Problem Set II

Exploring Geodesic Shooting for Diffeomorphic Image Registration

Kunsh Singh

10/9/22

Problem 1

For $dy/dt + 2y = 2 - e^{(} - 4t), y(0) = 1$, do the following:

(a) Derive its closed-form solution on your own.

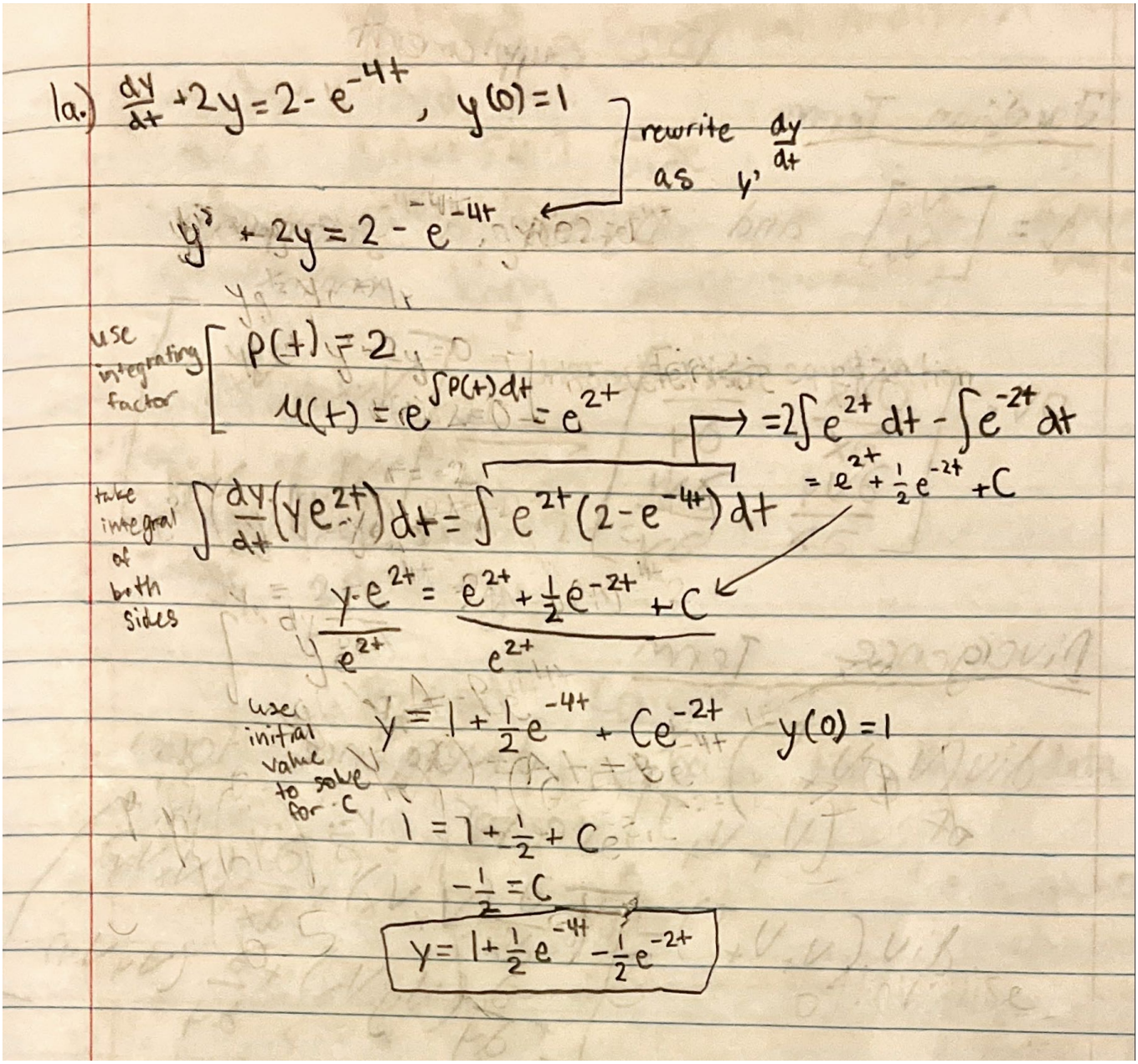


Figure 1: Derived Closed-Form Eulers

The following parts can be combined since we are looking at different values of t in part b and plotting them on the same figure. However, we need each of the step sizes from part c in order to do so.

(b) Use Euler's Method to find the approximation to the solution at $t = \{1, 2, 3, 4, 5\}$*, and compare to the exact solution in (a) by plotting them on a same figure.*
(c) Use different step size $h = \{0.1, 0.05, 0.01, 0.005, 0.001\}$ *and plot out your approximated function value.*

To approach this problem, we must consider Euler's equation $y_t = y_{t+1} + \frac{dy}{dt} \cdot \Delta t$. Euler's is quite useful for complicated differential equations since it can approximate with high precision. We don't need many packages to perform this computation, except for math which will be useful for taking e^n, where n is a computed value. We also use numpy and matplotlib.pyplot in order to display our graph.

The following are our imports.

```
import math
import matplotlib.pyplot as plt
import numpy as np
```

Next, let's define the derivative of our function, which we will use in the future when we compute Eulers. We can rearrange our initial equation $\frac{dy}{dt} + 2y = 2 - e^{-4t}$ to $\frac{dy}{dt} = 2 - e^{-4t} - 2y$ in order to isolate for $\frac{dy}{dt}$.

```
def func(self,x,y):
    return 2 - math.exp((-4 * x)) - 2 * y
```

Let's define our initial parameters as well. We are given our set of $t = \{1, 2, 3, 4, 5\}$ we must approximate our $y(t)$ for, and a step size of $h = \Delta t = \{0.1, 0.05, 0.01, 0.005, 0.001\}$, $y(0) = 1$.

```
def main(self):
    y0 = 1
    t = [1,2,3,4,5] #Given set
    max_step = t[-1] #Last in t
    step_size = [0.1,0.05,0.01,0.005,0.001] #Delta t = given h

    for change in step_size:
        self.eulers(y0, max_step, change)
```

For each of our step sizes Δt, we want to perform Euler's method to find our set of t's. We will use a continuous graph for our $y(t)$ passing through $[0, 5]$ as well as a continuous graph of our solution to better visualize the accuracy of Euler's method using a specific step size.

