CEE/MAE M20
Introduction to Computer Programming with MATLAB

Fall 2020

FINAL PROJECT

Due: Saturday, December 12, 2020, 11:59 pm

Contents

1 Background

1.1 Principal Component Analysis (PCA)

There are multiple popular dimensionality reduction techniques, such as singular value decomposition (SVD), principal component analysis (PCA), factor analysis (FA) and independent component analysis (ICA). Principal component analysis is widely used now in machine learning as unsupervised learning in the field of chemometrics and multivariate analysis.

To demonstrate the effectiveness of PCA for analyzing a dataset, we will be considering the COVID-19 data from 27 different countries, provided in the `covid_countries.csv` file. In the data, you are given six dimensional data, the difference among which can be potentially better represented by other axes, i.e. principal components. The objective of PCA is to reduce the dimensionality of the data by identifying components of the data that are most relevant. You are encouraged to learn about the theory behind PCA as it is a widely used tool in modern data analysis.

The data in the .csv file can be parsed by many functions, such as `readtable`, `readmatrix`, `csvimport`, `csvread`, `textscan`. Some of these functions also extract the column titles, which may be more useful as you can easily identify the data.

The main steps in the PCA algorithm are:

1. Normalizing data:
   Data normalization/feature scaling is a common preprocessing technique in machine learning in order to make the difference among the variables more comparable with each other. The data is represented as a 2D $n \times p$ matrix where the columns correspond to variables and rows are the observations. To center the data, for each column in the data, first compute and subtract the average of column from each element of the column and then divide each element of the column by the standard deviation of column.

2. Covariance Matrix Computation:
   The square covariance matrix $C \in \mathbb{R}^{p \times p}$ represents the pairwise correlation between the features and can be computed by the `cov` function.

3. Compute Eigenvalues/Eigenvectors:
   From the covariance matrix C, compute its eigenvalues and corresponding eigenvectors. Lookup MATLAB 's `eig` function to learn how this can be done.

4. Keeping Principal Components:
   By sorting the absolute value of the eigenvalues from greatest to smallest, we can determine the most important components of the data. The directions of these components (interpreted as vectors in p-dimensional space) are given by the corresponding eigenvectors. For our application, we will only keep the first two components so that we can visualize the data in the 2D space. However in general, we often try to find the smallest number of main components that can effectively represent the data without losing too much detail.

5. Converting Normalized Data:
   The last step is to project the normalized data onto this 2D vector subspace spanned by the two eigenvectors with the largest eigenvalues. The resulting data will then only be 2-dimensional instead of p-dimensional, and thus the goal of dimension reduction is realized.

1.2 Spatial S.I.R.

In Homework 3, you saw that the S.I.R. model can describe the spread of disease for a fixed population. The states of the model are: $S(t)$ - the number of susceptible individuals, $I(t)$ - the number of infected individuals, and $R(t)$ - the number of recovered individuals. The model satisfies $S(t) + I(t) + R(t) = P$, for some fixed constant $P > 0$, i.e., the total population does not change. The differential equations governing the dynamics of the model are given as:

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -\beta S(t)I(t), \tag{1}$$

$$\frac{\mathrm{d}I(t)}{\mathrm{d}t} = \beta S(t)I(t) - \gamma I(t), \tag{2}$$

$$\frac{\mathrm{d}R(t)}{\mathrm{d}t} = \gamma I(t), \tag{3}$$

where γ is the infectious rate and β is the contact rate. One limitation of the model is that it has no notion of spatial distribution; the model treats the entire population as a single lumped quantity. We would like to visualize how the infection spreads spatially over time, instead of only visualizing the total number of infections as a function of time. To this end, let us create a $M \times N$ grid in which each (x, y) grid tile corresponds to a local S.I.R. model.

