Kian Mohseni
UID: 505084983
CEE/MAE M20
December 10, 2020

Final Project

1 Main Component Analysis with PCA

1.1 Background and Introduction

In modern-day data analysis and machine learning, there are several different dimensionality reduction techniques used to simplify otherwise complicated datasets. In this problem, we created a function called myPCA that realizes one of these methods called Principle Component Analysis (PCA). We tested the effectiveness of our function by running it on a set of COVID-19 data from 27 different countries. The dataset contains six dimensions and it is the objective of myPCA to reduce its dimensionality by identifying the two principle components that are most relevant. Our main script will call the function and use a biplot to visualize the results and determine the relationships between the six variables (dimensions).

1.2 Implementing the PCA Algorithm

The myPCA function takes in an n x p matrix called "data" that represents the numerical values of the dataset, where n corresponds to the observations or number of countries (27 in this case) and p corresponds to number of variables (6 in this case). It then outputs a p x p matrix called coeffOrth, whose columns are the eigenvectors corresponding to the sorted eigenvalues, and a n x p matrix called pcaData that represents the data projected onto the principle components. When writing the myPCA algorithm, the main steps are:

1. Compute Means and Standard Deviations

   The function begins by using the ***size*** command to determine the number of rows (n) and number of columns (p) in the dataset. It then preallocates three different zero matrices: a p-dimensional vector called "average" that will store the means of each column, a p-dimensional vector called "deviation" that will store the standard deviations of each column, and a n x p matrix called "normal" that will store the normalized data.

   We populate the "average" and "deviation" vectors by running them through a for loop with p iterations and using MATLAB's ***mean*** and ***std*** commands for each column of data.

2. Normalize Data

   Once we've found the means and standard deviations of each column of data, we use two for loops to run through each data point and populate the "normal" matrix. For each element in the nested loop, we subtract the mean of the column from the element of the

corresponding column and then divide that by the standard deviation of that column. This algorithm is shown in Equation 1 below and this centering of the data is used to make the difference among the variables more comparable with one another.

$$X_i^1 = \frac{X_i - \bar{X}}{\sigma} \quad (1)$$

3. Compute Covariance Matrix

   The next step is to use the ***cov*** command on the "normal' matrix to compute the square p x p covariance matrix called "C" from the normalized data. The covariance matrix is used to represent the correlation between each pair of variables or dimensions.

4. Compute Eigenvalues and Eigenvectors

   From the covariance matrix "C", we use MATLAB's ***eig*** command to retrieve the diagonal p x p matrix of eigenvalues called "eigVal" and the full p x p matrix called "eigVec", whose columns are the corresponding eigenvectors.

5. Determine Principal Components

   Now, the function uses the ***abs*** command to determine the absolute values of every element in the matrix of eigenvalues. This new matrix called "A" will also be a diagonal matrix because the absolute values of all the zero elements remains zero. Next we use the ***diag*** command to extract the nonzero diagonal values of the matrix as they represent all the eigenvalues and we store them in a p-dimensional vector called "B".

   Now the function uses the ***sort*** command to sort the eigenvalues in "B" in descending order. The output of the ***sort*** command is a sorted vector and its original indices, but we only require the indices stored in a vector called "index" to tell us the order of eigenvectors in "eigVec". Using "index", we rearrange the columns of "eigVec" in descending order of the absolute value of their corresponding eigenvalues and this sorted p x p matrix is stored in one of the function's outputs called "coeffOrth". Therefore, the principal components of the original data are now stored in the first two columns of "coeffOrth".

6. Project Normalized Data

   The final step of the myPCA algorithm is projecting the normalized data onto the new 2D vector subspace spanned by the determined principal components. We achieve this by multiplying the n x p matrix "normal" by the p x p sorted matrix "coeffOrth" to get a n x p matrix called "pcaData" that is the function's main output. "pcaData" is reduced to 2D instead of p-dimensional and will therefore be easier to visually analyze.

1.3 Main Script

The objective of the main script is to test the effectiveness of our myPCA function on the COVID-19 data from our csv file. The main steps are:

1. Load Data

   The first step of the script is loading the covid_countries.csv file by using the ***readtable*** command and storing the information into a matrix called "data". Using this command, we are also able to eliminate the first row in the csv file which contains the variable names. Next, the script creates a cell array called "categories" that contains all six variable names.

   Now, we use the ***table2array*** command to only take the columns of "data" that contain numerical values and can therefore cut out the first two columns that contain country names and dates. This new n x p matrix is stored in "data1".

2. Run PCA model

   Then the script calls the myPCA function with the loaded data and stores the outputs in "coeffOrth1" and "pcaData1".

3. Plot Data

   The final step of the script is using the ***biplot*** command to visualize the original data projected onto the 2D vector subspace spanning the two principal components or eigenvectors. The "categories" array we established earlier is used to label the plot and when using ***biplot*** we only use the first two columns of the "coeffOrth1" and "pcaData1" matrices.

1.4 Calculations and Results

When the script is executed, Figure 1 is output to the screen.

All six variables are represented in the biplot by a blue vector and the direction and length of the vector indicate how each variable corresponds to the two principal components represented by the x and y axes. For example, the first principal component on the horizontal x-axis, has positive coefficients for every variable except Infection Rate (%) as they are all on the right half of the biplot. This means that those five variables are all positively correlated when it comes to that first principal component. The largest coefficient in the first principal component is Cures as its horizontal distance from the origin is the greatest.

The second principal component on the vertical y-axis of Figure 1 has positive coefficients for Cure Rate (%) and Cures, but negative coefficients for the other four variables. This means that the second principal component distinguishes among countries that have high Cures and Cure Rate, but low values for the other variables and vice versa.