Now that we've gotten our initial parameters taken care of, we can finally perform Euler's method. Our variable inv is the reciprocal of the step size.

```
for i in range(0,int((maxStep/step)+1),int(step*inv)):
                    #[0,maxStep] inclusive
    if ((i/inv) in t):
        y_t.append(y_approx)
    yVals = np.append(yVals,y_approx)
    y_approx += self.func((i/inv),y_approx)*(1/inv)
    xVals = np.append(xVals,(i/inv))
```

We iterate t from 0 to 5 such that $h = \Delta t = \{0.1, 0.05, 0.01, 0.005, 0.001\}$, for the equation $y_t = y_{t+1} + \frac{dy}{dt} \cdot \Delta t$, stored in our y_approx. We store each (x_n, y_n) coordinate pair using our xVals and yVals.

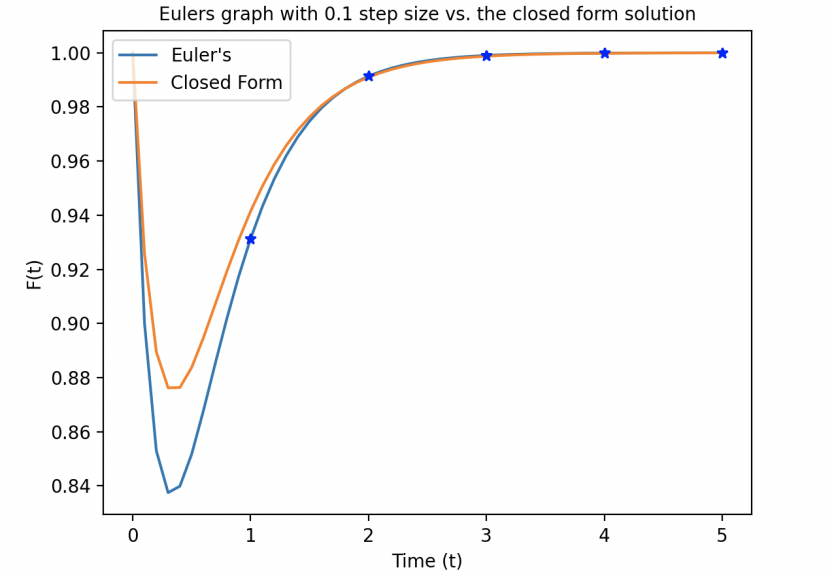
We must plot our Euler's method solution against our close form solution. The following method "closed" is used to retrieve our closed solution value at a point x — we can use this in order to gather a continuous graph for our closed form solution.

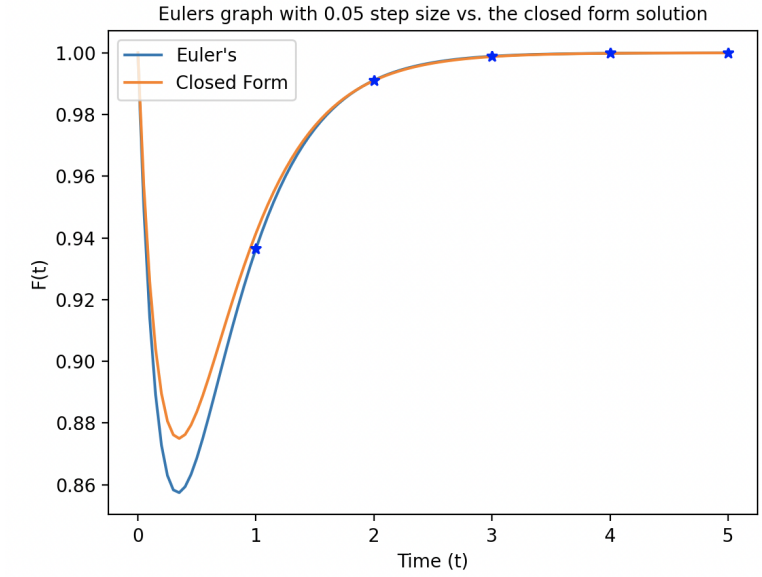
```
def closed(self,x):
    return 1 + (1/2) * math.exp((-4 * x)) - (1/2) * math.exp((-2 * x))

yVals2 = np.array([])
    for x in xVals:
        yVals2 = np.append(yVals2,self.closed(x))
```

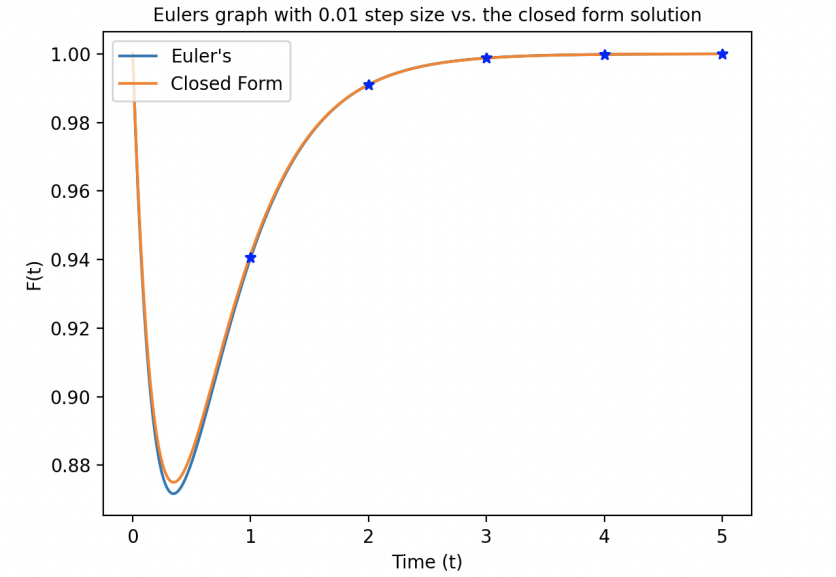
Finally, we can plot our closed form solution against our Euler's method solution to our initial differential equation $\frac{dy}{dt} + 2y = 2 - e^{-4t}$.

