Question 1: Singular Value Decomposition (SVD)

The SVD of a matrix $A \in \mathbb{R}^{m \times n}$ is a factorization of the form $A = U\Sigma V^T$, where:

- $U \in \mathbb{R}^{m \times m}$ is an orthogonal matrix whose columns are the left singular vectors of A.
- $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix with non-negative real numbers on the diagonal known as singular values.
- $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix whose columns are the right singular vectors of A.

To find the SVD of a given matrix, we typically follow these steps:

1. Compute $A^T A$ and AA^T.
2. Find the eigenvalues and the corresponding eigenvectors for $A^T A$ and AA^T.
3. Arrange the eigenvalues in decreasing order to form the diagonal of Σ.
4. The eigenvectors of $A^T A$ form the columns of V, and the eigenvectors of AA^T form the columns of U.

Let's proceed with the calculations.

(a) Matrix: $\begin{pmatrix} 3 & 0 \\ 0 & -2 \end{pmatrix}$

Since the matrix is diagonal, the SVD is simply the matrix itself with the singular values being the absolute values of the diagonal elements.

$$U = V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$$

Thus, the SVD of the matrix is:

$$A = U\Sigma V^T$$

(b) Matrix: $\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$

Again, since the matrix is diagonal, the SVD is the matrix itself with the singular values being the absolute values of the diagonal elements.

$$U = V = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$$

The SVD of the matrix is:

$$A = U\Sigma V^T$$

(c) Matrix: $\begin{pmatrix} 0 & 2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$

We calculate $A^T A$ and find its eigenvalues and eigenvectors.

$$A^T A = \begin{pmatrix} 0 & 0 \\ 0 & 4 \end{pmatrix}, \quad \text{eigenvalues are } \lambda_1 = 4, \lambda_2 = 0$$

The singular values are the square roots of the eigenvalues.

$$U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 2 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The SVD of the matrix is:

$$A = U\Sigma V^T$$

(d) Matrix: $\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$

We calculate $A^T A$ and find its eigenvalues and eigenvectors.

$$A^T A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \text{eigenvalues are } \lambda_1 = 2, \lambda_2 = 0$$

The singular values are the square roots of the eigenvalues.

$$U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \sqrt{2} & 0 \\ 0 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

The SVD of the matrix is:

$$A = U\Sigma V^T$$

(e) Matrix: $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$

We calculate $A^T A$ and find its eigenvalues and eigenvectors.

$$A^T A = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}, \quad \text{eigenvalues are } \lambda_1 = 4, \lambda_2 = 0$$

The singular values are the square roots of the eigenvalues.

$$U = \begin{pmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

The SVD of the matrix is:

$$A = U\Sigma V^T$$

Question 2: Eigenvalues and Eigenvectors

(a)

Given matrix A:

$$A = \begin{pmatrix} 2 & 2 \\ 5 & 1 \end{pmatrix}$$

The characteristic equation is obtained by calculating the determinant of $A - \lambda I$, where λ is an eigenvalue of A:

$$\det(A - \lambda I) = \det \begin{pmatrix} 2-\lambda & 2 \\ 5 & 1-\lambda \end{pmatrix} = (2-\lambda)(1-\lambda) - (2 \cdot 5)$$

$$= \lambda^2 - 3\lambda - 8 = 0$$

Solving this quadratic equation gives us the eigenvalues $\lambda_1 \approx 4.70156212$ and $\lambda_2 \approx -1.70156212$.

To find the corresponding eigenvectors, we solve $(A - \lambda_i I)\mathbf{x} = 0$ for each λ_i.

For λ_1:

$$\begin{pmatrix} 2 - 4.70156212 & 2 \\ 5 & 1 - 4.70156212 \end{pmatrix} \mathbf{x} = 0$$

The normalized eigenvector is approximately $\begin{pmatrix} 0.59500525 \\ 0.80372182 \end{pmatrix}$.

For λ_2:

$$\begin{pmatrix} 2 - (-1.70156212) & 2 \\ 5 & 1 - (-1.70156212) \end{pmatrix} \mathbf{x} = 0$$

The normalized eigenvector is approximately $\begin{pmatrix} -0.47536171 \\ 0.87979045 \end{pmatrix}$.

(b)

To prove that if a real matrix A has unique eigenvalues, then the eigenvectors are linearly independent, we proceed by contradiction.

Suppose A has unique eigenvalues λ_1 and λ_2 with corresponding eigenvectors $\mathbf{x}_1$ and $\mathbf{x}_2$ that are not linearly independent. This implies there exists a non-zero scalar c such that $c\mathbf{x}_1 = \mathbf{x}_2$.