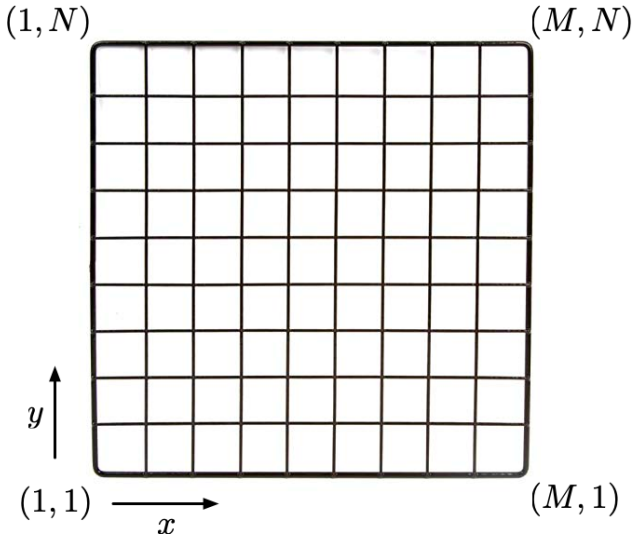

Figure 1: Each grid tile corresponds to a local S.I.R. model

For each $(x, y) \in \{1, \ldots, M\} \times \{1, \ldots, N\}$, we have a local S.I.R. model with states $S_{x,y}(t)$, $I_{x,y}(t)$, and $R_{x,y}(t)$ that satisfy $S_{x,y}(t) + I_{x,y}(t) + R_{x,y}(t) = 1$, where the population has been

normalized. Then, to model spatial influence on infection rates, we augment the dynamics as:

$$\frac{\mathrm{d}S_{x,y}(t)}{\mathrm{d}t} = -\left(\beta I_{x,y}(t) + \alpha \sum_{i,j} W(i,j) I_{x+i,y+j}(t)\right) S_{x,y}(t), \tag{4}$$

$$\frac{\mathrm{d}I_{x,y}(t)}{\mathrm{d}t} = \left(\beta I_{x,y}(t) + \alpha \sum_{i,j} W(i,j) I_{x+i,y+j}(t)\right) S_{x,y}(t) - \gamma I_{x,y}(t), \tag{5}$$

$$\frac{\mathrm{d}R_{x,y}(t)}{\mathrm{d}t} = \gamma I_{x,y}(t), \tag{6}$$

where $W(i,j)$ is a weighting function that describes the proximity of nearby neighbors and α is the spatial contact rate. In our model, we will define $W(i,j)$ according to Figures 2 and 3, where $W(i,j) = 0$ for all (i,j) outside of the specified grid.

$W(-1,1)$	$W(0,1)$	$W(1,1)$
$W(-1,0)$	$W(0,0)$	$W(1,0)$
$W(-1,-1)$	$W(0,-1)$	$W(1,-1)$

Figure 2: Locations of coordinates

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

Figure 3: Values of $W(i,j)$

For values of (x,y) that are at the boundaries of the grid in Figure 1, take the neighbors that are outside of the grid to have all states as 0.

1.3 ODE solver - Runge-Kutta methods

The S.I.R. system can be regarded as a dynamical system of the form given in equation (7). To solve the ordinary differential equation (ODE), our objective is to develop a method to numerically solve the differential equation.

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{y}) \tag{7}$$

In equation (7), $\mathbf{y} \in \mathbb{R}^n$ is a n-dimensional column vector representing the state of the system, $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is a (possibly nonlinear) function describing how the rate of change of the state relates to the current state, and t is the independent variable usually representing time.
In the forward Euler method, we use the information about the derivative of $\mathbf{y}$ at the given time step to linearly extrapolate the solution to the next step. However, the first order approximation is sometimes not accurate enough. To overcome this limitation, Runge-Kutta (RK) algorithms have been developed to take the information about the derivative at more than one time step to better extrapolate the solution.

