on the results. These only include the monitor locations, or locations of interest. The solution will also be visualized onto an animation of the sphere and illustrate how the susceptible, infected, and recovered rate will change throughout time. Finally, export the data of coordinates with each S.I.R value onto Microsoft Excel. The file will consist of the SIR data every 15 seconds until it reaches the final second. In order to organize the code and increase functionality, the code will be split up into a few functions:

1. Importing the Mesh
2. Implementing the Runge-Kutta Algorithm
3. Implementing the Dynamics Model
4. Solving the Dynamics Model
5. Time Series Plotting
6. 3D Animation
7. Write data to Excel Spreadsheets

These functions are integral to solving the main problem statement.

2.2. Model and Methods

A. Function `stlRead.m`

The purpose of this function is to import the mesh onto the main script from the file modifiedSphereSTL.txt. The mesh itself is a struct that contains two fields, location and neighbors. The main idea of the code is to store each 'corner' data from the data file and put in onto a matrix. Since the data could be repeating, we would want to create another variable to store the non-repeating (unique) location set. This will be important as each repeating node will contribute to the neighbors. Next, index each row (i.e., identify each row as an index where row 1 is index 1, row 2 is index 2, and so on). Once this is done, use the matrix with the repeating nodes and find each member of every index. Every 3 index is going to be a neighbor of each other. Then, the possible neighbors would come from nodes on other indices that have the same node. Thus, the neighbors of that said node will also be the neighbor of the node on the other index. So, in pseudocode:

Load data modifiedSphereSTL.txt
Preallocate matrix for corner data
Preallocate struct for mesh

Use a loop through all data rows to find a 'corner'
If the row is a corner, gather the data of the corner locations (1x3)
Store those locations onto a matrix, B
End loop

C is the nonrepeating nodes of B, use the function `unique` to get the results.

Use a loop through every three locations in B

Create 3 variables to identify each 3 rows in B
Find the index (i.e. the row number of the location in the variable) in matrix C use `ismember` to get this

Store the locations of the node from matrix C
Store the neighbors of the node from the index above and update any new neighbors using `union`

Delete any unused rows from the mesh struct

B. Function `RK4.m`

The function `RK4.m` is used as a method to solve differential equations. The method is called the Bogacki-Shampine 3rd Order Runge-Kutta Method. The general idea of this solving method is to do an iterative method from discretized equations on an adaptive time step. The adaptive time stepping will create a slightly more accurate values of the solution because it adjusts to the slope while also preserving the time taken. This means that we won't know how many time steps it will take for the solution. Since the algorithm is already given on the problem statement, I will attempt to explain the pseudocode of the function and any important lines of code that are useful.

Set constants h0 and e0 to a variable
1st step: preallocate an array for t and y
Set a counter, i
Set t(1) (or t(0) in general mathematics) as the 1st element in timespan
Set tf as the final element in the timespan

Set the initial condition as the 1st column of the preallocated y value

Set the first element of t and y into a new temporary variable tk and yk to track the changes for each loop.

Next is the time marching loop. The loop will go through as long as the time has not reached the final timespan. Thus, a while loop will be used since we do not know how many time elements will be present.

Inside the while loop:
Note: the steps below are described in the Final Project Instructions page 6.

Go through to find the time step size h0 and deterime the new time, t_new
Calculate k1, k2, and k3 using tk, yk, h0, and the function handle

Calculate the y_new using yk, k1, k2, k3, and h0

Calculate k4 using function handle with tk+h0 and y_new as the input
Calculate z_new using k1, k2, k3, yk, and h0

Store the new values, t_new and y_new, onto the preallocated array t and y.

Calculate ek+1 using the norm of yk+1 (y_new) and zk+1 (z_new)
Using ek+1 and e0, calculate the new time step.

```
e_new = norm((y_new) - (z_new), 2);
% from the e value calculate the next step (not constant)
h_new = h0*(e0/e_new)^0.2;
```

Then, update the values of t_new, y_new, h_new to tk, yk, and h0 for the new iteration
Update iteration counter `i = i + 1;`

Since our preallocation arrays t and y will be a 1 * n array, we will need to transpose both t and y to get a column matrix instead of a row.

```
t = t';
y = y';
```

The pseudocode above is a summary of the steps taken to get a running function that could be used to solve system of differential equations. One downside of this method is that the arrays are preallocated with an empty vector that might affect the efficiency of the code when the function is called.