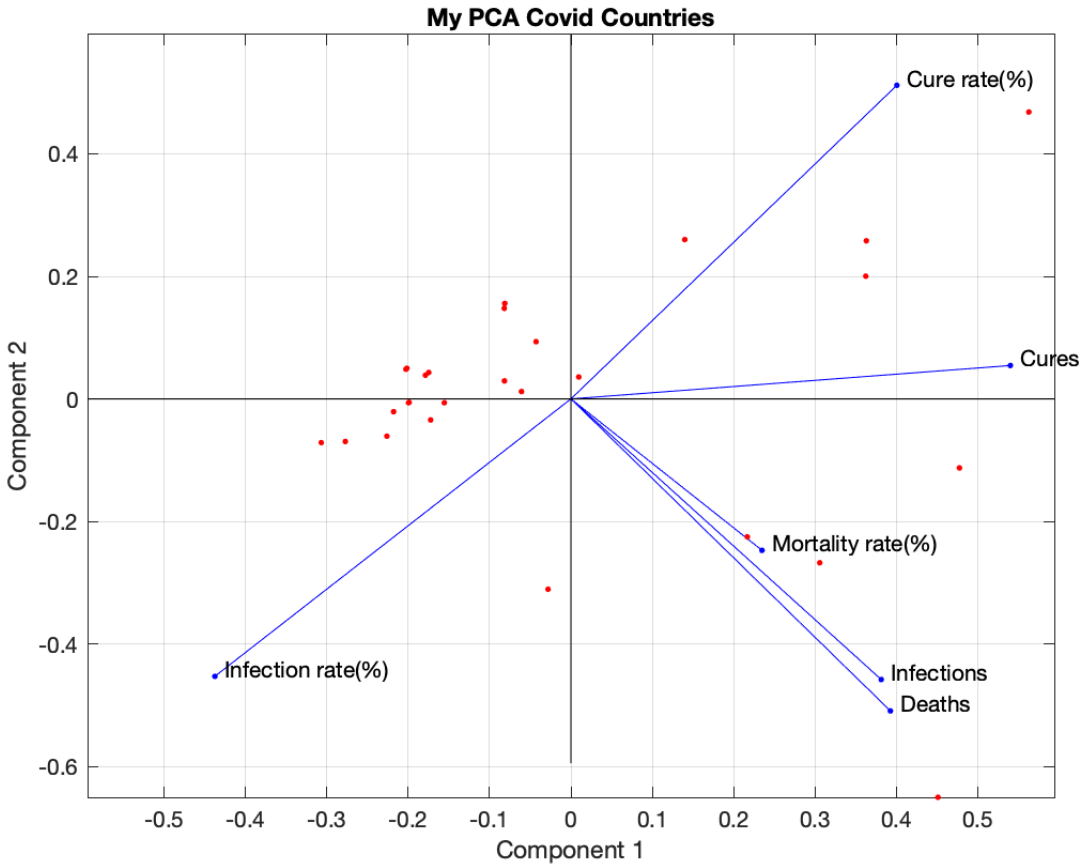

Figure 1: Biplot used to visualize how the six variables included in our COVID-19 data for each country are projected onto two new main components. The goal of reducing the data's number of dimensions is realized.

Moreover, when two vectors are close and forming a small angle like Mortality Rate (%) and Deaths, the two variables are positively related. On the other hand, two variables like Cure Rate (%) and Infection Rate (%) who vectors form a large angle close to 180°, are negatively correlated.

The biplot also includes a red point for each of the 27 observations or countries, whose coordinates indicate the score of each country for the two principal components. All the points are scaled with respect to the other scores and, therefore the plot only shows their relative locations.

1.5 Discussion

This problem clearly demonstrates the advantages of dimensionality reduction used in PCA and other similar algorithms, as it allows us to visually interpret data in practical ways. PCA doesn't discard any of the characteristics of the original data, but rather reduces the overwhelming number of dimensions some datasets contain. The algorithm focuses on a few principal components that can be used to categorize different sets of variables in areas they are related and

areas they aren't related. PCA biplots describe the variation among different variables in a way we can visualize in 2D space and it can be used to predict what factors cause the discrepancies we see.

Furthermore, the myPCA function we wrote is incredibly versatile, because it can be used to analyze a whole range of datasets from biological models to quality of life in different countries.

2 Solving the Spatial S.I.R. Model

2.1 Background and Introduction

In the world of Epidemiology, the S.I.R. model is a popular method that contains three states used to describe the spread of a disease through a fixed population. These states include: the number of susceptible individuals (S(t)), the number of infected individuals (I(t)), and the number of recovered individuals (R(t)), where the sum of all three states is equal to the constant population at any instance in time.

To visualize how the infection spreads spatially over time, we will create a M x N grid where each grid tile, expressed by the coordinates (x,y), corresponds to a local S.I.R. model. The states of every local S.I.R. model are $S_{x,y}(t)$, $I_{x,y}(t)$, and $R_{x,y}(t)$ where $S_{x,y}(t) + I_{x,y}(t) + R_{x,y}(t) = 1$ for every grid tile. We will consider γ to be the infection rate, β to be the contact rate, and α to be the spatial contact rate. Therefore, we can express the dynamics of model using the following differential equations:

$$\frac{dS_{x,y}(t)}{dt} = -[\beta I_{x,y}(t) + \alpha \sum W(i,j)\, I_{x+i,y+j}(t)] * S_{x,y}(t) \quad (2)$$

$$\frac{dI_{x,y}(t)}{dt} = [\beta I_{x,y}(t) + \alpha \sum W(i,j)\, I_{x+i,y+j}(t)] * S_{x,y}(t) - \gamma I_{x,y}(t) \quad (3)$$

$$\frac{dR_{x,y}(t)}{dt} = \gamma I_{x,y}(t) \quad (4)$$

In Equations 2 and 3, W(i,j) is a weighting function that describes how exposed each tile is to nearby neighbors. The values of W(i,j) are defined in Figure 2 below, where W(i,j) = 0 for the tile itself as well as all other tiles outside of the immediate 3 x 3 grid. The weighting function also equals 0 at any tile outside the overall M x N grid boundary.

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

Figure 2: Values of W(i,j) for the location of coordinates in the sub grid. The center tile represents W(0,0) and its 8 immediate surrounding neighbors that contribute to the weighting function at that tile.

We will construct a function called dynamicsSIR that will compute these spatial dynamics and the next step will be to develop an ODE solver that will be able to numerically solve the differential equation that models the dynamics of our S.I.R. system. The differential equation is of the form shown in Equation 5.

$$\frac{dy}{dt} = f(t, y) \tag{5}$$

Given that the data we will analyzing has four dimensions, we will need to use the Fourth-Order Runge-Kutta Method to create a function called RK4. RK4 will behave similarly to MATLAB's built-in ode45 function and will implement the following algorithm given in Equations 6-11.

$$k_1 = hf(t_n, y_n) \tag{6}$$

$$k_2 = hf(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \tag{7}$$

$$k_3 = hf(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \tag{8}$$

$$k_4 = hf(t_n + h, y_n + k_3) \tag{9}$$

$$t_{n+1} = t_n + h \tag{10}$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{11}$$

Here, k_1, k_2, k_3, and k_4 represent the four slopes used to calculate the derivative of y at each instance in time. RK4 will take in the output of our dynamicsSIR function and then another

function called solveSpatialSIR will be called to create a fourth-dimensional matrix representing the state of the grid vs time.

Finally, we will visualize our results by creating 2D plots of the three S.I.R. states against time for different spatial locations on the grid. We will also use a 2D animation to analyze the spread of infection throughout the grid overtime.

2.2 Implementing the Fourth-Order Runge-Kutta Algorithm

The RK4 function is designed to numerically solve higher order differential equations using the Fourth-Order Runge-Kutta algorithm with a fixed time step equal to 0.1 seconds. The inputs of the function include: a function handle of f(t,y) called "f", a 1 x 2 array containing the start time and end time of the simulation called "tspan", and the initial conditions for the differential equation called "y0". The outputs are: a T x 1 vector called "t" that represents the time sequence, and a T x n matrix called "y" that represents the solution of the differential equations. Here "T" is the number of timesteps and n is the number of dimensions (in our problem it is 4).