In this section, we will introduce the second and fourth order Runge-Kutta methods, in which the local truncation error (LTE) is $\mathrm{O}(h^3)$ and $\mathrm{O}(h^5)$, respectively. [1]
To begin, consider the Taylor series expansion of $\mathbf{y}$ in the neighborhood around t_n:

$$\mathbf{y}(t_{n+1}) = \mathbf{y}(t_n) + h\frac{\mathrm{d}\mathbf{y}(t_n)}{\mathrm{d}t}\bigg|_{t_n} + \frac{h^2}{2}\frac{\mathrm{d}^2\mathbf{y}(t_n)}{\mathrm{d}t^2}\bigg|_{t_n} + \mathrm{O}(h^3), \tag{8}$$

where $t_{n+1} = t_n + h$. From the equation (7), the second order derivative of $\mathbf{f}$ is:

$$\begin{aligned}
\frac{\mathrm{d}^2\mathbf{y}(t)}{\mathrm{d}t^2} &= \frac{\partial\mathbf{f}(t,\mathbf{y})}{\partial t}, &(9)\\
&= \frac{\partial\mathbf{f}(t,\mathbf{y})}{\partial t} + \frac{\partial\mathbf{f}(t,\mathbf{y})}{\partial\mathbf{y}}\frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{d}t}, &(10)\\
&= \frac{\partial\mathbf{f}(t,\mathbf{y})}{\partial t} + \frac{\partial\mathbf{f}(t,\mathbf{y})}{\partial\mathbf{y}}\mathbf{f}(t,\mathbf{y}). &(11)
\end{aligned}$$

Note that the quantity $\frac{\partial\mathbf{f}(t,\mathbf{y})}{\partial\mathbf{y}}$ is a $n \times n$ matrix. Combining equations (8) and (11), we get:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t_n,\mathbf{y}_n) + \frac{h^2}{2}\left(\frac{\partial\mathbf{f}(t_n,\mathbf{y}_n)}{\partial t} + \frac{\partial\mathbf{f}(t_n,\mathbf{y}_n)}{\partial\mathbf{y}}\mathbf{f}(t_n,\mathbf{y}_n)\right) + \mathrm{O}(h^3), \tag{12}$$

where $\mathbf{y}_{n+1} := \mathbf{y}(t_{n+1})$ and $\mathbf{y}_n := \mathbf{y}(t_n)$. Let us introduce the variables $\mathbf{k}_1$ and $\mathbf{k}_2$:

$$\begin{aligned}
\mathbf{k}_1 &= h\mathbf{f}(t_n,\mathbf{y}_n),\\
\mathbf{k}_2 &= h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{1}{2}\mathbf{k}_1\right),\\
&= h\left(\mathbf{f}(t_n,\mathbf{y}_n) + \frac{h}{2}\frac{\partial\mathbf{f}(t_n,\mathbf{y}_n)}{\partial t} + \frac{1}{2}\frac{\partial\mathbf{f}(t_n,\mathbf{y}_n)}{\partial t}\mathbf{k}_1\right) + \mathrm{O}(h^3).
\end{aligned} \tag{13}$$

Substituting equation (13) into equations (8) and (12), we can obtain the formulas for RK2:

$$\begin{aligned}
\mathbf{k}_1 &= h\mathbf{f}(t_n,\mathbf{y}_n),\\
\mathbf{k}_2 &= h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{1}{2}\mathbf{k}_1\right),\\
t_{n+1} &= t_n + h,\\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{2}\mathbf{k}_1 + \frac{1}{2}\mathbf{k}_2.
\end{aligned} \tag{14}$$

In a similar derivation, Runge-Kutta methods for higher orders can also be developed. Here we give the formulas of the fourth order Runge-Kutta method(RK4):

[1] Local truncation error is an estimate of the error introduced in a single iteration of the method.

$$
\begin{aligned}
\mathbf{k}_1 &= h\mathbf{f}(t_n, \mathbf{y}_n), \\
\mathbf{k}_2 &= h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{1}{2}\mathbf{k}_1\right), \\
\mathbf{k}_3 &= h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{1}{2}\mathbf{k}_2\right), \\
\mathbf{k}_4 &= hf(t_n + h, \mathbf{y}_n + \mathbf{k}_3), \\
t_{n+1} &= t_n + h, \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4).
\end{aligned}
\tag{15}
$$

More information about Runge-Kutta method can be found online.[2]

1.4 Plots and animation

To visualize the spread of the infection among the population, we can create an animation in which the color of each pixel corresponds to the condition of that tile. The `image` function can be used to create this animation.