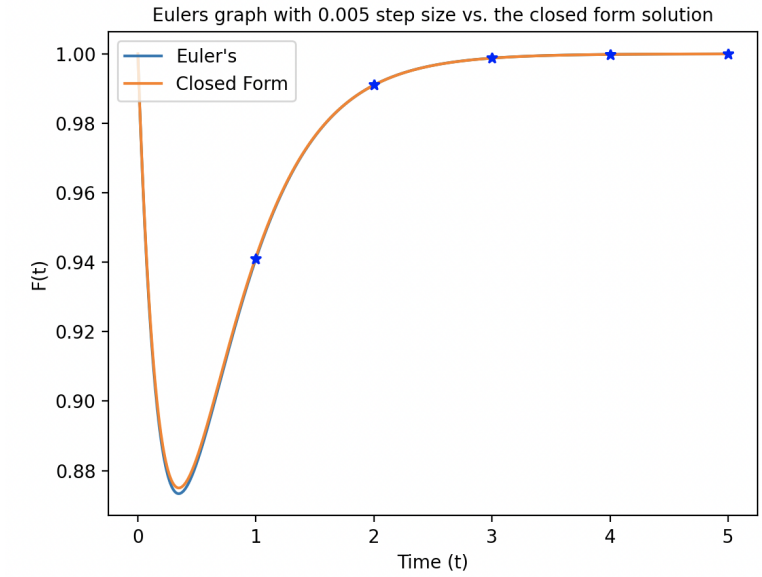
```
plt.plot(xVals, yVals, label="Euler's")
plt.plot(xVals, yVals2, label="Closed Form")
plt.plot(t, y_t, 'b*')
plt.xlabel("Time (t)")
plt.ylabel("F(t)")
title = "Eulers with " + str(step) + " step size vs. closed form solution"
plt.title(title, fontsize=10)
plt.legend(loc="upper left")
plt.show()
```

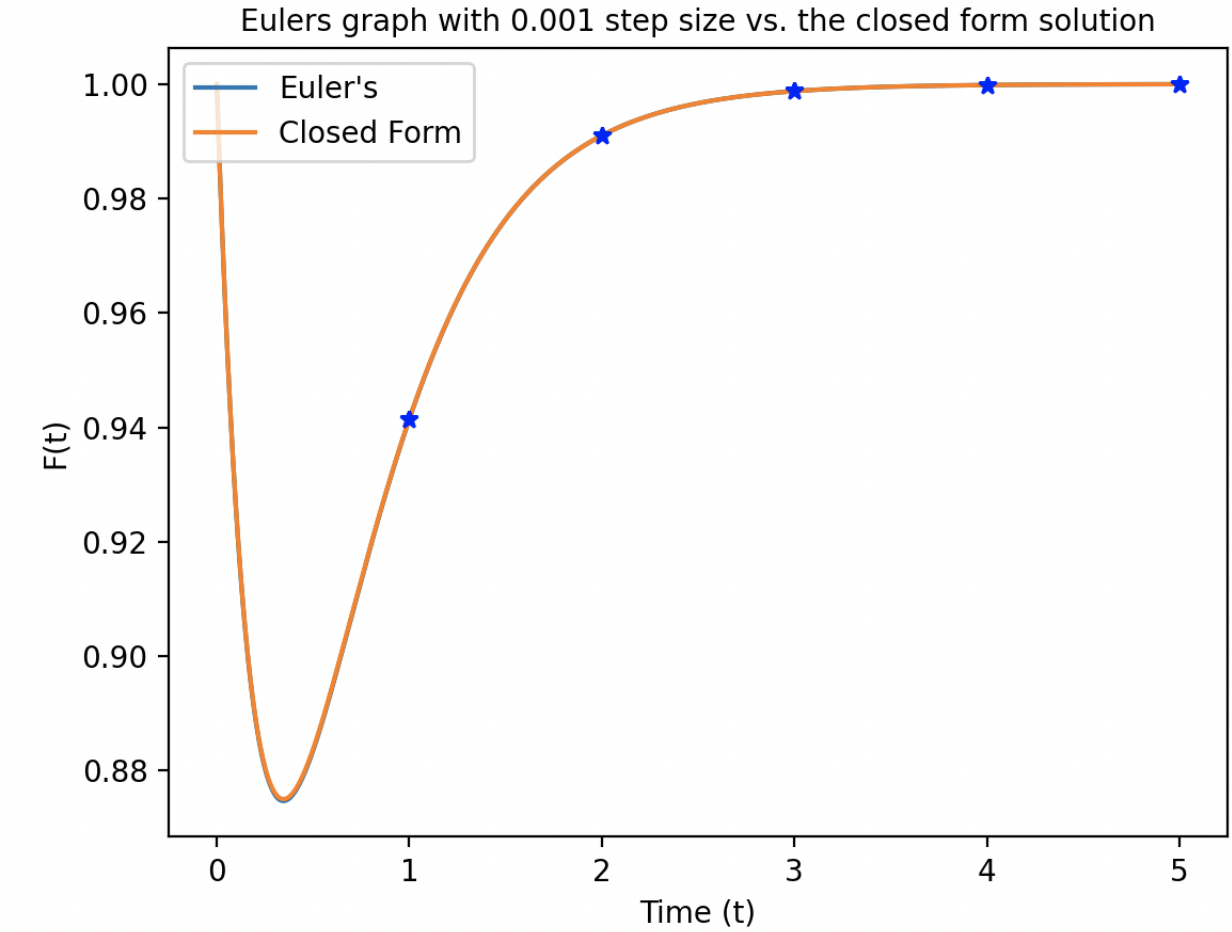
Our results for each step size can be found below:











Problem 2

Write your own program of PCA (no packages allowed) and test your implementation on MNIST hand-written digits database (provided in "mnist.mat") .