The main steps used in this function are the following:

1. Preallocate Outputs and Initial Conditions

   The first step in the function is to initialize the time step size as 0.1 seconds and to define one of our outputs, a time vector called "t" that spans from the first element to the second element of "tspan". Then we use the ***transpose*** command to turn "t" into a column vector.

   Now, we can preallocate a zeros T x n matrix to represent our second output "y", where T is the length of the time vector "t" and n is the length of the initial conditions vector "y0". Next, we make the first row of "y" equal to "y0" because it represents the conditions of the system at 0 seconds. Then the script defines a new variable called "y_n" equal to "y0", but the value of "y_n" will be updated after each iteration in the for loop. Finally, we define the number of steps called "nSteps", which will be used in the solver to be 1 smaller than the length of the time vector "t" because we don't count 0 seconds as one of the steps.

2. Loop Through Matrices Using Runge-Kutta Equations

   Now, the function implements the Equations 6-11 by running a for loop with "nSteps" iterations to calculate every row of "y", where each row represents the state of the system at a particular instance in time.

   We create a new variable called "t_n" to update for each iteration and represent the next value in the time vector, following the method shown in Equation 10. After running the four slope equations given in Equations 6-9, an intermediate variable called "y_nplus1" is used to store the new value of "y_n" and it follows the method shown in Equation 11. For every k^{th} iteration in the loop, we store the value of "y_nplus1" into the $(k+1)^{th}$ row of the

"y" matrix. Finally, the function places the value of "y_nplus1" back into "y_n", to be used in the next iteration of the loop.

Once the loop terminates, our RK4 function will output the time vector "t" and the solution "y".

2.3 Implementing the Dynamics Model

Our dynamicsSIR function is designed to compute the rate of change of the model and vectorize all the states used in our system to be compatible with the ODE solvers (RK4 and ode45).

The function's inputs include: a one-dimensional column vector of length 3*M*N called "x", "M" and "N" which represent the size of the grid, and the model parameters given by "alpha", "beta", and "gamma". The only output is a one-dimensional column vector of length 3*M*N called "dxdt" that represents the vectorized time derivative of state. When running the RK4 or ode45 solver, dynamicsSIR is input as the function handle "f".

The main steps of the function include:

1. Devectorize Input

   We begin by devectorizing "x" and using the ***reshape*** command to turn it into a 3D M x N x 3 matrix called "devectorized". Next, we extract the first and second layers of "devectorized" to represent the data of susceptible and infected individuals in the grid. These new matrices are stored in "S" and "I", respectively.

   Now we initialize a 2D matrix called "W" that will store the summation of the weighting function multiplied by the number of surrounding infected individuals at each coordinate in the 2D grid.

2. Calculate Weighting Function

   Next, the function uses two for loops to run through each coordinate in the grid to calculate W(i,j) and store it in the matrix "W". We'll need to account for special boundary cases like the corners and edges of the grid where there will be less than 8 surrounding neighbors, by using several nested if statements.

   For example, the bottom left corner of the grid is where i=1 and j=1, so we know that there will only be 3 neighbors to consider in that location. Looking back at Figure 2, we can determine the values of the weighting function for these three neighbors and multiply them by their corresponding infection state "I". This means that W(i,j) at the bottom left corner is equal to the sum of the "up", "upright", and "right" neighbors. For example, "upright" in this case is equal to $\frac{I(i+1,j+1)}{\sqrt{2}}$ because the value of the weighting function there is $\frac{1}{\sqrt{2}}$.

3. Compute Dynamics

   Once we have computed "W" for all the locations in the grid, we can implement Equations 2-4 to compute the dynamics of the system for all three states and we initialize a 3D M x N x 3 matrix called "derivatives" to store these values. We use two for loops to run through all the grid locations and populate "derivatives", with the first layer corresponding to the susceptible state, the second layer corresponding to the infected state, and the third layer corresponding to the recovered state.

4. Vectorize Dynamics

   Finally, the last step in the function is to vectorize "derivatives" so it is compatible with the ODE solvers RK4 and ode45. The vectorized state is called "dxdt" and is the only output of the function.

2.4 Solving the Dynamics Model

Our next task is creating a function called solveSpatialSIR that is designed to simulate our spatial S.I.R. model using and ODE solver function handle (RK4 or ode45) as one of the input arguments.

The inputs of the function include: the end time for the simulation called "tFinal", a function handle for an ode45-compatible solver called "odeSolver", a M x N x 3 matrix called "initialCondition" that sums to 1 in the third dimension, and model parameters given by "alpha", "beta", and "gamma". The outputs are: a vector of timesteps called "t" and a 4D M x N x 3 x length(t) matrix called "x" that represents the state of the system vs time.

The main steps in writing the function include:

1. Infer Initial Conditions

   We begin by inferring the values of "M" and "N" using the size of the "initialCondition" matrix. We then vectorize "initialCondition" and store it in "y0" so that it is compatible with the RK4 and ode45 solvers. Assuming the start time of the simulation is 0 seconds and using "tFinal", we can also define the array "tSpan" that will be used as an input of the odeSolver function.

2. Wrap Function

   Next we can call the dynamicsSIR function and use "x", "M", "N", "alpha", "beta", and "gamma" as input arguments. It will be used as the function handle "f" for the odeSolver and the other two input arguments will be "tspan" and "y0". Therefore, we are wrapping the odeSolver function around the dynamicsSIR function and the outputs of odeSolver are a vector of timesteps called "t" and a 2D T x n matrix called "out".

3. Reshape Output

   Finally, we need to reshape the output matrix "out" from a 2D T x n matrix into a 4D M x N x 3 x length(t) matrix. First we initialize a matrix called "x" with the correct 4D dimensions. Next the function loops through the length of "t", and uses the ***reshape*** command to reshape the corresponding row of "out" to a 3D M x N x 3 matrix representing the state of the system. This state is then stored in the first three dimensions of "x" for every timestep, giving us a 4D matrix output that represents the state of the system vs time.

2.5 Time Series Plotting

Now we want to visualize the system, so we created a function called plotTimeSeries that is designed to visualize a particular local S.I.R. model by specifying its spatial coordinates (x,y) and displaying how the ratio of infected, recovered, and susceptible individuals changes over time at that location on the grid.

The inputs include: a vector of timesteps called "t", the spatial x coordinate on the grid called "x", the spatial y coordinate on the grid called "y", and a 4D M x N x 3 x length(t) matrix called "X". Each grid tile in the M*N space represents a local S.I.R. model whose values are between 0 and 1 and the local model is repeated for every timestep. The input "X" is retrieved from the output "x" of the solveSpatialSIR function. The function has no outputs, but it creates a plot of each state vs time.

