Diana Zhen Zhang
UID: 805777341
CEE/MAE M20
December 4, 2021

Simulating the Spread of COVID-19

1. Main Component Analysis with PCA

1.1 Introduction

Principal component analysis (PCA) is a dimensionality reduction technique widely used to reduce the dimensions of a large data set by transforming the dimensions of a large data set into small variables containing most of the information found in the large dataset. Of course, the reduction of the variables in the dataset comes at the expense of precision, but the secret to this element reduction is to sacrifice a little accuracy for simplicity. Because smaller data sets are easier and faster for machine learning algorithms without extraneous variables to process. In this problem, the PCA function is developed from scratch and the data extracted is plotted.

1.2 Model and Methods

1.2.a Implementing the PCA Algorithm

This solution demonstrates the effectiveness of PCA for analyzing a dataset by considering COVID-19 data from 27 different countries given in the `covid_countries.csv` file. The analysis is executed in a function `myPCA`, which follows the steps shown in the following graphic[1].

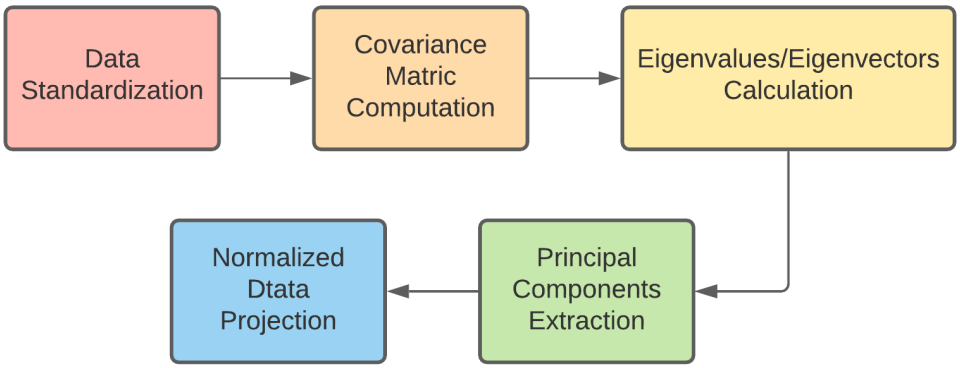

Fig. 1 PCA Algorithm Steps

1.2.a.i Data Standardization

The script for the `myPCA` function opens with the steps of data standardization. Avoiding the pre-programmed MATLAB functions to find mean and standard deviation, the following functions were created to calculate these factors for each column of the data (dimensions n x p).

```
%% Mean computation function
    function [mean] = myMean(data)
        mean = sum(data,1)/length(data);
    end
%% Standard deviation computation function
    function [stdev] = myStdev(data)
        mean = sum(data,1)/numel(data);
        for j = 1:numel(data)
            stdev = sqrt(sum((data-mean).^2)/(numel(data)-1));
        end
    end
end
```

The data is then normalized by subtracting the respective mean from each column and dividing each element by its respective standard deviation. This is implemented as follows.

```
% Normalizing data
XNorm = (data - means) ./ stdev;
```

1.2.a.ii Covariance Matrix Computation

With a standardized data set, the square covariance matrix $C \in \mathbb{R}^{pxp}$ can be computed using the following mathematical definition. The covariance matrix of C is matrix M.

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

Where *m* is the number of rows of *M*. This definition is implemented in the script as shown below.

```
% Covariance Matrix computation
C = (XNorm' * XNorm) / (27 -1);
```

1.2.a.iii Eigenvalues/Eigenvectors Calculation and Principal Components Extraction

From the covariance matrix calculated in the previous step, the eigenvalues and eigenvectors of this matrix are found using the MATLAB `eig` function. This is done to aid the understanding of linear transformations, as the eigenvectors are the "axes" (directions) along which linear transformation acts simply by "stretching/compressing" and/or "flipping". The eigenvalues provide the factors by which the stretching occurs. This process is implemented in the script as shown below.

```
% Find eigenvalues and eigenvectors
[eigVec,eigVal] = eig(C);
[~, idx] = sort(diag(eigVal),'descend'); % Sorting in descending
order
eigVecSort = eigVec(:,idx);
coeffOrth = eigVecSort;
```

This particular solution requires the data to be sorted in descending order, as only the first two columns of data will be taken to make the two-dimensional calculations.