We move on to Principal Component Analysis (PCA). Given an assortment of images, we can determine certain attributes such as eigenvalues and eigenvectors. PCA can be used to retain variance while limiting the number of features and dimensions. Thus, PCA is highly useful for visualizing and analyzing highly dimensional data, and can be widely applicable to analyzing data sets when it comes to face recognition, image compression, and image denoising.

Though we cannot use any libraries to immideately compute the PCA, we will still use basic libraries in order to perform matrix operations and plot various functions with ease. The following are our imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
```

Lets proceed with getting our input data from "mnist.mat." We will call this matrix 'x'.

```
# Load data from mnist.mat
data = loadmat('MINIST_Q2/mnist.mat')
input = data['trainX']
```

Next, we will write our method to determine our eigenvalues and eigenvectors using singular-value decomposition. We first must normalize our data by subtracting by our mean and dividing by our standard deviation (essentially finding z-scores). From there, we determine our covariance matrix by using the formula $(x^T \cdot x)/columns$. Finally, we can use np.linalg.svd on our covariance matrix to determine eigenvalues and eigenvectors.

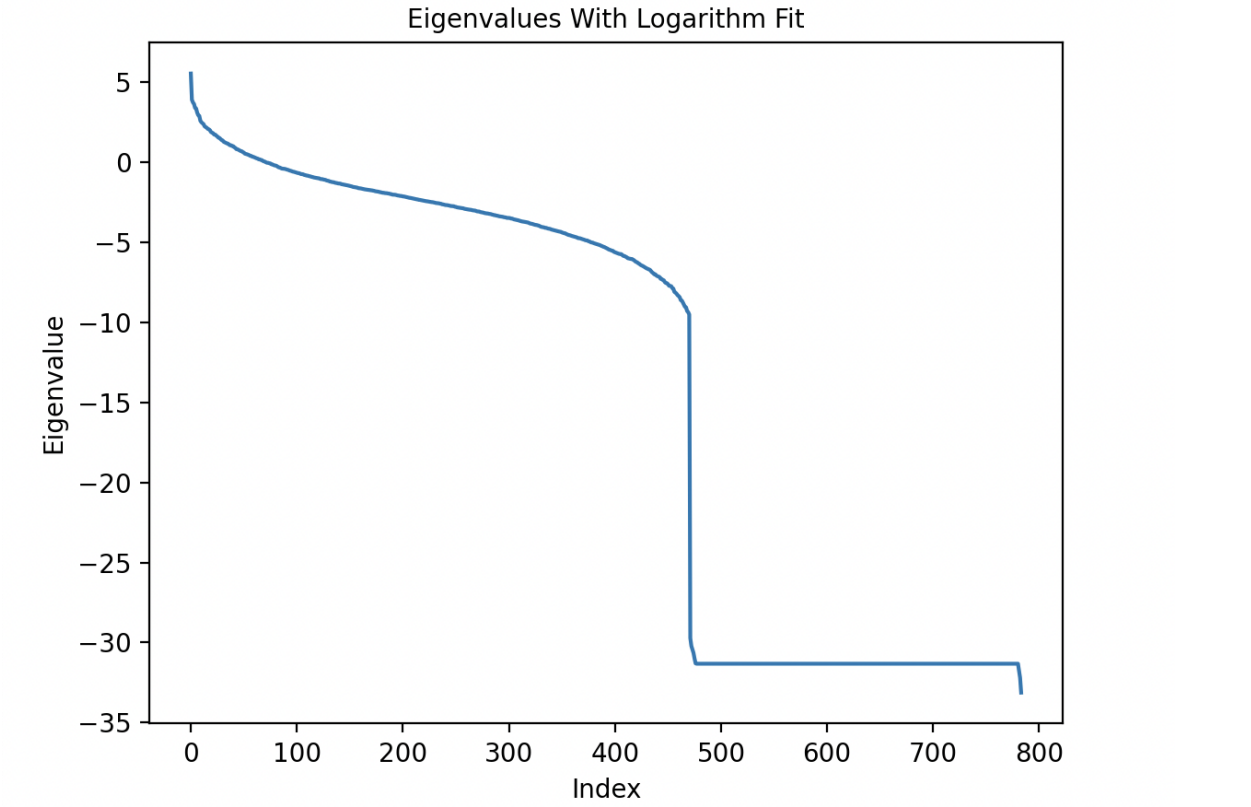
```
def pca(self,x):
    # Normalize the input matrix
    x = (x - x.mean()) / x.std()

    # Determine the covariance matrix
    x = np.matrix(x)
    cov = (np.transpose(x) * x) / x.shape[1]

    # Perform Singular-Value Decomposition
    U, S, V = np.linalg.svd(cov)

    return U, S

# Get the principal components
    eigenvecs, eigenvals = self.pca(input)
```

We can use our following eigenvalues and eigenvectors in order to solve the following parts with ease.

a) Plot all eigenvalues.

Matplotlib.pyplot can help us display our eigenvalues from our PCA.