The image function takes a $M \times N \times 3$ matrix as an input, where $M \times N$ is the size of the grid. A red-green-blue (RGB) triplet is a three-element vector that specifies the intensities of the red, green, and blue components of the color of each pixel. The coordinates $(:, :, 1)$ of the 3-D array contain the red components, $(:, :, 2)$ contains the green components, and $(:, :, 3)$ contains the blue components. Each color can have an intensity of 0 to 255, but the input of image is the normalized intensity, between 0 and 1, where $[0, 0, 0]$ corresponds to black and $[1, 1, 1]$ corresponds to white.

The S.I.R. model has three outputs at each time for each grid location, so each of these outputs can be displayed as one of the RGB colors in the animation to show how the percentage of each group is changing in the space. This image can be plotted for each time step to get an animation showing the spread of the disease.

[2] https://en.wikipedia.org/wiki/Runge-Kutta_methods

2 Problem Statement

2.1 Main Component Analysis with PCA

2.1.1 Implementing the PCA Algorithm

Create a function in MATLAB to realize principal component analysis (PCA). The interface of your function should look as follows:

```
function [coeffOrth,pcaData] = myPCA(data)
% myPCA analyzes the principal components of given COVID-19 statistical ...
    data from multiple countries - covid_countries.csv
%   Inputs:
%         data: A nxp matrix representing only the numerical parts of ...
    the dataset
%   Outputs:
%         coeffOrth: a pxp matrix whose columns are the eigenvectors ...
    corresponding to the sorted eigenvalues
%         pcaData: a nxp matrix representing the data projected onto ...
    the principal components
```

2.1.2 Problem 1 Main Script

Create your main script titled `project_UID_p1.m`, replacing `UID` with your numerical UID. In this main script, the general structure of your script should follow the steps:

1. Load the provided `covid_countries.csv`.
2. Call your `myPCA` function with the loaded data.
3. Use MATLAB 's `biplot` function to visualize the first two principle component eigenvectors and the projected data onto this 2D vector subspace and embed the plot to your report.
4. Provide explanations for what the axes of the plot represent and describe how the data can now be interpreted.

2.2 Solving the Spatial S.I.R. Model

2.2.1 Implementing the Fourth-Order Runge-Kutta Algorithm

Create a function in MATLAB that can solve differential equations using the RK4 method. Fix the time step size $h = 0.1$. Since the step size is fixed, preallocate matrices of appropriate sizes for your function's outputs. To ensure that your implementation is compatible, follow the interface given below:

```
function [t, y] = RK4(f, tspan, y0)
% RK4 Numerically solves the differential equation using the ...
    fourth-order Runge-Kutta algorithm
%   Inputs:
%       f: function handle of f(t, y)
%       tspan: the time period for simulation (should be a 1x2 array ...
    contain start time and end time)
%       y0: the initial conditions for the differential equation
%   Outputs:
%       t: corresponding time sequence as a T x 1 vector
%       y: the solution of the differential equation as a T x n matrix, ...
    where T is the number of time steps and n is the dimension of y
```

The pseudocode of RK4 is given in Algorithm 1. Be sure that the dimensions of your output are correct.