Applying A to both sides gives:

$$A(c\mathbf{x}_1) = A\mathbf{x}_2$$

$$cA\mathbf{x}_1 = \lambda_2\mathbf{x}_2$$

$$c\lambda_1\mathbf{x}_1 = \lambda_2\mathbf{x}_2$$

$$c\lambda_1\mathbf{x}_1 = \lambda_2(c\mathbf{x}_1)$$

$$c\lambda_1 = c\lambda_2$$

Since c is non-zero, we can divide both sides by c to obtain $\lambda_1 = \lambda_2$, which is a contradiction because we assumed that the eigenvalues are unique. Therefore, the eigenvectors must be linearly independent.

HW3 - Q3: Power method (30 points)

Notes:

- For programming solutions, properly add comments to your code.

(a) Write function `power_method(A,x)`, which takes as input matrix A and a vector x, and uses power method to calculate eigenvalue and eigenvector. Get the largest (in absolute value) eigenvalue and the corresponding eigenvector for matrix $A=\begin{bmatrix} 2 & 1 & 2 \\ 1 & 3 & 2 \\ 2 & 4 & 1 \end{bmatrix}$

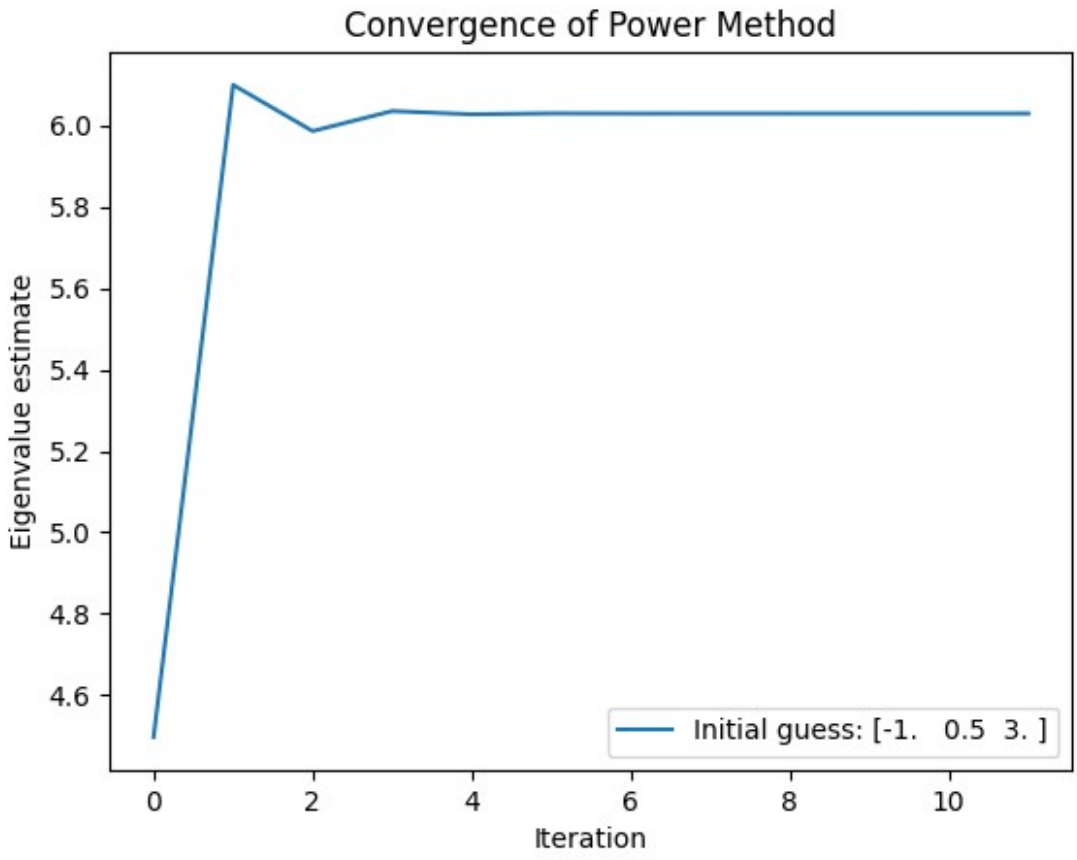
using above function. Start with intial eigenvector guesses: `[-1, 0.5, 3]` and `[2,-6,0.2]`. For each of the vectors, iterate until convergence. Plot how the eigenvalue changes w.r.t. iterations. Report the number of steps it took to converge, eigenvalue and eigenvector. Match your output with the results generated by the numpy API: `numpy.linalg.eig`

Note that you only need to look at magnitudes of eigenvalues. Use an absolute tolerance of 10^{-6} between eigenvalue output of previous and current iteration as stopping criteria. You may also need to normalize the final eigenvector to match with output of numpy API `numpy.linalg.eig`. (15 points)