```
# Plot the eigenvalues
eigen_logs = np.log(eigenvals)
plt.plot(eigen_logs)
plt.xlabel('Index')
plt.ylabel('Eigenvalue')
plt.title("Eigenvalues With Logarithm Fit", fontsize=10)
plt.show()
```



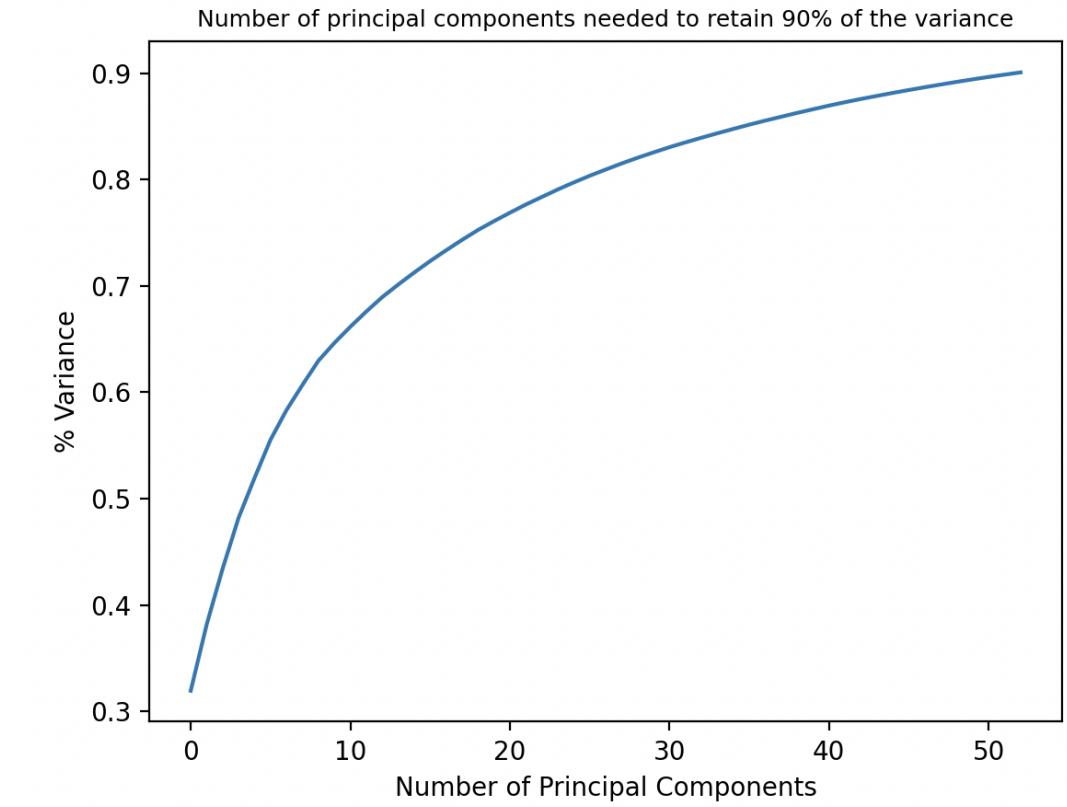
b) How many numbers of principal components would you choose to achieve at least 90% of the data variance? Include a plot of (number of principal components) vs. (accumulated data variance).

To find the principal components that retain 90% of the data variance, we must keep track of a summation of variance against the total variance as we append each percent variance to a list.

```
while (float(current_variance / total_variance) < 0.9):
    current_variance += eigenvals[k]
    p_variance.append(float(current_variance/total_variance))
    k += 1
```

And finally we plot our number of principal components against our % variance.

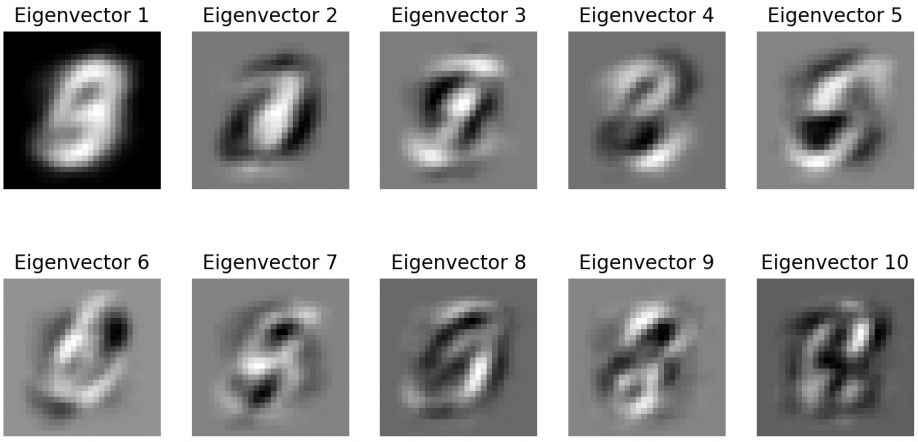
```
plt.plot(p_variance)
plt.xlabel('Number of Principal Components')
plt.ylabel('% Variance')
plt.title("Num. of p components to retain 90% of variance", fontsize=9)
plt.show()

print("Num of p components to retain 90% of variance: ", len(p_variance))
```



(c) Plot the first 10 eigenvectors (same dimension as digit images).

Since np.linalg.svd already sorts our eigenvectors by descending order, we do not need to re-sort them. The following code isolates the first 10 eigenvectors and reshapes each eigenvector from a 1x784 column to a 28x28 image.

```
i10eigenvecs = eigenvecs[:, :10]
fig, ax = plt.subplots(2, 5, figsize=(10, 5))

# Plot the first 10 eigenvectors
for i in range(2):
    for j in range(5):
        ax[i][j].imshow(i10eigenvecs[:,i*5+j].reshape(28, 28),cmap="gray")
        title = "Eigenvector " + str(5*i + j)
        ax[i][j].set_title(title)
        ax[i][j].axis('off')
plt.show()
```



Problem 3

Geodesic shooting for diffeomorphic image registration. *Implement geodesic shooting by the following strategy and compute the final transformation ϕ_1 at time point $t = 1$. Deform a given source image (included in the data folder) by using the transformation ϕ_1.*

We move on to implementing geodesic shooting for diffeomorphic image registration. We are tasked with calculating a transformation ϕ_1 for $t = 1$ in order to deform a source image. Geodesic shooting has two main equations:

$$\frac{dv_t}{dt} = -K[(Dv_t)^T \cdot v_t + div(v_t v_t^T)]$$

$$\frac{d\phi_t}{dt} = v_t \circ \phi_t$$

Such that $\frac{dv_t}{dt}$ is the velocity at each pixel and $\frac{d\phi_t}{dt}$ is a transformation that will deform our source image. In order to compute both differential equations, we will be using Euler's.

Let's start with our package imports. In addition to the standard numpy, opencv, pyplot, and math, we are also using sitk to help us input our data, scipy.ndimage.map_coordinates to help us interpolate.