Algorithm 1 Fourth order Runge-Kutta Method

$\mathbf{f} \leftarrow \mathbf{f}(t, \mathbf{y})$
$t_0, \mathbf{y}_0 \leftarrow$ *initial conditions*
$h \leftarrow$ *time step size*
$nSteps \leftarrow$ *number of steps*
$k \leftarrow 0$
store t_0 *and* $\mathbf{y}_0$ *in* t *and* $\mathbf{y}$*, respectively*
update t_k *and* $\mathbf{y}_k$ *with* t_0 *and* $\mathbf{y}_0$
while $k \leq nSteps$ **do**
  $\mathbf{k}_1 \leftarrow h\mathbf{f}(t_k, \mathbf{y}_k)$
  $\mathbf{k}_2 \leftarrow h\mathbf{f}(t_k + h/2, \mathbf{y}_k + \mathbf{k}_1/2)$
  $\mathbf{k}_3 \leftarrow h\mathbf{f}(t_k + h/2, \mathbf{y}_k + \mathbf{k}_2/2)$
  $\mathbf{k}_4 \leftarrow h\mathbf{f}(t_k + h, \mathbf{y}_k + \mathbf{k}_3)$
  $t_k \leftarrow t_k + h$
  $\mathbf{y}_k \leftarrow \mathbf{y}_k + (\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)/6$
  store t_k *and* y_k *in* t *and* $\mathbf{y}$ *respectively*
  $k \leftarrow k + 1$
end while

2.2.2 Implementing the Dynamics Model

In order to simulate the system using your implementation of the Runge-Kutta algorithm or `ode45`, the states that represent your system must be vectorized. This means that matrices of dimension 2 or higher must be expressed as a single dimension. For example, if `A` is a 2×2 matrix, then the shorthand for vectorizing in MATLAB is `A(:)`. You may notice that we normally label matrices in row-major format, but MATLAB vectorized matrices in column-major format. To convert the vector back into the corresponding matrix, look up the function `reshape` and figure out what arguments should be passed. Describe your findings in the report.

$$\texttt{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \texttt{A(:)} = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 4 \end{bmatrix}. \tag{16}$$

To implement the S.I.R. dynamics given in equations (4)-(6), create a function called `dynamicsSIR` that has the following interface:

```
function dxdt = dynamicsSIR(x, M, N, alpha, beta, gamma)
% dynamicsSIR Compute the rate of change of the model
%   Inputs:
%       x: vectorized state
%       M, N: size of the grid
%       alpha, beta, gamma: model parameters
%   Output:
%       dxdt: vectorized time derivative of state
```

Here, `x` is a one-dimensional column vector of length $3MN$, and `alpha, beta, gamma` are the model parameters. Your output `dxdt` should also be a one-dimensional column vector of length $3MN$. Carefully ensure that the order of your variables in `x` and `dxdt` are consistent.

2.2.3 Solving the Dynamics Model

To facilitate easy comparison of your ODE solver with MATLAB 's `ode45`, create another function that simulates your spatial S.I.R. model using an ODE solver function handle passed in as an argument. Specifically, your function should be called `solveSpatialSIR` and have the following interface:

```
function [t, x] = solveSpatialSIR(tFinal, initialCondition, alpha, ...
    beta, gamma, odeSolver)
%solveSpatialSIR Solve the spatial SIR model
%   Inputs:
%       tFinal:  end time for the simulation (assuming start is t=0)
%       initialCondition: a MxNx3 matrix that sums to 1 in third dimension
%       alpha, beta, gamma: model parameters
%       odeSolver: a function handle for an ode45-compatible solver
%   Outputs:
%       t: a vector of the time-steps
%       x: MxNx3xlength(t) matrix representing the state vs. time
```

The `initialCondition` argument is a $M \times N \times 3$ matrix in which each local S.I.R. model has its states summing to 1: $S_{x,y}(0) + I_{x,y}(0) + R_{x,y}(0) = 1$. You can infer M and N by the size of this matrix. The output of your function should be a $M \times N \times 3 \times T$ matrix, where T is the number of time steps generated by the `odeSolver`. Note that the output of the `odeSolver` may not have the variables in the same format as `x`, so you may have the reorder the dimensions. Useful functions that may help you are: `reshape`, `squeeze`, `permute`.