C. Function `dynamicsSIR.m`

This function's goal is to create the system of differential equations that we will then solve. So, essentially this is the main function that will be used in the next function using the Runge-Kutta method developed and the MATLAB built-in code function `ode45`.

The inputs for this function are x, the mesh, alpha, beta, and gamma.
The output for this function is a function in the form of dxdt where x is an independent variable

First, we're going to reshape the mesh since the mesh is a struct.
Reshape x into a length(mesh),3 array.

Preallocate a length(mesh),3 array for the derivatives (dx/dt)

Start the time marching loop to create an dS(1, 2,…, n)/dt dI(1, 2,…, n)/dt and dR(1, 2,…, n)/dt. This will be calculated using the S.I.R dynamics given on the Final Project Instructions page 5.

First, calculate the neighbor contribution. Calculate the number of members each node has. Then, start the neighbor contribution by 0

Create a time loop to go through all the neighbors on each node.

```
for j = 1:M % evaluating each neighbor of a node to calculate
neighbor contribution
    neighborContribution = neighborContribution + (alpha/M)*... %
calcualte the neighbor contribution

(x(mesh(i).neighbors(j),2)/(norm(mesh(mesh(i).neighbors(j)).locat
ion - mesh(i).location)));
    end
```

Note that the neighbor contribution is added on to for each neighbor because of the summation involved in the equation.

Then, we can store each derivative onto their respective array

```
derivatives(i,1) = - (beta * x(i,2) + neighborContribution) * ...
  x(i,1); % the derivative of S, dSdt
derivatives(i,2) = + (beta * x(i,2) + neighborContribution) * ...
  x(i,1) - gamma * x(i,2); % the derivative of I, dIdt
derivatives(i,3) = + gamma * x(i,2); % the derivative of R, dRdt
```

derivatives(i,1) is the ds/dt, (i, 2) is the di/dt, and (i, 3) is dr/dt.

Lastly, we need to vectorize the matrix. In order to do that we will use one line of code

```
dxdt = derivatives(:);
```

Now then form of dxdt is ds1/dt ds2/dt … dsn/dt di1/dt di2/dt … din/dt dr1/dt dr2/dt … drn/dt.

This will be the function handle with independent variable x which will then be solved using `solveSpatialSIR.m` and the ode solvers.

D. Function `solveSpatialSIR.m`

The function discussed below is the main solver of the S.I.R model. The inputs of this function are tFinal, mesh, initial conditions, alpha, beta, gamma, and the ode solver. The outputs are t (time) and x (solution to differential equations).

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

The line above is the series of differential equations with an x variable that will be solved using the spatial S.I.R function. Then, since there's a function presented, we can solve the system of differential equations.

```
[t,y] = odeSolver(dSIRdt, [0 tFinal], initialCondition(:));
```

This will result in a n*1 array of y values that is the S up to the nth value, I to the nth value, and R to the nth value. Thus, we will need to do a reshape in order to get S in

one column, I in one column, and R in one column. To do that, a preallocated 3-d matrix will be preallocated to store the values. Then, do a for loop so that it stores each S, I, and R value to the right row. So, essentially there will be a length(t) of elements, 3 (S, I, and R), and the location nodes matrix.

E. Function `plotTimeSeries.m`

This function is where plots will be displayed as the output. The inputs for this function are the mesh, t, X, and coord. From the mesh, we would need the first field, locations. The location needs to match the coord, which is the monitor locations. When this is the same, we can get its S, I, R values and plot it with time on the x axis and subplots of S, I, and R. So, the pseudocode is as follows:

Calculate the length of X (number of nodes)
Set t0 and tf

For loop for length X loops
If the ith location node is equal to the coord node
Reshape the X matrix at the ith location into a 2-d matrix.