The main steps needed to write the function are:

1. Determine States

   The first step in the function is to call the ***figure*** command to generate a new figure each time the function is run and store it in a variable "h". Next we initialize column vectors of length "t" called "S", "I", and "R" that will store the ratios of susceptible, infected, and recovered individuals, respectively. Now we run a for loop and iterate through the length of "t" to populate "S", "I", and "R" at the grid coordinates specified in the inputs "x" and "y". The first layer of the 4D "X" matrix is stored in "S", the second layer is stored in "I", and the third layer is stored in "R" at every timestep.

1. Subplot

   Now, the function uses the ***subplot*** command to create a 3x1 array of plots in the same figure. From top to bottom, the figure includes the plots of ratio of susceptible individuals vs time, ratio of infected individuals vs time, and ratio of recovered individuals vs time. We also use the ***sprintf*** command to programmatically include the (x,y) location in the title string of each plot. Knowing the ratio of any state can never exceed 1, we can also set the y-axis limits as 0 and 1 to better visualize the plots. Finally, we use the ***saveas*** command to save the generated figure of plots as a png file with the grid coordinates (x,y) again included in the file name.

2.6 2D Animation

The final function we need to include is called animate and it is designed to provide a 2D visual animation of how the disease spreads through the population over time. We will use red pixels to represent the infected, green for recovered, and blue for susceptible individuals.

The only input of the function is a 4D M x N x 3 x length(t) matrix called "X", the same as the input "X" taken in plotTimeSeries, where "X" is the output of the solveSpatialSIR function.

The main steps taken to create the function are:

1. Determine Color Matrix

   The first step is retrieving the size array of the input matrix "X" and assigning its first and second elements to "M" and "N", respectively. Then we initialize a 4D matrix called "color", whose dimensions are the same as those of "X". Next we set the first, second, and third layers of "X" to "St", "It", and "Rt", respectively and these represent the susceptible, infected, and recovered individuals.

   Now we set the third layer of the "color" matrix equal to "St", the first layer of "color" equal to "It", and the second layer of "color" equal to "Rt". This way we are assigning the susceptible individuals to display as blue, the infected individuals to display as red, and the recovered individuals to display as green. Next we use the ***figure*** command to generate a new figure each time the animate function is run.

2. Run Animation

   Finally, the function runs a for loop to iterate through the "color" matrix a total of $\frac{length(t)}{10}$ times. Inside the loop, we perform the ***squeeze*** command within the ***image*** command to generate an image for every iteration and eliminate any single dimension in the "color" matrix. Within these two commands is the "color" matrix, whose fourth dimension is t*10 to ensure we are only displaying the image at every 10^{th} timestep and thus only having 60 frames instead of 600. Finally, the function uses the ***pause*** command to create a pause of 0.1 seconds before changing to the next image in the animation. This will slightly elongate the animation, allowing the user to see what is going on. The animation should therefore take about 6 seconds to run.

2.7 Main Script

The objective of the main script is to solve a spatial S.I.R. model on a 50 x 75 grid using our solveSpatialSIR function and comparing the performance when using our RK4 implementation against MATLAB's built-in ode45 solver. We will also visualize the data using our plotTimeSeries and animate functions.

The main steps in the script include:

1. Load Data

   Our first step in the main script is loading our provided "initialValues.mat" file using the ***load*** command. After loading the data, a 3D M x N x 3 matrix called "initialConditions" will be stored in our Workspace and we can use the ***size*** command to extract its first two dimensions and set them equal to "M" and "N", respectively. We will also define the model parameters as the following:

$$\alpha = 1 \quad (12)$$
$$\beta = 0.5 \quad (13)$$
$$\gamma = 0.1 \quad (14)$$
$$t_{final} = 60 \quad (15)$$

   These parameters as well as "M" and "N" will be used in our dynamicsSIR function as input arguments.

2. Solve Spatial S.I.R. Model

   Now, the script needs to run the solveSpatialSIR function on both our RK4 solver and on MATLAB's built-in ode45 solver. We will use the ***tic*** and ***toc*** commands to benchmark the runtimes for both solvers and use the ***fprintf*** command to print both their results to the output window. The outputs of solveSpatialSIR when running the RK4 solver are stored in "t_RK4" and "x_RK4" while the outputs when using the ode45 solver are stored in "t_ode45" and "x_ode45".

3. Plot Data

   The next step is calling our plotTimeSeries function on the system solved with our RK4 solver at three spatial grid coordinates: (1,1), (5,18), and (30,70). Therefore, "t_RK4" and "x_RK4" will be used as input arguments in the function.

4. 2D Animation

   Finally, we call the animate function on the system solved with our RK4 solver and generate a 2D visual animation of the system with "x_RK4" as the input argument.

2.8 Calculations and Results

When the program is executed, the following output and figures are printed to the screen.

```
The runtime results when using the RK4 solver are 0.9799 seconds.
The runtime results when using the ode45 solver are 0.2596 seconds.
```

Figure 3: The script prints the benchmark runtime results when using both the RK4 ODE solver and MATLAB's built-in ode45 solver.

Figure 3 exhibits our solveSpatialSIR function taking 0.9799 seconds to run when using our RK4 solver and 0.2596 seconds when using the ode45 solver. The values of these runtimes slightly vary for every trial, but using the ode45 solver is consistently about 4 times faster than using our RK4 implementation. This ratio makes logical sense because the length of time vector "t_ode45" is 161 whereas the length of time vector "t_RK4" is 601. This means that solveSpatialSIR has to compute about 4 times more data when running the RK4 solver than when running the ode45 solver, which would take about 4 times more time. The ode45 function determines its own timesteps based off default tolerance values, so it is natural for its step size to be larger than that of RK4 (0.1 seconds), causing it to use a lower number of steps.