2.2.4 Time Series Plotting

In our spatial S.I.R. system, $S_{x,y}(t)$ is the ratio of susceptible individuals, $I_{x,y}(t)$ is the ratio of infected individuals, and $R_{x,y}(t)$ is the ratio of recovered individuals. We wish to visualize a particular local S.I.R. model by specifying its spatial (x, y) coordinates. To do that, create a function to generate time series plotting for a local S.I.R. distribution.

Create a function called `plotTimeSeries` that generates a plot with three subplots to show $S_{x,y}(t)$, $I_{x,y}(t)$, and $R_{x,y}(t)$ for a specified (x, y). Use the function `subplot` to place the three plots on the same figure. More information about the usage of `subplot` can be obtained from https://www.mathworks.com/help/matlab/ref/subplot.html. Your function should programmatically save the generated figure as `time_series_x_y.png` file, where `x` and `y` are the inputs to the function. Embed the generated figures into your report. Be sure to label all of your axes and give meaningful titles to your subplots. The (x, y) location information should also be programmatically included in the title string. Hint: `sprintf` can be used to format a string.

Your `plotTimeSeries` function should have the following interface:

```
function plotTimeSeries(t, X, x, y)
% plotTimeSeries: a function that plots and saves the local S.I.R ...
    distribution at spatial coordianate (x, y).
%   Inputs:
%       t: a vector of time steps
%       X: an M*N*3*length(t) matrix, where each point in the M*N space
%       corresponds to a local S.I.R. model with states whose values ...
    are between 0 and 1. This 3D matrix is repeated for each time step, ...
    making it a 4D matrix.
%       x: the spatial x-coordinate on the grid
%       y: the spatial y-coordinate on the grid
%   Outputs:
%       This function has no outputs
```

2.2.5 2D Animation

In this problem, we will use red to show the infected, green for recovered, and blue for susceptible individuals. The pixels will start blue, as the majority of the population in each pixel starts susceptible, then some grids will turn red, showing that the majority of the population have been infected, and then change to green as the population recovers. A sample animation .gif file has been included for your reference.

Show how the disease spreads through the population with time. Write a function that takes the $M \times N \times 3 \times \text{length}(t)$ matrix X that has the result of the S.I.R. simulation as an input, and plots an image with a color representing the percent of susceptible, infected, and recovered individuals of each grid in each pixel. Display the image at every 10^{th} time step with a pause of $0.1s$ before changing to the next image. Since $t_{\text{Final}} = 60$ and $h = 0.1$, for this problem, your animation will have a total of 60 frames. You can use commands such as `image`, or `pause` to achieve this goal.

Note that the input of the image function is a 3D matrix while the output of `solveSpatialSIR` function is a 4D matrix.

Create a function called `animate` that accepts a matrix of size $M \times N \times 3 \times T$, where the first two dimensions determine the (x, y) position in the grid, the third dimension specifies 3-dimensional local S.I.R. state, and T specifies the number of time-steps. Your function should have the following interface:

```
function animate(X)
% animate: a function that shows the change in the ratio of susceptible,
% infected, and recovered individuals in the grid as an image.
%   Inputs:
%       X: an M*N*3*length(t) matrix, where each point in the M*N space
%       corresponds to a single grid with 3 numbers between 0 and 1 showing
%       the SIR result. this 3D matrix is repeated for each time step
%       making it a 4D matrix.
%
%   Output: this function does not output anything.
```

2.2.6 Problem 2 Main Script

Create your main script titled `project_UID_p2.m`, replacing `UID` with your numerical UID. In this main script, you will be solving for a spatial S.I.R. system on a grid of size 50×75 using your `solveSpatialSIR` function. You will also be benchmarking the performance of solving with your `RK4` implementation, as well as MATLAB 's `ode45` solver. Finally, your script should visualize the data by executing your `plotTimeSeries` and `animate` functions.
The general structure of your script should follow the steps:

1. Load the provided `InitialValues.mat` file, an $M \times N \times 3$ array including the initial S, I and R values in the grid, and set the following parameters:

$$\alpha = 0.1, \tag{17}$$
$$\beta = 0.05, \tag{18}$$
$$\gamma = 0.1, \tag{19}$$
$$t_{\text{final}} = 60. \tag{20}$$

2. Solve the spatial S.I.R. system by calling `solveSpatialSIR.m` twice; once by passing your `RK4.m` solver as an input, and once by passing MATLAB 's `ode45.m`. Be sure to use `tic` and `toc` to benchmark the runtime of both solvers, and print the runtime results. These values must be included in the report. Comment on any noticeable difference between the performance in the report. Suggest possible explanations for why one of the ODE solvers may be faster than the other.

3. Call your `plotTimeSeries` function on the system solved with your `RK4.m` solver at the following grid coordinates: $(1, 1), (5, 18), (30, 70)$.

4. Call your `animate` function on the system solved with your `RK4.m` solver to generate the 2D visual animation.

Given the large size of this problem, your script may take several seconds to minutes to run. During the debugging phase of your development, you may consider testing on a smaller grid and shorter t_{Final}.

3 Project Deliverables

Using the naming convention presented in the syllabus, submit **two** separate files to the CCLE course website: (1) your .pdf report named as `project_UID.pdf`, where `UID` is your university ID number, and (2) a .zip folder, named `project_UID.zip`, that contains all your code files.

3.1 Code (70%)

You should submit the following files zipped together as `project_UID.zip`. If you used helper functions or scripts, you should include those in your submission as well. Every file must include a few comment lines (your name, UID, filename, description) at the beginning. The description in this top comment section should be a few sentences on the inputs, outputs, and methods. The entire code should be properly **commented**.

- `myPCA.m`
- `project_UID_p1.m`
- `RK4.m`
- `dynamicsSIR.m`
- `solveSpatialSIR.m`
- `plotTimeSeries.m`
- `animate.m`
- `project_UID_p2.m`

The following deductions apply to all the scripts in your submission:

- Insufficient commenting (-25% per script)
- Failure to follow required function interfaces (-25% per script)
- Very inefficient implementation (-20% per script)
- Logic errors (-15% per instance)
- Unused variables (-5% per instance)
- Matrix variables changing size on each loop iteration / no preallocation (-5% per instance)
- Plots missing labels, titles, and/or legends (-5% per instance)
- Typos or lack of proof-reading (-5% per instance)
- Missing few comment lines with your name, UID, filename, and description at the beginning of each file (-10% per script)

3.2 Report (30%)

Your report should be named `project_UID.pdf` and document all the important steps of your code, similar to the reports for the homework. Your report should be formatted as a journal article and divided into multiple sections and subsections (and, if necessary, a reference section). There is no page limit. It should read like a story with coherently organized sections, figures, and plots. Any code that requires a detailed explanation should also be included in your report. Be mindful of the following common pitfalls.

- Your report should have multiple plots, images, and schematics. Make sure to number these graphic elements (e.g. Figure 1, Figure 2, ...). Include a caption for each of them. These elements must be referenced and described in the text of the report.
- Your report must be typed. Do not use images of handwritten equations or pictures of plots taken with a camera.
- Do not take screenshots of plots from MATLAB. Save the plot using `saveas()` as an image file in pdf/png/jpg/other format and then insert it into the report.
- Do not take screenshots of code snippets. Type it out in the report. Every code snippet (<5 lines) must be described in the report.
- Try to use pseudocodes, flowcharts, or schematics instead of MATLAB code in the report.
- Do not take screenshots of this document. If you are copying any part of this document to use in your report, please cite this document in your report. Any recognized citation style is acceptable.

You are also encouraged to take this opportunity to learn LaTeX, given the abundance of scientific notations. You are also highly encouraged to use `git` as version control for your files, and also `GitHub`[3] as a place to store your code and serve as a portfolio of your projects.

[3] GitHub available at https://github.com/