Define figure
Subplot 1
Subplot 2
Subplot 3

End if
End for

Save the figure as a .png file

In the subplots mentioned above, there should be axis labels, grid, and adjustment to the axis. The title would be quite a challenge because for every figure, there will be a different coordinate. Thus, sprintf function can be used for this case so that there's a place holder for the coordinates. This way, the title would output each monitor location correctly.

F. Function `animate.m`

The animate function creates an animation of the SIR model through a spherical simulation with nodes. The goal is to create one that can justify how the S, I, and R model could be simulated for every location on the sphere (i.e., on earth). Initially, we want to start at each monitor location and animate the plot from there.

Pseudocode:

Initialize an array for all the coordinates and the x values.

Create a for loop that will store the location nodes onto the coordinate matrix.
Store the SIR data accordingly so that it has an RGB color format. Blue for susceptible, green for recovered, and red for infected. Store these data onto a matrix called 'color'.

The next step will be the animation procedure. We will need to use a for loop to track the time. So, the for loop goes through all the time elements. But before that, we can initialize a time and increment calculator. This will be used to plot the data not at every time element but at a fixed time increment for efficiency.

If the t(i) is greater than the currTime, then use pcshow.

```
pcshow(coord, color(:,:,i), 'MarkerSize', 500);
```

This function creates a render of the colors at the each time increment. Each vertex is 500pt size. Then, label the axis, initialize a pause every 0.1 and do drawnow. Lastly, update the time increment so that it updates the animation.

G. Function `write2Excel.m`

The data we get through solveSpatialSIR.m will be transferred to a series of spreadsheets on Microsoft Excel. Thus, this function enables us to export the data from MATLAB to Microsoft Excel. The inputs of the function are mesh, t, x, and filename. The pseudocode is as follows:

Calculate the length of t
Calcualate the length of mesh

Initialize column vectors for S, I, and R.
Initialize column vector for coordinates

Create variable names that will be used on the topmost row.

Get the matrix locations from the mesh locations and store them onto the column vector.

Create a time step so that it can be plotted every 15s.

For loop through N time steps
If the time is greater than the timestep
Create a sheet and generate all the data of the location coordinates, S rate, I rate, and R rate.

```
 T = table(corX, corY, corZ, S, I, R, 'VariableNames',
        variableNames);
nameOfSheet = sprintf("T = %f", t(i)); % sheet name
% create table to Excel with table, sheet, and filename
%starting from A1
```

```
writetable(T,filename,'Sheet', nameOfSheet, 'Range', 'A1');
```

The code above creates a table with each data for each columns along with the variable name desired. Write table will export the table to a sheet with all the data and a filename for the Microsoft Excel Spreadsheet.

Lastly update the time for every 15s and repeat the loop.

H. Main Script `project_105785820_p2.m`

All the function files above will be called onto the main script file. For each function, a timer will be used to track the time taken to run the function
The format is below:

```
ticStart = tic;

call function

fprintf('Done in %.3f seconds\n',toc(ticStart));
```

The first function called is the mesh from stlRead.m.

```
mesh = stlRead("modifiedSphereSTL.txt");
```

The input file is already provided but note that this file needs to be in the same folder as the stlRead.m and the main file in order to run.

Next, load the SIRparameters.mat file to get all the constants, initial infections, and monitor locations onto the workspace. Create an initial condition matrix for S I R where S is 1, I is 0 and R is 0. Then, use a for loop to find the location of the initial infection on the mesh and update the SIR value so that S is 0, I is 1, and R is 0.

Next solve the initial conditions using two ode solvers, RK4.m and ode45. RK4.m is the function we developed and ode45 is the built-in MATLAB ode solver.

```
[ty, y] = solveSpatialSIR(tFinal, mesh, initialConditions, alpha,...
    beta, gamma, @RK4);

[ty1, y1] = solveSpatialSIR(tFinal, mesh, initialConditions,
alpha,...
    beta, gamma, @ode45);
```

The code above uses 2 ode solvers to solve the initial value problems. RK4 takes slightly longer time to run than ode45.