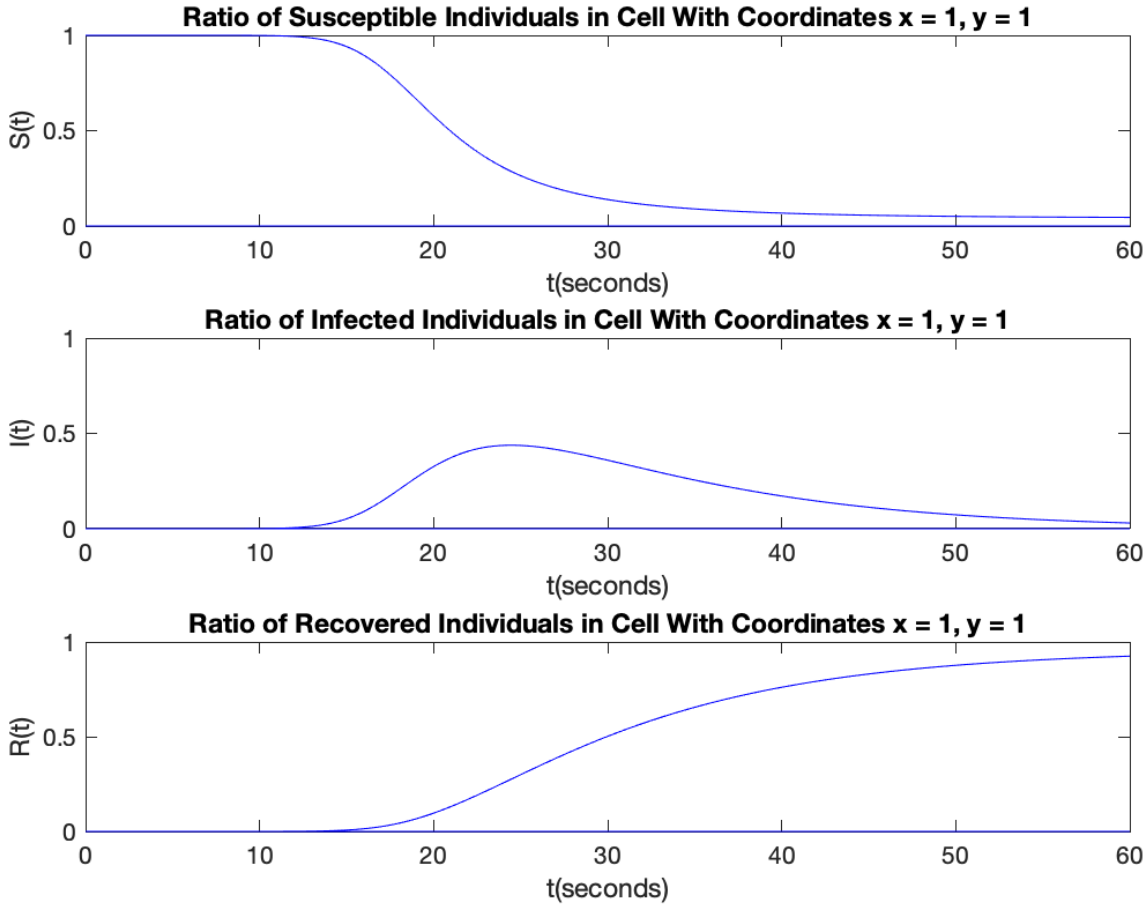

Figure 4: The three subplots display how the local S.I.R. distribution at the spatial grid location (1,1) changes from t=0 seconds to t=60 seconds.

Figure 4 includes the three plots that show how the local S.I.R. model at (1,1) changes over the course of the simulation from t=0 seconds to t=60 seconds. We can see that $S_{1,1}(0)=1$, $I_{1,1}(0)=0$, and $R_{1,1}(0)=0$ therefore the tile started out with 100% of the local population being susceptible individuals. The ratio of infected individuals peaks at around t=25 seconds where $I_{1,1}(25)=0.5$, meaning about 50% of the population is infected. By the end of the simulation at t=60 seconds, almost 100% of the local population has recovered from the disease and the ratio of infected and susceptible individuals is 0.

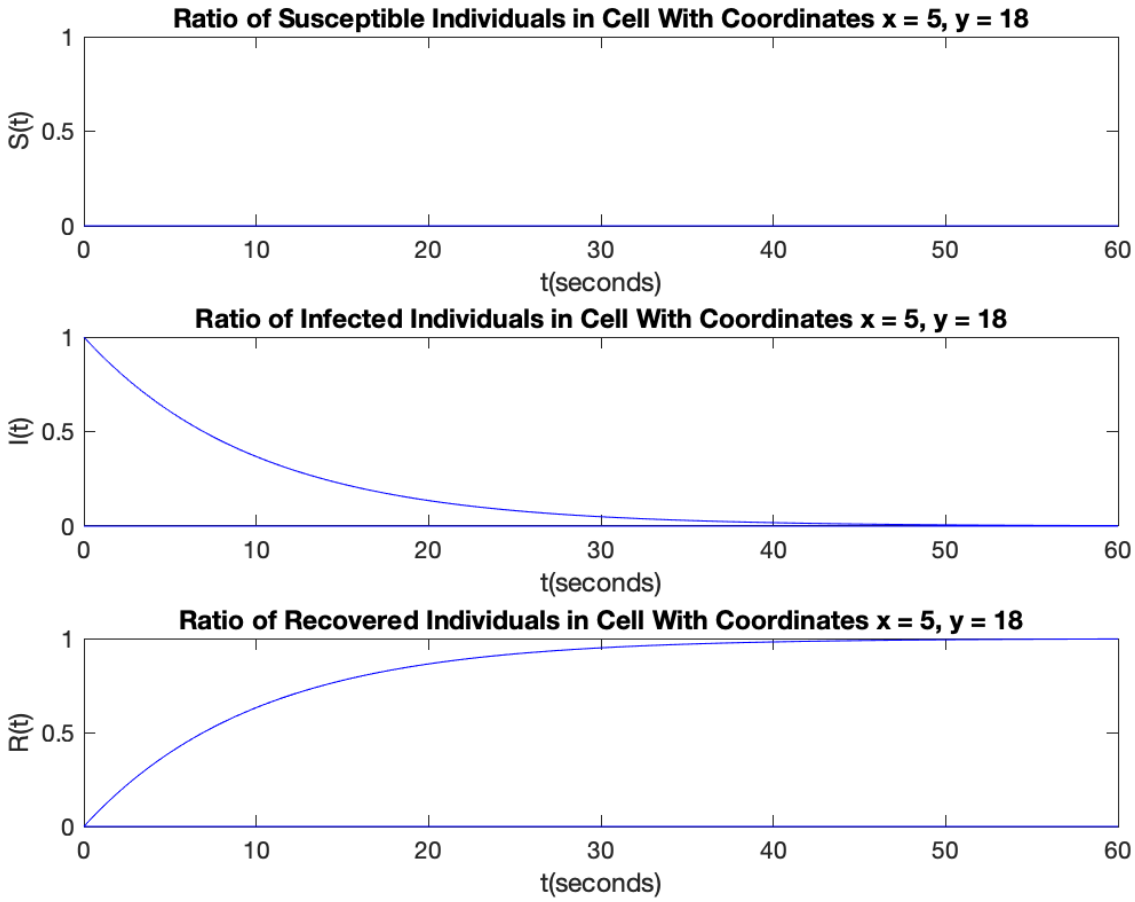

Figure 5: The three subplots display how the local S.I.R. distribution at the spatial grid location (5,18) changes from t=0 seconds to t=60 seconds.