```
import math
import SimpleITK as sitk
import numpy as np
import scipy as sp
from scipy.ndimage import map_coordinates as mc
import matplotlib.pyplot as plt
import cv2
from skimage.color import rgb2gray
```

We are given the following functions to help us input our data as well.

```
def velocity(self): #Gets velocity based on input image
    return sitk.GetArrayFromImage(sitk.ReadImage(self.PATHOFVELOCITY))

def src(self): #Gets src based on input image
    return sitk.GetArrayFromImage(sitk.ReadImage(self.PATHOFSOURCE))
```

And we see the return of the following functions from PS1:

```
def forward_difference_x(self,image):#Computes x component of the gradient
    rows, cols = image.shape
    d = np.zeros((rows,cols))
    d[:,1:cols-1] = image[:,1:cols-1] - image[:,0:cols-2]
    d[:,0] = image[:,0] - image[:,cols-1]
    return d

def forward_difference_y(self,image):#Computes y component of the gradient
    rows, cols = image.shape
    d = np.zeros((rows,cols))
    d[1:rows-1, :] = image[1:rows-1, :] - image[0:rows-2, :]
    d[0,:] = image[0,:] - image[rows-1,:]
    return d
def dx(self, x): #Determines fowardX at each pixel, shortens name
    return self.forward_difference_x(x)

def dy(self, y): #Determines fowardY at each pixel, shortens name
    return self.forward_difference_y(y)
```

However, we see a slight change in our divergence functions from PS1.

```
def divx(self, vx,vy, epsilon=0.000001): #Detemines the divergence of vx
    return self.dx(vx*vx+epsilon)+self.dx(vx*vy+epsilon)
def divy(self, vx,vy, epsilon=0.000001): #Detemines the divergence of vy
    return self.dy(vy*vx+epsilon)+self.dy(vy*vy+epsilon)
```

Last but not least, we have our code for performing the smoothing Kernel K from our PS1 adjusted to exclude frequencies beyond 16^2 from the Fourier domain.

```
def smoothing(self, u): #Smooths image (from pset1)
    #Perform's fourier transform on input image
    ftShift = np.fft.fftshift(np.fft.fft2(u, axes=(0,1)))

    mask16 = np.zeros_like(u)
    x = int(mask16.shape[1]/2) #CenterX of frquency domain
    y = int(mask16.shape[0]/2) #CenterY of frquency domain

    #Mask fourier transform to exclude high frequencies beyond 16^2
    cv2.rectangle(mask16,((x-16),(y-16)),((x+16),(y+16)),(255,255,255),-1)
    ftShift *= mask16[0]/255 #Apply mask

    #Reconstruct our fourier transform from frequency domain
    u = np.abs(np.fft.ifft2(np.fft.ifftshift(ftShift), axes=(0,1)))

    return u
```

To begin geodesic shooting, we must first initialize our constants. We are given a 1x100x100x3 which from we can extract v_x and v_y, both of which are 100x100 matrices. We input 3 source images: 1 given, 1 lena, and 1 coordinate plane. We use multiple source images in order to better understand the deformation caused by the transformation *phi* in our results. Finally, we initialize ϕ_x and ϕ_y based on a 100x100x2 meshgrid we get from np.mgrid.

```
STEPS = 2

src = np.squeeze((self.src())) #Get source image
src2 = plt.imread('lena.png') #Testing lena too!
src3 = plt.imread('graph.jpeg') #And a standard cartesian graph

src2 = rgb2gray(src2)
src3 = rgb2gray(src3)
src2 = cv2.resize(src2, (0,0), fx = 0.19531, fy = 0.19531) #Downsize Lena
src3 = cv2.resize(src3, (0,0), fx = 0.3125, fy = 0.313479) #Downsize graph

input = self.velocity() #Get input velocities
input = np.squeeze(input)
```

```
vx =  input[:,:,0] #Splice input matrix to get the x component of velocity
vy =  input[:,:,1] #Splice input matrix to get the y component of velocity

phi_x, phi_y  = np.mgrid[0:100, 0:100].astype(float) #Mesh grid
```

Our Geodesic Shooting algorithm is comprised of three steps: calculating the Jacobian of v_t, determining the new v_t based on Euler's method, and then finally determining the new ϕ_t based on Euler's method.

Rather than directly computing the Jacobian, we separate the computation of $Dv_t \cdot v_t$ into the equations $v_{xx} \cdot v_x + v_{xy} \cdot v_y$ and $v_{yx} \cdot v_x + v_{yy} \cdot v_y$ in order to avoid computations of greater than 2 dimensions.

```
#Find Jacobian components of velocity matrix
vxx = self.dx(vx)
vxy = self.dy(vx)
vyx = self.dx(vy)
vyy = self.dy(vy)

#Transpose Jacobian components
jacobian_transpose = np.transpose([[vxx, vxy],[vyx, vyy]])
vxx = jacobian_transpose[:,:,0,0]
vxy = jacobian_transpose[:,:,0,1]
vyx = jacobian_transpose[:,:,1,0]
vyy = jacobian_transpose[:,:,1,1]