The next step is to plot the S rate, I rate, and R rate for the monitor locations. We can easily use the for loop so that it would plot 1 figure for each monitor location. Thus, there will be 3 figures on the output.

```
for i = 1 : numel(monitorLocations)
    % plot the SIR model for each monitor location
    plotTimeSeries(mesh,ty, y, monitorLocations{i});

end
```

Notice that the monitorLocations variable has i so that it would plot for every monitor locations.

Next, call the animate function and the write2excel function. This is fairly easy since we only need to copy the function call onto the main file.

2.3. Calculations and Results

When the main script is run, it will have a few outputs. The outputs will be showed down below:

```
Calling stlRead to load mesh...
Done in 1.747 seconds
Calling SIRparameters.mat and creating initial conditions...
Done in 0.006 seconds
calling solveSpatialSIR with RK4.m...
Done in 8.499 seconds
calling solveSpatialSIR with ode45...
Done in 3.047 seconds
Calling plotTimeSeries at the specified coordinates...
Done in 3.064 seconds
Calling animate to generate animation...
Done in 10.061 seconds
Calling write2Excel to export data...
Done in 1.981 seconds
```

The output above is shown in the command window since it's printed. This shows the amount of CPU time it took to run each respective functions.

The next output is the plots for the SIR solutions for the monitor locations. So, there will be 3 plots.

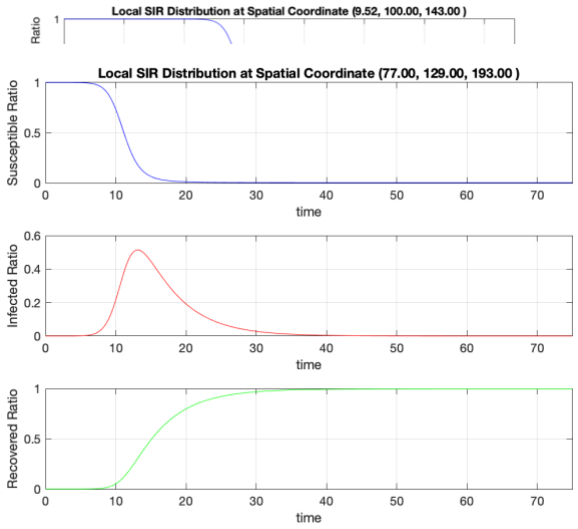

Figure 3 Plot of SIR model for (77.00, 129.00, 193.00)

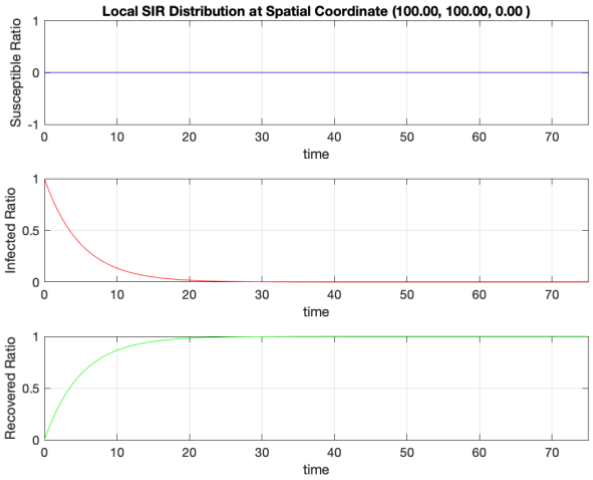

Figure 4 Plot of SIR model for (100.00, 100.00, 0.00)

The figure above shows the S I R ratio of the monitor locations. Notice that each of the locations have a different behavior.