In Figure 5, we now see the three plots that show how the local S.I.R. model at (5,18) changes over the course of the simulation from t=0 seconds to t=60 seconds. It's clear that $S_{5,18}(0)=0$, $I_{5,18}(0)=1$, and $R_{5,18}(0)=0$ therefore the tile started out with 100% of the local population being infected individuals. Because the local model began with the entire population being infected, none of the local population was ever susceptible during the course of the simulation and $S_{5,18}(t)=0$ at all values of t. By the end of the simulation at t=60 seconds, 100% of the local population has recovered from the disease and the ratio of infected and susceptible individuals is 0.

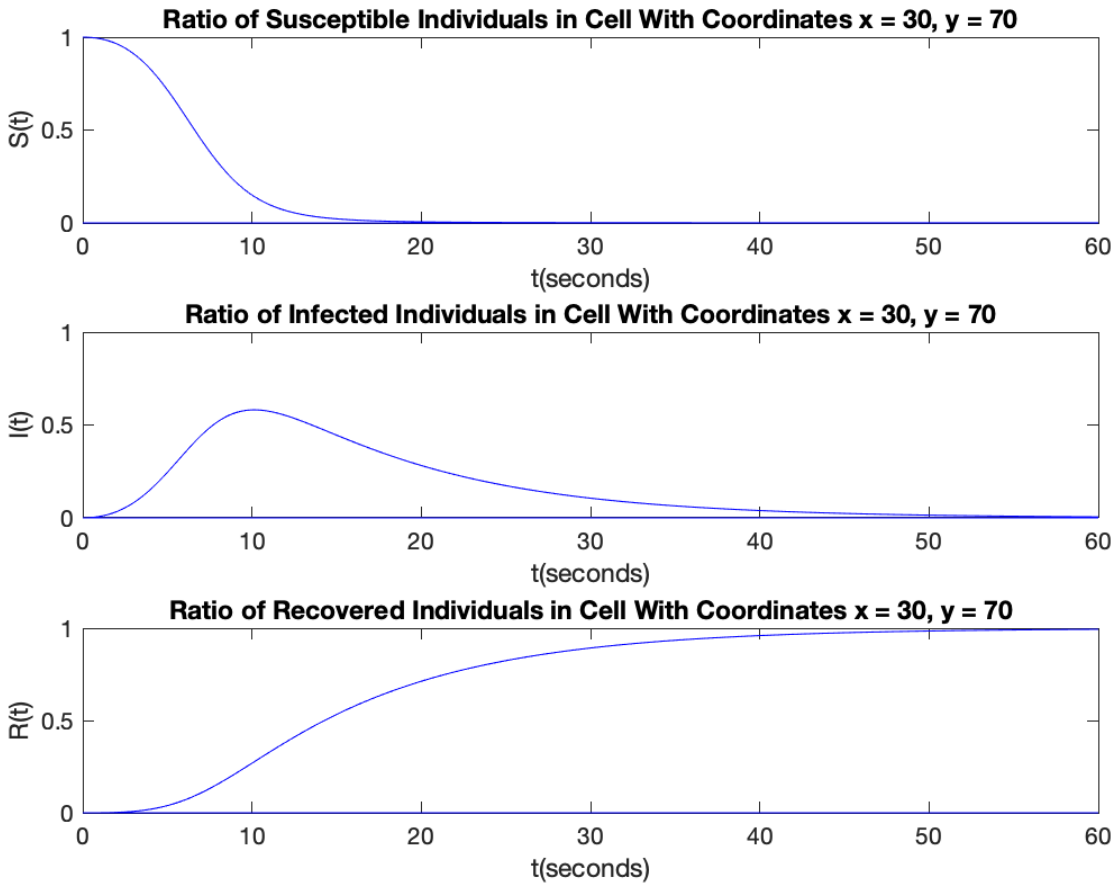

Figure 6: The three subplots display how the local S.I.R. distribution at the spatial grid location (30,70) changes from t=0 seconds to t=60 seconds.

In Figure 6, we finally see the three plots that show how the local S.I.R. model at (30,70) changes over the course of the simulation from t=0 seconds to t=60 seconds. It's clear that $S_{30,70}(0)=1$, $I_{30,70}(0)=0$, and $R_{30,70}(0)=0$ therefore the tile started out with 100% of the local population being susceptible individuals. The ratio of infected individuals peaks at around t=10 seconds where $I_{30,70}(10)=0.6$, meaning about 60% of the local population is infected. By the end of the simulation at t=60 seconds, 100% of the local population has recovered from the disease and the ratio of infected and susceptible individuals is 0.

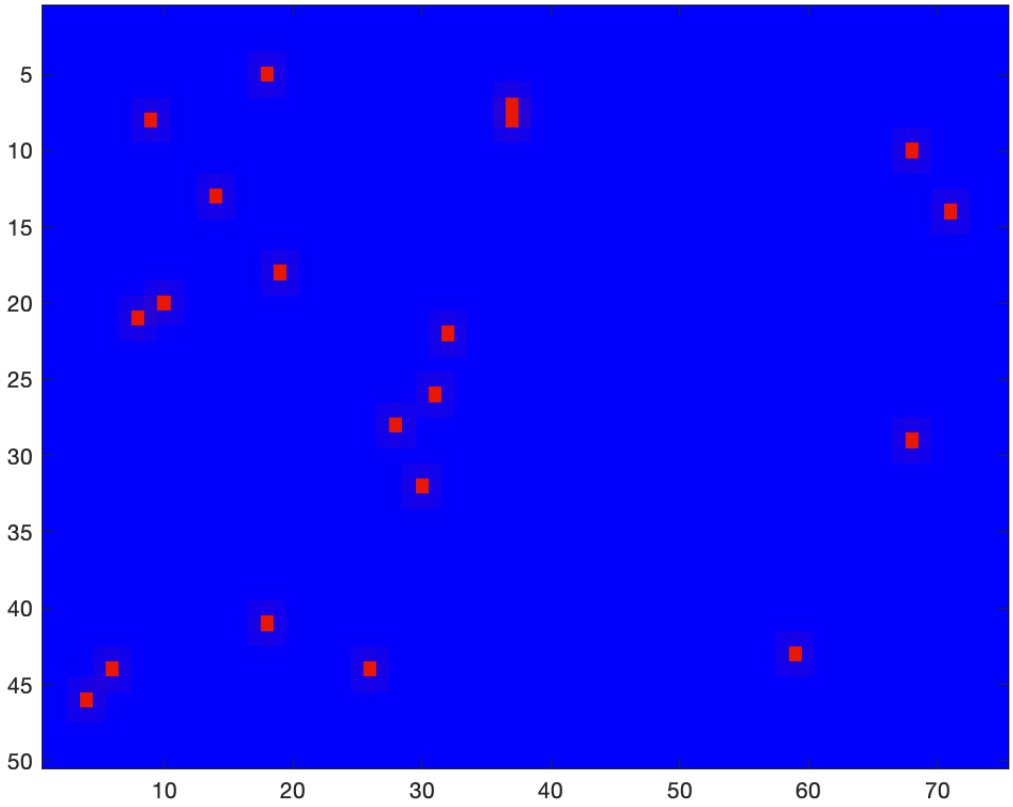

Figure 7: The 2D animation at t=10 seconds exhibits the evolution of S.I.R. states within the grid. The blue pixels represent susceptible individuals and the red pixels correspond to areas on the grid where the infection is beginning to spread.