1.2.a.iv Normalized Data Projection

The final step is to project the normalized data onto the previously calculated 2D vector subspace spanned by the two eigenvectors with the largest corresponding eigenvalues. This is done in the script with one line shown below.

```
% Normalized Data Projection
pcaData = XNorm * eigVecSort;
```

1.2.b myPCA Main Script

The main script used to execute `myPCA` first loads the data from `covid_countries.csv`. After the table's titles are preserved and the data is converted to an array using the MATLAB `table2array` function, it is used as an input in `myPCA`

```
data =
readtable('covid_countries.csv','VariableNamingRule','preserve')
;
% Transform to array and preserve titles

% Call myPCA function to obtain data
[coeffOrth1, pcaData1] = myPCA(dataDuct);
```

Then the first two columns of the data are extracted and plotted in two dimension using the MATLAB `biplot` function in the following process.

```
% Only take first two columns of data
pcaData2D = pcaData1(:,1:2);
E2D = coeffOrth1(:,1:2);
% Plot the data
biplot(E2D,"Scores",pcaData2D,"VarLabels",vbls);
```

1.3 Calculations and Results

Upon execution, the following graph is generated from the data.

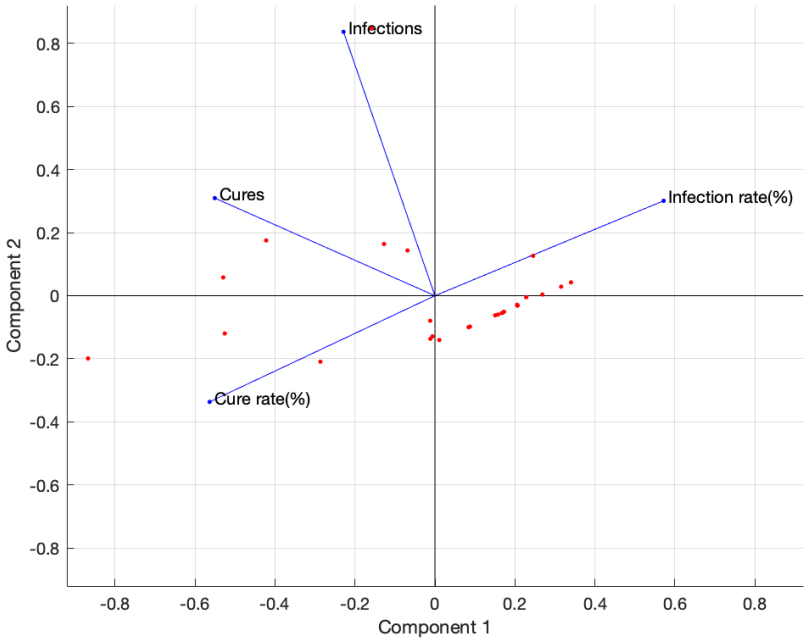

Fig. 2 PCA biplot of normalized covid countries data

This PCA biplot[3] shows both the scores of the sample (the dots) and the loadings of variables (vectors). The further away these vectors are from a PC origin, the more influence they have on that PC. With this plot, the relativity between each variable can also be analyzed. When two vectors form a small angle they are positively correlated, if they meet at a 90º angle, they are likely not correlated, and they are negatively correlated if they diverge and form a large angle (close to 180º).

This reveals that the cure rate (%) and infection rate(%) are negatively correlated, while the number of cures and cure rate (%) are positively correlated. However, the infections and infection rate (%) do not appear to be very strongly correlated, as their vectors form an angle that is around 90º.

1.4 Discussion

In this solution, the data was normalized prior to computing the covariance matrix. But what is normalization? And why is it important? Though there are a variety of explanations and descriptions of what data normalization is, the general concept can be described as a process whereby data within a database is reorganized and manipulated in a way so that the user can properly utilize and implement it for further queries and analysis. In this specific case, the objective of this solution was to demonstrate the effectiveness of principal component analysis (PCA) for analyzing a dataset. A principal value of PCA involves sacrificing a little accuracy for efficiency when working with a large dataset. Normalization allows the PCA algorithm to

operate at its most efficient potential, as all the data is formatted in an easily accessible manner (eigenvectors and eigenvalues).