#Compute Jacobian term multiplied with velocity matrix
jacobian_x = vxx*vx + vxy*vy
jacobian_y = vyx*vx + vyy*vy
```

Next, we perform Euler's on our v_t. We use an intermediary step to compute each $\frac{dv_x}{dt}$ and $\frac{dv_y}{dt}$ in order to simplify determining v_t. We keep terms split into x and y components in order to simplify computations and once again avoid any computations with matrices greater than order 2.

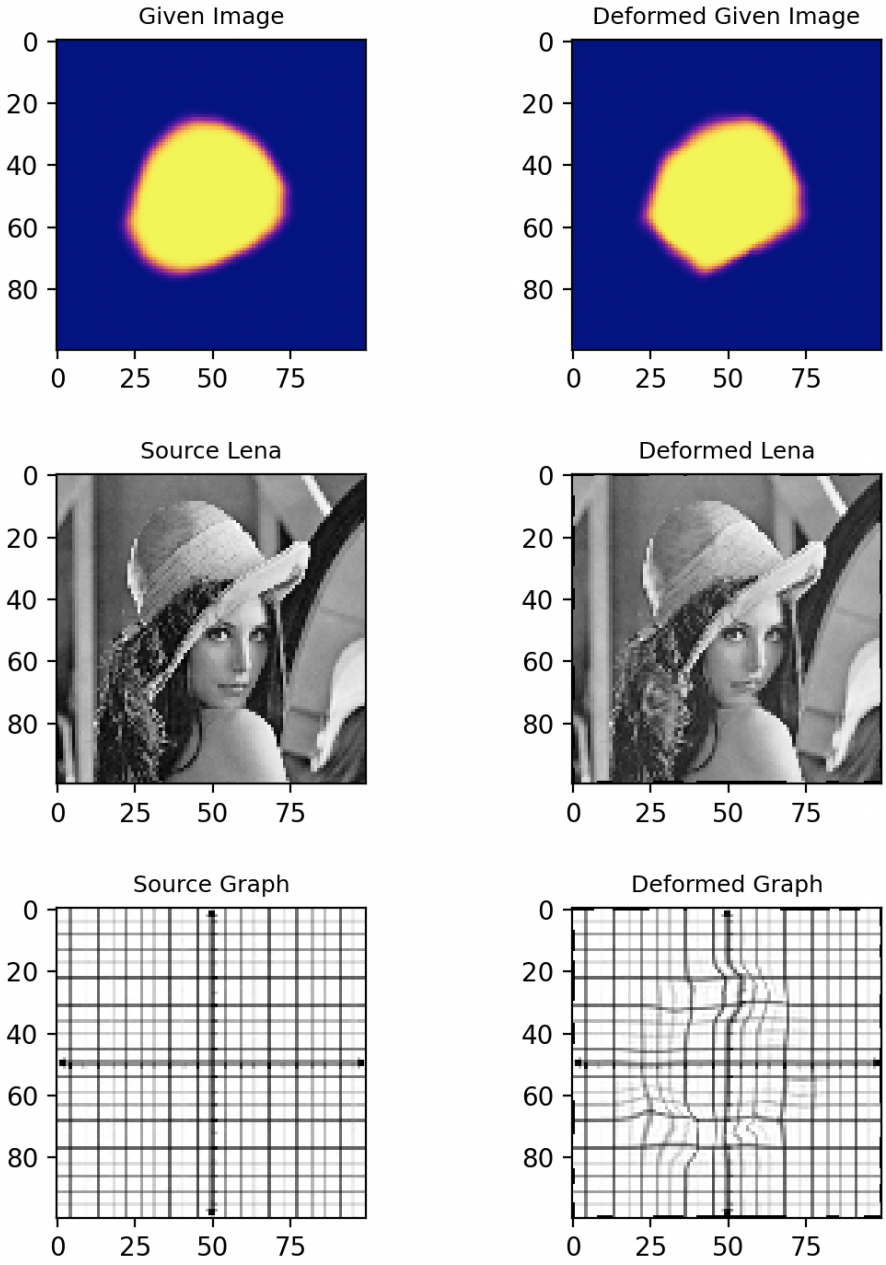
```
#Perform Eulers with smoothing operator
dvx_dt = -1*self.smoothing(jacobian_x + self.divx(vx,vy)) #dvx/dt
dvy_dt = -1*self.smoothing(jacobian_y + self.divy(vx,vy)) #dvy/dt
vx += dvx_dt * 1/STEPS
vy += dvy_dt * 1/STEPS
```

Finally, we perform Euler's on our ϕ_t. We use the map_coordinates function, denoted as 'mc,' with order=3 to interpolate our v_t with our ϕ_t. Our ϕ_t term is divided into two components: one for x and one for y.

```
#Perform interpolation to find dphi_dt
dphi_x_dt = mc(vx, (phi_x, phi_y), order=3)
dphi_y_dt = mc(vy, (phi_x,phi_y), order=3)

phi_x += dphi_x_dt * 1/STEPS
phi_y += dphi_y_dt * 1/STEPS
```

To best visualize our transformation ϕ_t, we can interpolate our source images with ϕ_t, deforming it. We plot our deformed results against our source images as shown below:

```
#Interpolate source images by phi
    #Display initial conditions
    ax[0][0].imshow(src, cmap = 'plasma')
    ax[0][0].set_title("Given Image", fontsize=9)
    ax[1][0].imshow(src2, cmap='gray')
    ax[1][0].set_title("Source Lena", fontsize=9)
    ax[2][0].imshow(src3, cmap='gray')
    ax[2][0].set_title("Source Graph", fontsize=9)

    dst = mc(src, (phi_x,phi_y), order=3)
    dst2 = mc(src2, (phi_x,phi_y), order=3)
    dst3 = mc(src3, (phi_x,phi_y), order=3)