Figure 7 demonstrates the states present in the 50 x 75 grid at t=10 seconds in the simulation. We can see the red pixels representing the birth of the infection at different locations on the grid, yet the majority of the grid still consists of susceptible individuals, as represented by the sea of blue pixels.

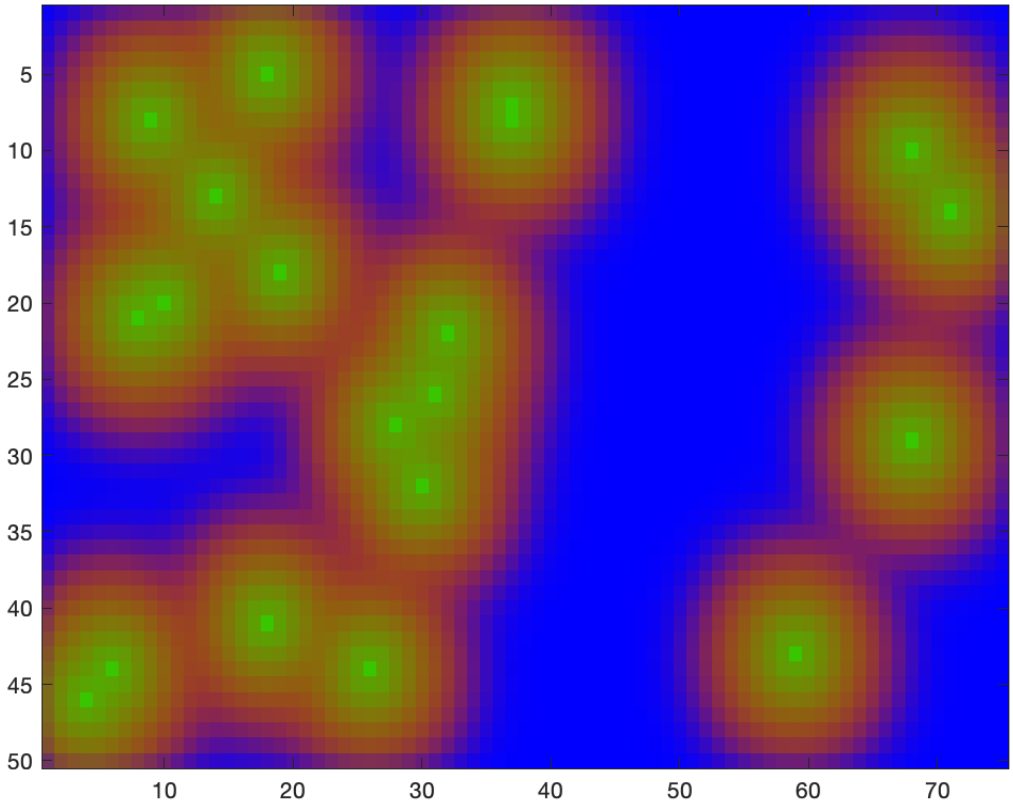

Figure 8: The 2D animation at t=150 seconds exhibits the evolution of S.I.R. states within the grid. The red pixels show that the infection is rapidly increasing throughout the grid.

In Figure 8, we can see the states of the grid at t=150 seconds, a quarter way through the simulation. The red pixels from Figure 7 have rapidly multiplied in area and the infection is expanding throughout the grid, in all directions. The green pixels at the center of every red bubble represent the individuals who were originally infected with the disease, but have now recovered.

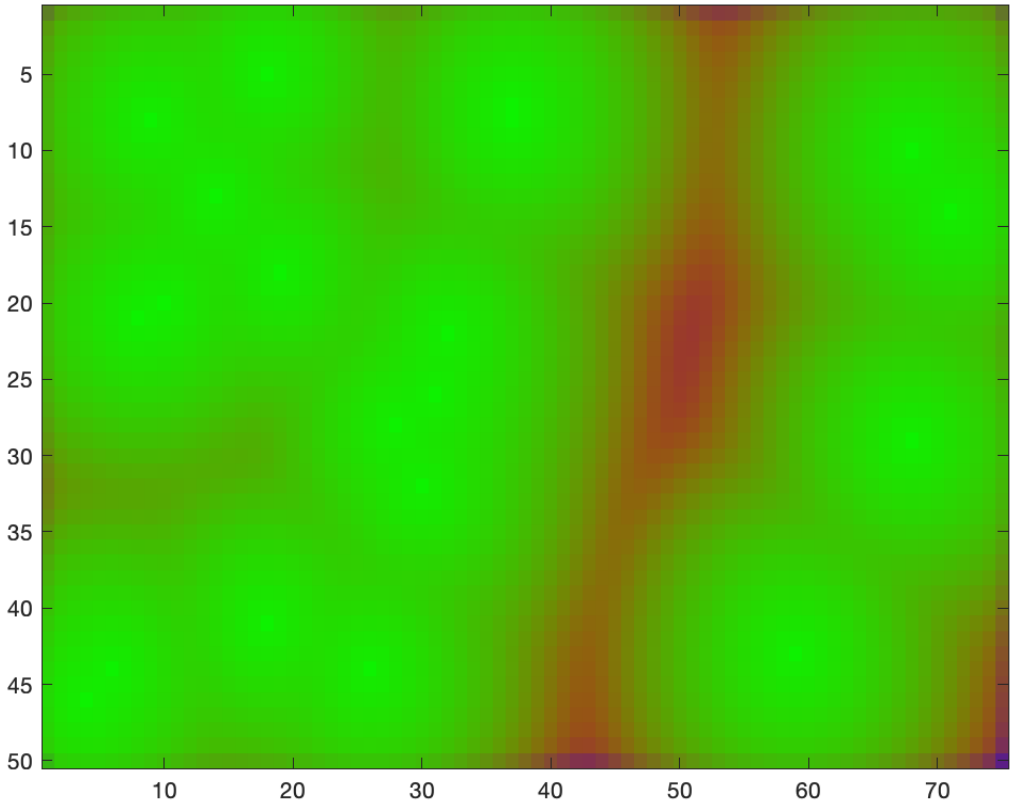

Figure 9: The 2D animation at t=300 seconds exhibits the evolution of S.I.R. states within the grid at the halfway point of the simulation. The green pixels represent the growing number of recovered individuals and the number of infections is rapidly decreasing.

Figure 9 shows the states of the grid at t=300 seconds, the halfway point of the simulation. There is now a majority of green pixels on the grid, which represent all the individuals who were once infected but have now recovered. The small red areas correspond to local populations that have recently been infected and haven't yet recovered.