The next will be the output from the animate function. This is multiple snapshots of the function at a specified t value below:

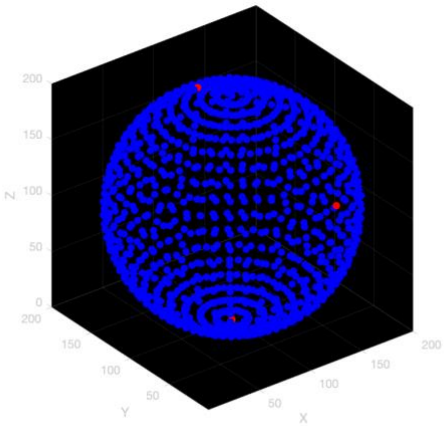

Figure 5 Mesh at t = 0s

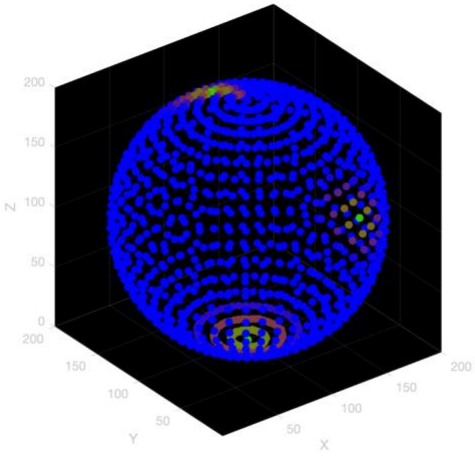

Figure 6 Mesh at t = 8.1456s

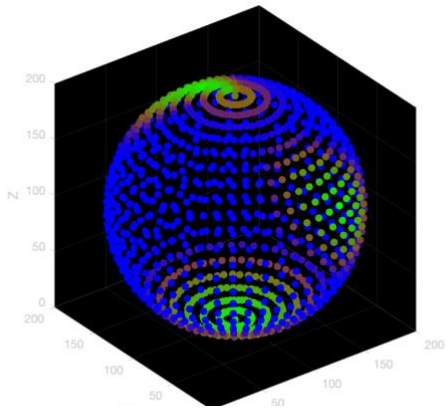

Figure 7 Mesh at t = 17.7447s

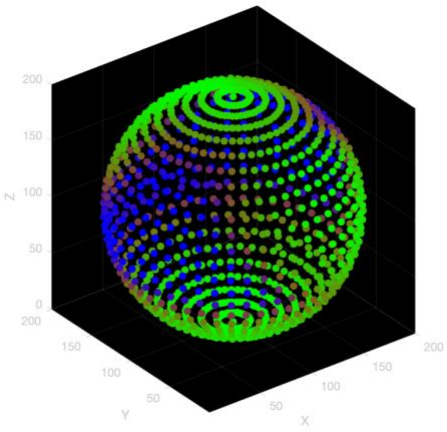

Figure 8 Mesh at t = 26.3848s

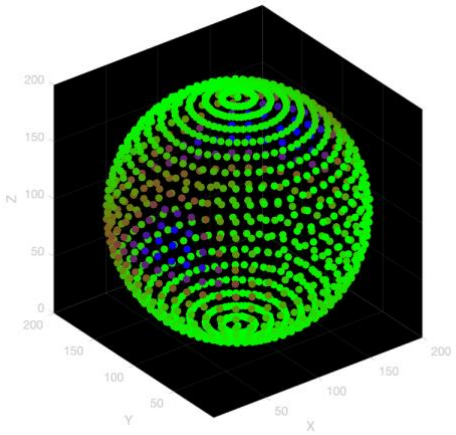

Figure 9 Mesh at t = 34.0637s

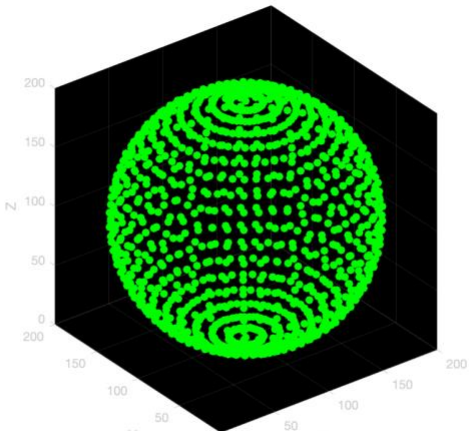

Figure 10 Mesh at t = 75s

From the snapshots above, we can see how the sphere nodes change color from blue, red, and green. This shows how the rates of each S, I, and R change with respect to time. There are 241 time steps, thus there's a lot going on each time step but here, only a few steps that show distinguished figure are shown.