    #Show Results
    ax[0][1].imshow(dst, cmap='plasma')
    ax[0][1].set_title("Deformed Given Image", fontsize=9)
    ax[1][1].imshow(dst2, cmap='gray')
    ax[1][1].set_title("Deformed Lena", fontsize=9)
    ax[2][1].imshow(dst3, cmap='gray')
    ax[2][1].set_title("Deformed Graph", fontsize=9)
    plt.show()
```



As we can see, the transformation ϕ_1 causes almost a circular rotation of certain points, leading to the images appearing to be bent inward on the lower right side. We continue testing similar transformation in part 4.

Problem 4 *Bonus!*

Compute the final transformation ϕ_1 at time point $t = 1$ by the following strategy. Compute the deformed image and compare the differences between the final transformations ϕ_1 and above.

We are given the following new formulas:

$$\frac{dv_t}{dt} = K[(Dv_t)^T \cdot v_t + div(v_t v_t^T)]$$
$$\frac{d\phi_t}{dt} = -D\phi_t \cdot v_t$$

We do not need to change much in our algorithm. However, we must compute the jacobian components of ϕ_t and multiply them by v_t in order to get components $\frac{d\phi_{tx}}{dt}$ and $\frac{d\phi_{ty}}{dt}$ for our $\frac{d\phi_t}{dt}$. Similar to part 3 where we computed components for $\frac{d}{dt}$ (and we will still do so again), we also do the same for $\frac{d\phi_t}{dt}$ in order to avoid complicated calculations with matrices with greater than 2 dimensions.

We can compute our jacobian of ϕ multiplied by our v_t using the component equations $\phi_{xx} \cdot v_x + \phi_{xy} \cdot v_y$ and $\phi_{yx} \cdot v_x + \phi_{yy} \cdot v_y$. The code is as follows:

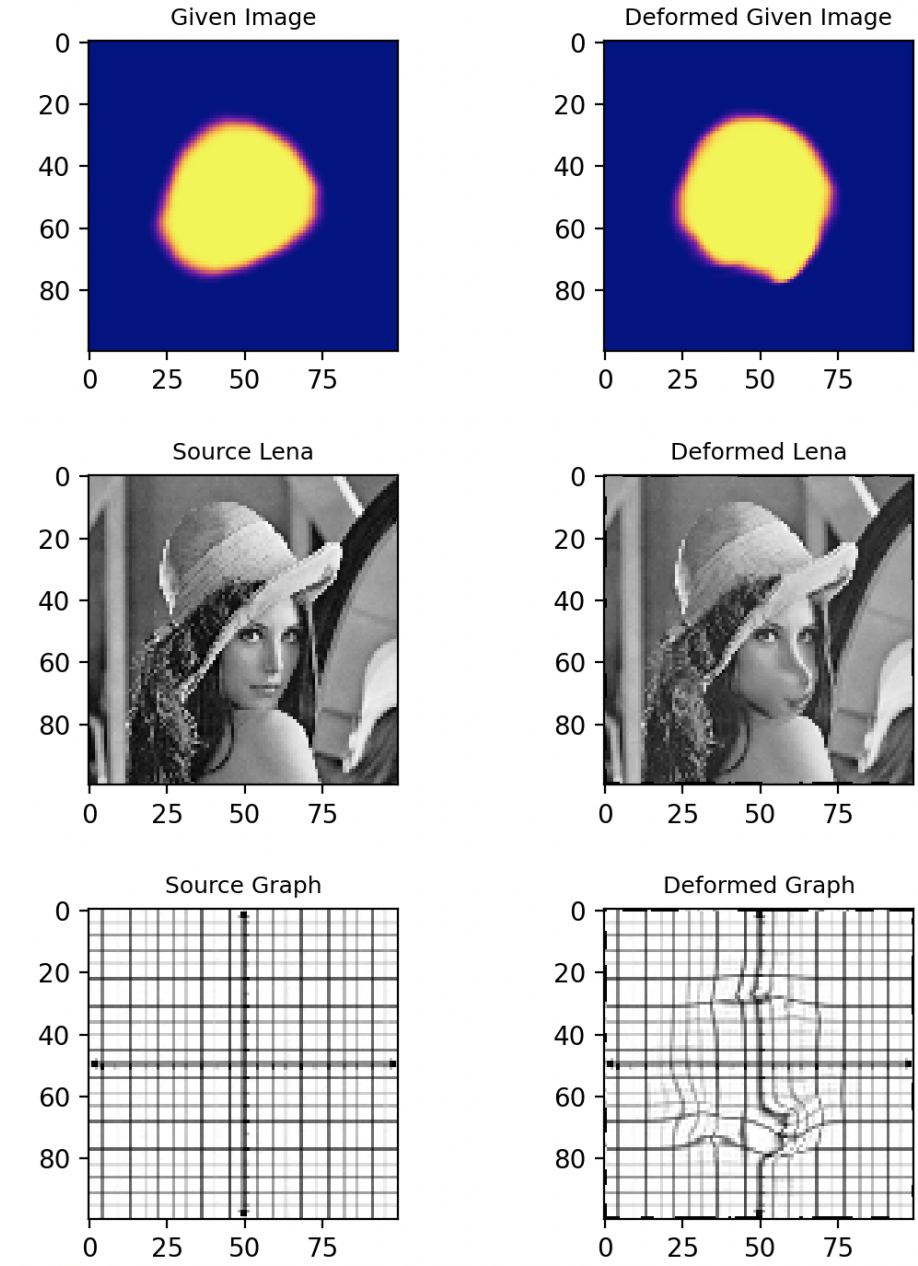
```
#Find Jacobian components of phi matrix
dphi_xx_dt = self.dx(phi_x)
dphi_xy_dt = self.dx(phi_y)
dphi_yx_dt = self.dy(phi_x)
dphi_yy_dt = self.dx(phi_y)

#Compute Jacobian term multiplied with phi matrix
jacobian_phi_x = dphi_xx_dt*vx + dphi_xy_dt*vy
jacobian_phi_y = dphi_yx_dt*vx  + dphi_yy_dt*vy
```

To change the way we do our jacobian of v_t in the calculation for $\frac{dv_x}{dt}$ and $\frac{dv_y}{dt}$, we simply remove the multiplication by -1 in our code:

```
#Compute Jacobian term multiplied with velocity matrix
jacobian_x = vxx*vx + vxy*vy
jacobian_y = vyx*vx + vyy*vy

#Perform Eulers with smoothing operator
dvx_dt = self.smoothing(jacobian_x + self.divx(vx,vy)) #dvx/dt
dvy_dt = self.smoothing(jacobian_y + self.divy(vx,vy)) #dvy/dt
```

```
vx += dvx_dt * 1/STEPS
vy += dvy_dt * 1/STEPS
```

The new formula we are given to compute our deformation field is much more drastic and makes "sharper" changes. Additionally, instead of deforming the pixels inward towards the center like the deformation in 3, the deformation in 4 appears to bend outward, towards the bottom right corner of the image. This can especially be seen with Lena, with her nose bent outward towards the bottom right of the image.



All graphics can be found within the root directory of the Problem Set 2 submission to be viewed at full scale.