2. Solving the Spatial S.I.R. Model

2.1 Introduction

This part of the solution focuses on solving the spatial S.I.R. model to simulate and visualize the spread of disease for multiple fixed populations represented by nodes on a mesh. Given a modified version of an STL file representing the meshed surface of a sample sphere as a text file, the data can be manipulated into a mesh, solved using the differential equations[3][4][5] that govern the dynamics of the model, and then visualized.

$$\frac{dS_i(t)}{dt} = -\left(\beta I_i(t) + \frac{\alpha}{|N(i)|}\sum_{j \in N(i)} \frac{I_j(t)}{d(i,j)}\right) S_i(t) \tag{3}$$

$$\frac{dI_i(t)}{dt} = \left(\beta I_i(t) + \frac{\alpha}{|N(i)|}\sum_{j \in N(i)} \frac{I_j(t)}{d(i,j)}\right) S_i(t) - \gamma I_i(t) \tag{4}$$

$$\frac{dR_i(t)}{dt} = \gamma I_i(t) \tag{5}$$

2.2 Model and Methods

2.2.a Importing the Mesh

To import the mesh, a function `stlRead` is written to read the modified STL file provided in which lies the information to generate a sphere composed of triangular facets. This function takes the name of a file and generates a 1 x N array of structs (referred to as nodes for context) that will be the mesh. Each struct contains two fields: a “location” member with a 1 x 3 array containing the cartesian coordinates of a particular node, and “neighbors” member with a 1 x $|N(i)|$ array of neighbor indices. A neighbor is classified as a node that shares a coordinate with another node. For example, all nodes on a singular triangular facet are classified as each others’ neighbors. The indices of the neighbors refers to the index of the node in the final output array.

The algorithm used to append coordinates to each node and identify neighbors is modeled in Figure 6 shown below.

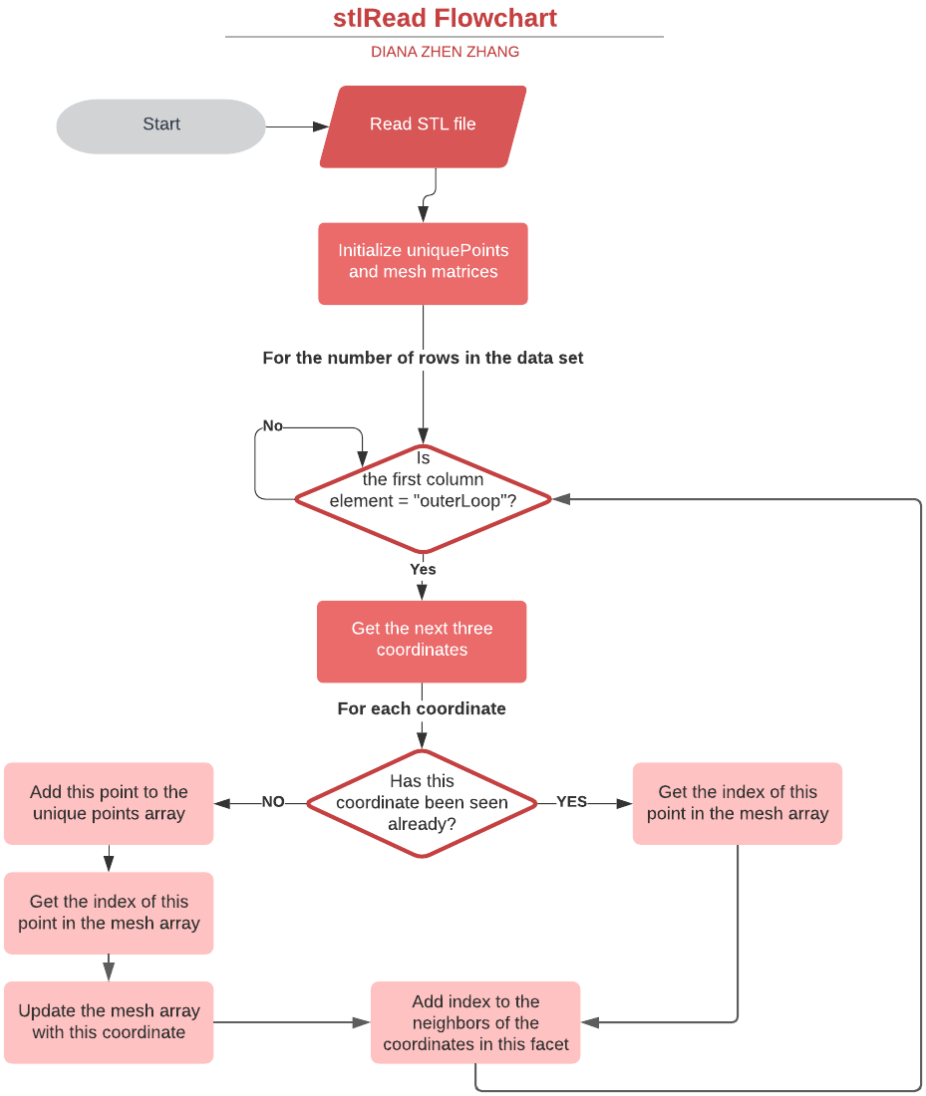

Fig. 6 stlRead function algorithm

2.2.b Implementing the Runge-Kutta Algorithm

The governing equations[3][4][5] are regarded as a dynamical system of the following form[7].