The last output is the Excel file with the name 'SIRData.xlsx'. Since the data is long, I cannot show the excerpt here. However, if the main script is run, there will be an additional Microsoft Excel File that can be opened. As mentioned above, the Excel file has 6 sheets for each t = 15s and each sheet has x, y, and z coordinates with Susceptible ratio, infected ratio, and recovered ratio for every node.

2.4. Discussion

The first file mentioned, stlRead.m, is one of the hardest functions to create in this problem because it requires to convert a .txt file to a matrix to a struct. The locations of the nodes are easy because we simply need to solicit the corner nodes and store them onto a matrix. The neighbors are slightly more challenging since it requires to sort the matrix so that the repeating nodes do not appear twice. Then, the unsorted matrix will be used again to determine the neighbors of the sorted matrix. Only then we can store the locations and neighbors onto the mesh struct.

The RK4.m file solves the spatial SIR model in 8.449 seconds while the built-in MATLAB ode solver, ode45, takes only 3.047 seconds. This is highly expected considering the way the code is written. Ode45 involves very heavy and complex iteration method and calculation once it comes to evaluating the differential equation while RK4.m basically does the Bogacki-Shampine 3rd Order Runge-Kutta Method. For evaluations that involve small sized matrices, RK4.m could solve it in quite adequate time. The SIR model contains 1178 data of SIR for every time step. Thus, the CPU will take more time to go through each node one by one. Another reason is the order of the method. Ode45 uses a higher

method but also uses larger steps when evaluating the data. Thus, less elements will be present and less iterations are needed to solve the differential equations.

Each figure presented on the output shows the SIR ratios for each respective location. For example, at (100,100,0) there's 0 susceptible ratio throughout the time. This means that there are no population at this location. However, due to neighbor contributions, the infected ratio starts at 1 and drops gradually to 0 while the recovered starts at 0 and increases to 1. This means that at the 0^{th} second, there's equal amount of infected with the population. Then, the amount of infected people decreases while the recovered increases. These are inversely related since it can only make sense if the amount of recovered can only increase when there's infected. For the two other specified locations, (77.00, 129.00, 193.00) and (9.52, 100.00, 143.00), have similar behavior. The only difference is when the 'disease' start arriving at their respective locations. For the former, the infection starts arriving around t = 7 seconds while the latter is at about t = 25 seconds. The behavior of both is the infection rate starts from zero and rises for about 40 s then goes down to 0. The recovered rate goes from zero up to 1. The susceptible goes from 1 to 0. This shows that at the end, every single population will have recovered from the infection.

To better show how the infection spreads throughout each location, the animation is used. Note that blue is susceptible, red is infected, and green is recovered. The first animation figure, figure 5, shows that there are 3 locations with infected. Then, as we simulate through the animation, that infection spreads throughout the sphere. This creates a slightly red area of nodes. However, as the red spreads, some of the reds slowly turn to green. In the end, every point node turns to green.

The excel shows how the rate changes for each node. This way, we can keep track of the ratios for S, I, and R for every coordinate. The multiple sheets describe the time when the SIR ratio is evaluated. The values of these ratios vary from 0 to 1.

Throughout this project, many hardships and adversity have been experienced. Many problems in this project are some that I've never seen before so I believe that I have more knowledge now than I did before. This is a simple step to get better and better every day. All in all, this is indeed a very interesting project and the SIR model is indeed a very useful tool to identify and evaluate a spreading disease.

Works Cited

PCA. Analyze Quality of Life in U.S. Cities Using PCA - MATLAB & Simulink. (n.d.). Retrieved December 3, 2021, from https://www.mathworks.com/help/stats/quality-of-life-in-u-s-cities.html.

Senan, N. A. F. (2009). A brief introduction to using ode45 in MATLAB. *ode45berkeley.Pdf*. Retrieved December 3, 2021, from https://www.eng.auburn.edu/~tplacek/courses/3600/ode45berkley.pdf.

Wikimedia Foundation. (2021, December 3). *Principal component analysis*. Wikipedia. Retrieved December 3, 2021, from https://en.wikipedia.org/wiki/Principal_component_analysis.