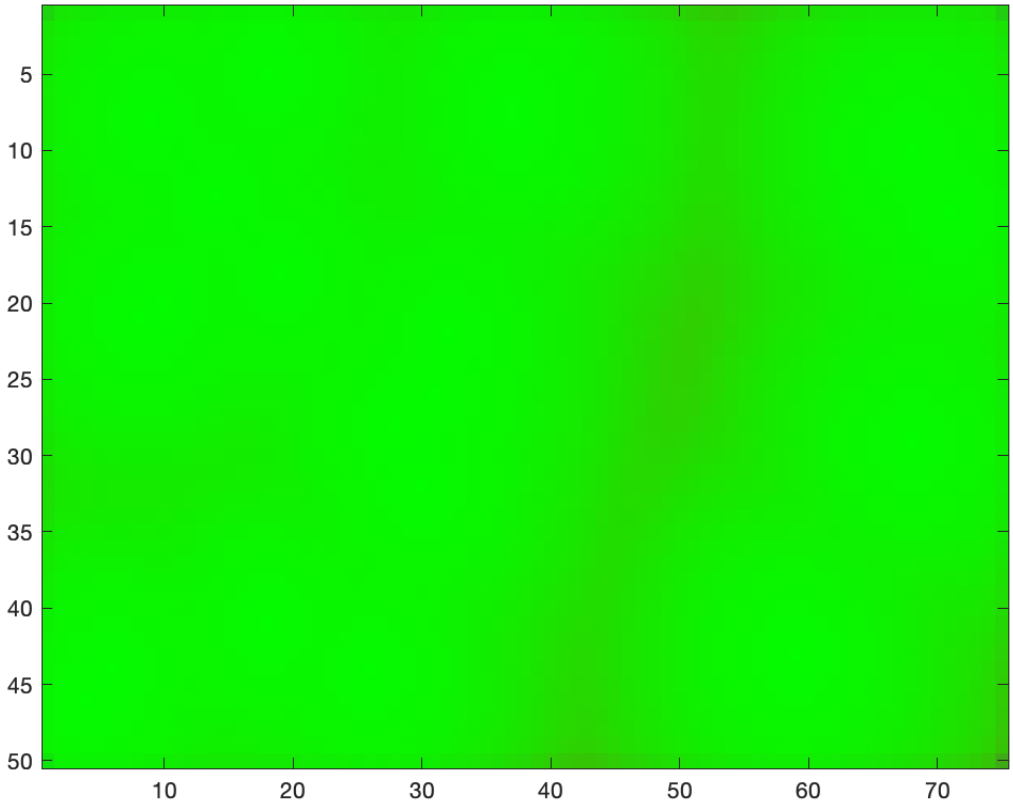

Figure 10: The 2D animation at t=450 seconds exhibits the evolution of S.I.R. states within the grid. Almost all the population has now recovered from the disease .

Figure 10 displays the states of the grid at t=450 seconds, when the simulation is 75% completed. The grid is overwhelmingly green, meaning there is a high ratio of recovered individuals for every local S.I.R. model. There is still a very faint curve of red pixels, where the infection persists but it is dying out because it has already spread through all the local populations.

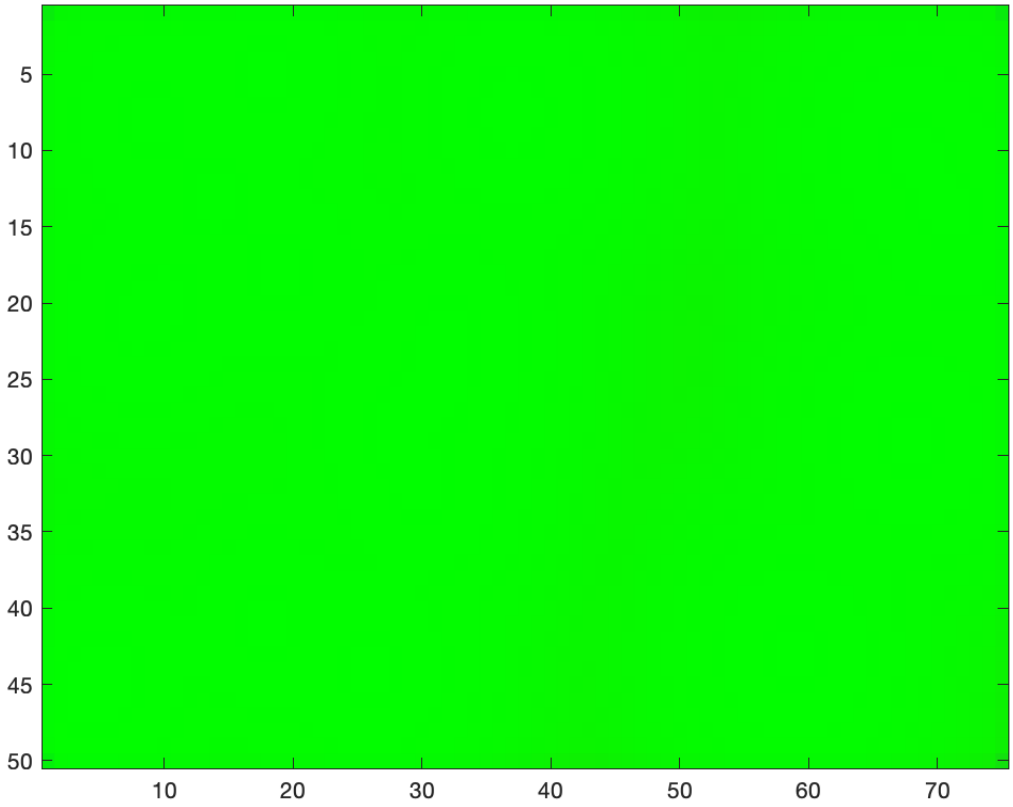

Figure 11: The 2D animation at t=600 seconds exhibits the evolution of S.I.R. states within the grid. At the end of the simulation, the grid is entirely covered with green pixels which signifies that the entire population has been infected and now recovered from the disease.

In Figure 11, we see the final states of the grid at the end of the simulation. The entire grid is green, signifying the infection has died out and there is no longer any susceptible or infected individuals in any of the local populations of the grid. Everyone has now recovered from the disease and the infection has run its course.

2.9 Discussion

This problem reveals our ability to create multiple functions that work together to solve a higher order spatial model. By implementing the Fourth-Order Runge-Kutta Algorithm into RK4, we were able to design an ODE solver that mimics the behavior of MATLAB's built-in ode45 function. The network of functions used in the main script was successful because each function manipulated and reshaped their output matrices to be compatible with the next function. This allowed us to accurately visualize the dynamics of the S.I.R. model using plots and animations, and the same network of functions can be used to simulate various other real-world systems.