$$\frac{dy}{dt} = f(t, y) \tag{7}$$

Where $y \in \mathbb{R}^n$ is a n-dimensional column vector that represents the state of the system and $f\colon \mathbb{R} * \mathbb{R}^n \to \mathbb{R}^n$ is a function (possibly non-linear) that describes how the rate of change of the state relates to the current state, and t is the independent variable that represents time. The Runge-Kutta algorithms have been developed to take the information about the derivative at multiple time steps, as the forward Euler method does not provide an accurate enough

approximation with the first order calculations. This specific implementation of the Runge-Kutta method is called the Bogacki-Shampine method, as it is a particular implementation of the third-order Runge Kutta method that incorporates adaptive time-stepping.

The script begins by declaring the initial conditions and establishing the time progression, as shown below.

```
% Initialization and declaration
y(:,1) = y0;
t(1) = tspan(1); % t will be a giant column vector
e0 = 1e-4;
yk = y0;
hk = 0.1;
tk = tspan(1);
```

Next, the script loops through the Bogacki-Shampine method of the Runge-Kutta method. The pseudocode for this process can be explained and derived from here. y denotes the numerical solution at time t_n and h_n is the step size defined by $h_n = t_{n+1} - t_n$. One step of this method is given by the algorithm shown below.

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + \tfrac{1}{2}h_n, y_n + \tfrac{1}{2}h_n k_1)$$
$$k_3 = f(t_n + \tfrac{3}{4}h_n, y_n + \tfrac{3}{4}h_n k_2)$$
$$y_{n+1} = y_n + \tfrac{2}{9}h_n k_1 + \tfrac{1}{3}h_n k_2 + \tfrac{4}{9}h_n k_3$$
$$k_4 = f(t_n + h_n, y_{n+1})$$
$$z_{n+1} = y_n + \tfrac{7}{24}h_n k_1 + \tfrac{1}{4}h_n k_2 + \tfrac{1}{3}h_n k_3 + \tfrac{1}{5}h_n k_4$$

2.2.c Implementing the Dynamics Model

To simulate the system using `RK4` or MATLAB's built in ODE solver `ode45`, the states representing the system must be vectorized. In other words, the matrices with two or more dimensions must be expressed as a single dimension, or a vector. Using the MATLAB `reshape` function, the column-major formatted vectors made using the : operator can be re-formed into the corresponding matrix. The function `dynamicsSIR` uses a vectorized version of the state (S, I, or R), an underlying mesh, and model parameters to generate a vectorized time derivative of state. The script for this function begins by reshaping the vectorized states and adding them to a

matrix's columns. This matrix will be called `matReshaped` in this solution. This process is shown below.

```
N = length(mesh); % number of nodes, in this case they are the
rows
matReshaped = reshape(x,[N,3]); % 3 is used because there are
three states (S, I, and R)
S = matReshaped(:,1);
I = matReshaped(:,2);
% R = matReshaped(:,3); Commented for understanding, but unused
```

Next, the calculation of the dynamics begins. After preallocating and array of size N x 3 to hold the derivatives, it is populated by applying the governing equations[3][4][5] to the states with attention to the contribution of the surrounding neighbors, who's indices are obtained by accessing the second field of the *i*-th node in the mesh array. This process is shown below.

```
for i = 1:N
   nbrContr = 0; % Initialize neighbor contribution per loop
   n = length(mesh(i).neighbors); % Number of neighbors
   for j = 1:n
% Computing the distance between two point
            nbrContr = nbrContr + I(mesh(i).neighbors(j)) /
vecnorm(mesh(mesh(i).neighbors(j)).location - mesh(i).location);
   end
   % Differentiate
   derivatives(i,1) = -(beta*I(i) + alpha/n * nbrContr) * S(i);
   derivatives(i,2) = (beta*I(i) + alpha/n * nbrContr) * S(i) -
gamma*I(i);
   derivatives(i,3) = gamma*I(i);
end
```

At the end, the output, `dxdt,` is assigned to a vectorized version of the derivatives matrix.

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

2.2.d Solving the Dynamics Model

Due to the problem's request to compare the ODE solver in this solution to MATLAB's built-in `ode45` function, a function `solveSpatialSIR` is scripted to execute this comparison. It

requires an end time for the simulation, an underlying mesh, initial conditions, and an ODE solver. Because the ODE solver is an input of this function, it allows for easy comparison, as both solvers can be executed using the same function and the same initial conditions.

The script begins by calling the `dynamicsSIR` function found in section 2.2.c. Because the rate of spread is constant in this simulation, *t* (time) is not used in the `dSIRdt` function. This is the function handle that will be passed through the selected ODE solver (`RK4` or `ode45` in this case). The declaration of this function handle is shown below.

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

Once the preferred ODE solver is called, the script loops through the specified time span and calculates the solution for each step using the solution *y* obtained from the ODE solver. These local solutions are reshaped (`dynamicsSIR` produces a vectorized time derivative) and loaded into the output *x* as a matrix of dimensions *N* x 3 x length(t) where *N* is the number of nodes. This loop is provided below.

```
for i = 1:length(t)
   localSol = y(i,:); % Solution at step i
   reshapedLocalSol = reshape(localSol, [N,3]); % Reshape local
solution form array to Nx3 matrix
   x(:,:,i) = reshapedLocalSol; % Load into x matrix
end
end
```

2.2.e Time Series Plotting

In order to visualize a particular local S.I.R. model, a function `plotTimeSeries` can be written to do so when given a specific coordinate. This script will display the susceptible, infected, and recovered rates for three different nodes and automatically save the generated figure as a .png file.

To begin, the script extracts the individual x, y, and z coordinates from the coordinates provided in the input and locates its index within the mesh. With this, it can look through the solution, *X*, input to obtain the solutions of each plane. The matrix `SIR` is a collection of all the solutions for all three variables. These two processes are shown below.

```
% Obtain individual elements of the location given in the input
x = coord(:,1,1);
y = coord(:,2,1);
z = coord(:,3,1);
```

```
% Find index of the coordinate given
locIndx = ismember(vertcat(mesh.location), coord, 'rows');
SIR = X(locIndx,:,:);
```

Next, the script stores the solutions for each variable, extracted from `SIR`.

```
for i = 1:length(SIR)
   susceptible(i) = SIR(:,1,i);
   infected(i) = SIR(:,2,i);
   recovered(i) = SIR(:,3,i);
end
```

Lastly, the plots are generated and saved as .png files under the following title format: `time_series_x_y_z.png` (x, y, and z are replaced with the node's coordinates in each instance). One instance of the plotting sequence and the image generating command are shown below.

```
% Plot susceptible
subplot(3,1,1);
plot(t,susceptible,'r-');
xlabel('Time');
ylabel('Ratio of Susceptible');
xlim([0 max(t)]); % x axis displays numbers from 0 to the
maximum time elapsed

% Save plot as .png file
title = sprintf('time_series_%.3f_%.3f_%.3f',x,y,z);
saveas(f,sprintf('%s.png',title));
```

2.2.f 3D Animation

With the mesh, time, and solutions found using `stlRead`, `RK4`, and `solveSpatialSIR` respectively, an animation can be generated to simulate the spread of the disease across a globe. In this animation, susceptible individuals are represented by blue, infected by red, and recovered by green. The final video will show a blue globe with three red points in the beginning, representing the first three infected, and spread across the surface as each node turns from blue, to red, and eventually all green. The `animate` function opens by allocating arrays and matrices to represent the number of nodes, coordinates of the nodes in the mesh, and the colors to represent S, I, and R.

```
N = length(mesh); % Number of nodes
coords = zeros(N,3); % Coordinates of the nodes in mesh
colors = zeros(size(X)); % Colors representing S, I, and R
```

Then, after obtaining the coordinates of each location in the mesh, the colors matrix is filled by referencing the location in *X,* in which each *N* x 3 matrix corresponds to the state of the S.I.R. system at a particular instance in time.

```
colors(:,1,:) = X(:, 2, :); % Red for infected
colors(:,2,:) = X(:, 3, :); % Green for recovered
colors(:,3,:) = X(:, 1, :); % Red for susceptible
```

Lastly, the point is plotted once for every unit of time to generate the animation.

Though not included in the final script for the sake of efficiency, a video was generated of the animation to include in the calculations analysis portion. This was done using the MATLAB `VideoWriter` object. This video titled 'animateSIR.avi' is saved to the folder in which the `animate.m` script is located.

2.2.g Write Data to Excel Spreadsheets

To store the data from this S.I.R. system, a Microsoft Excel file containing a series of sheets where each sheet stores the nodes' information, infected rate, recovered rate, and susceptible rate at a specific time can be created. This allows for the data to be easily accessible across many platforms, as Excel is made to store, organize, and manipulate data like this.

This script begins by initializing arrays and matrices to hold the number of time steps, locations, S.I.R. rates, and X, Y, Z coordinated. Preallocation is one of the most effective ways to ensure efficiency.

Next, an array of strings containing the titles of the table columns is stored to use in table creation.

```
varNames = ["X Coordinate", "Y Coordinate", "Z
Coordinate","Susceptible rate", "Infected rate", "Recovered
rate"];
```

Now, for each time step, all of its respective S, I, and R rates are stored and vectorized. The coordinates of each point in the mesh are obtained, and each component (X, Y, and Z) is stored independently in its own array. These array assignments are shown below.

```
if t(i) >= targetTime

      S(:) = X(:, 1, i); % All the susceptible cases
      I(:) = X(:, 2, i); % All the infected cases
      R(:) = X(:, 3, i); % All the recovered cases
      for j = 1:n
            xCoord(j) = mesh(j).location(:,1); % Get x
            yCoord(j) = mesh(j).location(:,2); % Get y
            zCoord(j) = mesh(j).location(:,3); % Get z
      end
```

For each time step, which is 15 in this case, a new sheet must be generated. Thus, the table creation sequence occurs inside the loop before it moves onto the next step. At the end of each sheet creation, the target time is updated by 15, indicating that the next sheet will hold solutions that occur 15 time steps after the current. The sheet creation sequence shown below.

```
T = table(xCoord, yCoord, zCoord, S, I, R, 'VariableNames',
varNames);
       sheetName = sprintf('T = %f', t(i)); % Name the sheet
with the current time
       writetable(T, filename, 'Sheet', sheetName, 'Range',
'A1');
```

This will occur until the time that remains is no longer more than the target time.

2.2.h Main Script for Solving the Spatial SIR Model

With the scripted functions, the main script executes and implements them after importing and reading all given initial conditions. First, the mesh is loaded using `stlRead` and the given `modifiedSphere.txt` data as the input. Then the nodes are checked for infection to establish the initial infection. To do this, the distance between nodes is calculated and in the event that the distance is less than zero, then the node of concern is infected. The implementation of this logic is shown below.

```
for i = 1:numInfected
   % Where in the mesh are there infections?
   for j = 1:N % Check each node for infection
       distance = norm(mesh(j).location - initialInfections{i});
% The distance between infected node at i and mesh node at j
       if distance < 1e-10
           indx = j; % This area is infected
           break
```

```
        end
    end
    initialConditions(indx, :) = [0, 1, 0]; % S = 0; I = 1; R = 0
end
```

All of the following functions, except `plotTimeSeries`, are called in the following format[7].

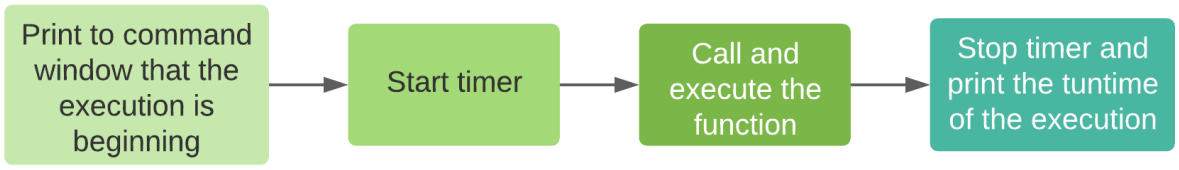

Fig. 7 General function execution sequence

`plotTimeSeries` takes the given `monitorLocations` 1 x 3 cell array that contains three coordinates that will be monitored and plotted. The function is called in a simple loop that applies each element in `monitorLocations`.

```
for i = 1:numel(monitorLocations)
    plotTimeSeries(mesh, tx, x, monitorLocations{i}); % Call
plotTimeSeries.m function
end
```

2.3 Calculations and Results

Upon execution, the program prints to the command window the calling sequence and runtime of each function. From this, the efficiency of `RK4` can be compared against MATLAB's `ode45` function that serves the same purpose. The runtimes of each of these ODE solvers is shown next.

```
Calling solveSpatialSIR with RK4...
Done in 10.249 seconds
Calling solveSpatialSIR with ode45...
Done in 3.688 seconds
```

It is obvious that `ode45` operates a lot more quickly than the written `RK4` function. This is due to the lack of preallocation in the `RK4` script, as arrays and matrices that change size every loop require more time to compute. Due to the versatility that `RK4` must have, it would seem like preallocation would inhibit the applicability. However, an option that allows for both versatility

and preallocation is initializing each array/matrix as one of zeros, then removing any non-replaced elements at the end of the function.

The program also produces the following:

1. Three .png files of the S.I.R. progression of the points given in `monitorLocations`

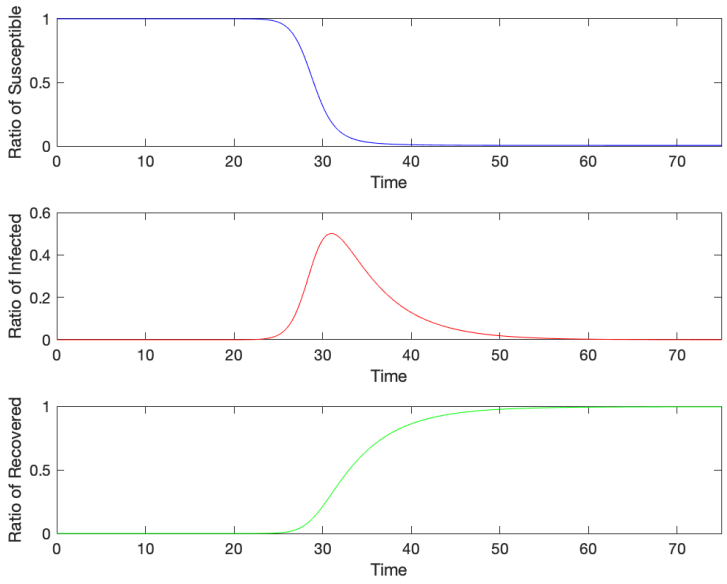

Fig. 8 S.I.R. trends for the node located at (9.520, 100.000, 143.000)

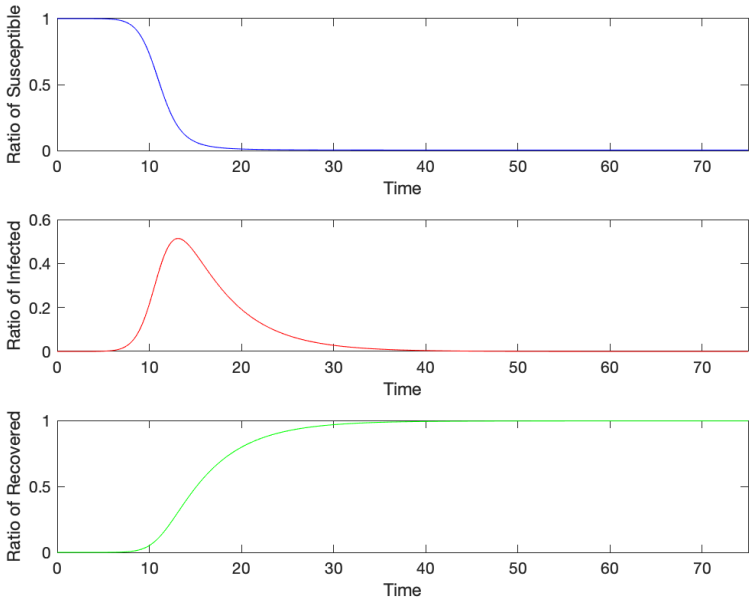

Fig. 9 S.I.R. trends for the node located at (77.000, 129.000, 193.000)

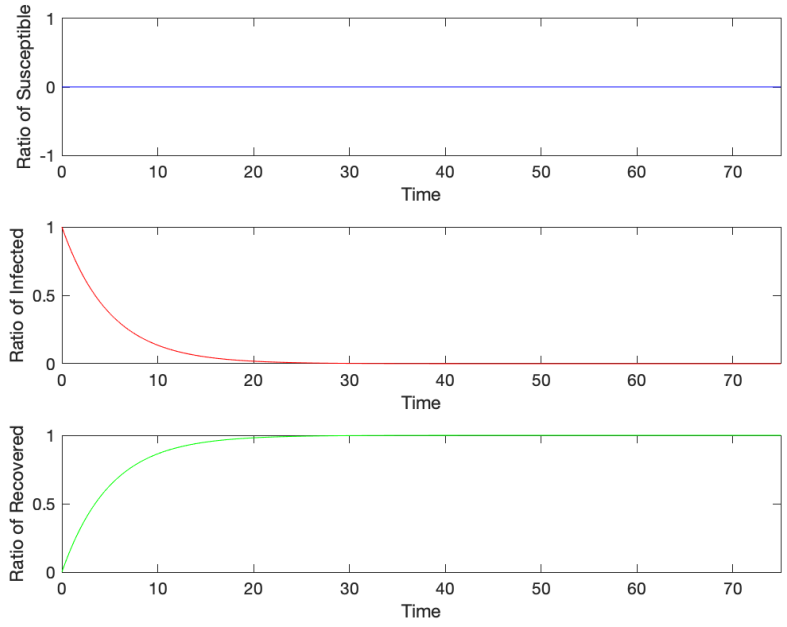

Fig. 10 S.I.R. trends for the node located at (100.000, 100.000, 0.000)

These graphs depict a very clear relationship between infection, susceptible, and recovered ratios. Figures 8 and 9 show a node that is not infected at first, but catches the infection and slowly recovers. Where there is exponential growth in the infected ratio, there is exponential decay in the susceptible ratio. Then, as the infected ratio decreases, the ratio of recovered individuals increases.

However, Figure 10 shows the progression of a point that begins as an infected point. This can be concluded as there is no sign of susceptibility, meaning that this node began as an infected node. This node only experiences the inverse correlation between the infected ratio and recovered ratio because of this.

2. An animation of the S.I.R. system over a globe

The animation can be found here on YouTube for efficiency, though the proper and most accurate animation can only be observed through execution of the program, as the uploading process to YouTube resized the video, making the axes labels nearly invisible. It is suggested that the video be watched at 0.5x its original speed to allow for closer observation of the gradual spread of the disease.

This animation serves an important role in understanding the behavior and progression of the S.I.R. model, as it incorporates the data in a familiar way that is not a graph. The ability to see the three dimensions of this simulation is crucial to understand each component of the program.

3. An Excel spreadsheet with each point's coordinates and S, I, R behaviors at a specific point in time.

This spread is included in the .zip file containing the program, but is not embedded in this section for the sake of conciseness.

The conversion of the data to a spreadsheet is useful for applying this data elsewhere, as Microsoft Excel is programmed specifically for manipulating and storing data such as this.

2.4 Discussion

With so many components and a large dataset, speed is a difficult factor to observe and minimize in this solution. There are parts, such as the coordinate parsing in `stlRead`, that would operate faster in a loop since it is a repeated section of code.

One feature that is helpful when optimizing speed is the `Run and Time` option for running the program. This process is called profiling, and it will benchmark how long it takes for each of the scripts to run. When executed, a graph is displayed along with a table that shows where the majority of the runtime is coming from along with many other useful details such as the frequency at which a certain script is called. These statistics for one execution of this solution are provided below.

Profile Summary (Total time: 37.318 s)

Generated 04-Dec-2021 18:43:45 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)
project_805777341_p2	1	36.991	0.625
solveSpatialSIR	2	15.396	0.049
solveSpatialSIR>@(t,x)dynamicsSIR(x,mesh,alpha,beta,gamma)	1291	15.173	0.020
dynamicsSIR	1291	15.152	15.152
animate	1	11.351	4.096
RK4	1	11.312	0.063
pcshow	36	6.349	0.045
stlRead	1	5.288	0.207
ode45	1	4.034	0.088
initializePCSceneControl	36	4.026	0.015
ismember	12956	3.029	0.178
plotTimeSeries	3	2.880	0.110
ismember>ismemberR2012a	12956	2.771	0.523
initializePCSceneControl>registerCallbacks	36	2.471	0.181
saveas	3	2.349	0.019
print	3	2.254	0.313
unique	32988	1.962	0.970
readcell	1	1.547	0.007
printing/private/alternatePrintPath	3	1.542	0.015
write2Excel	1	1.434	0.030
findall	806	1.362	1.063
rotate3d	72	1.287	0.016
writetable	6	1.264	0.033
rotate3d>setState	72	1.212	0.055
writeXLSFile	6	1.194	0.849
newplot	81	1.152	0.037
activateuimode	72	1.137	0.009
uimodemanager.uimodemanager>uimodemanager.set.CurrentMode	72	1.107	0.014
uimodemanager.uimodemanager>localSetMode	72	1.093	0.024