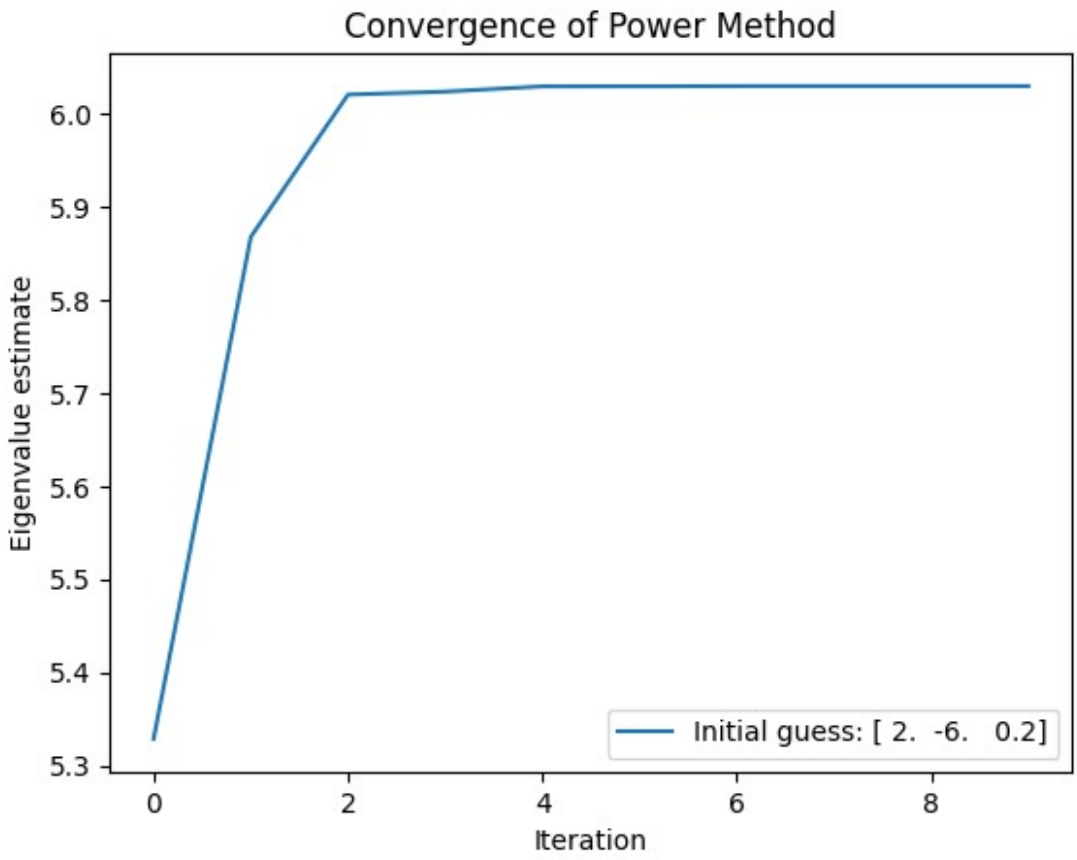
```
# !!!! YOUR CODE HERE !!!!
# Here is the Python code for part (a) with step-by-step comments.
import numpy as np
import matplotlib.pyplot as plt

def power_method(A, x, tol=1e-6, max_iterations=10000):
    """
    Power method for approximating the largest eigenvalue and
corresponding eigenvector.

    Parameters:
    A (ndarray): A square matrix.
    x (ndarray): The initial guess for the eigenvector.
    tol (float): The tolerance for convergence.
    max_iterations (int): The maximum number of iterations.

    Returns:
```

```
    lambda_new (float): The approximated largest eigenvalue.
    x (ndarray): The corresponding normalized eigenvector.
    num_iterations (int): The number of iterations to converge.
    eigenvalue_history (list): History of eigenvalue approximations
for plotting.
    """
    n = A.shape[0]  # Get the size of the matrix
    x = x / np.linalg.norm(x)  # Normalize initial vector
    lambda_old = 0  # Initialize old lambda for convergence check
    eigenvalue_history = []  # List to store the history of
eigenvalues

    for i in range(max_iterations):
        y = np.dot(A, x)  # Multiply A with the current eigenvector
estimate
        x = y / np.linalg.norm(y)  # Update the eigenvector estimate
and normalize

        lambda_new = np.dot(x.T, np.dot(A, x))  # Calculate the
Rayleigh quotient
        eigenvalue_history.append(lambda_new)  # Append the new
eigenvalue to the history

        if np.abs(lambda_new - lambda_old) < tol:  # Check for
convergence
            break  # Exit the loop if converged

        lambda_old = lambda_new  # Update old lambda

    num_iterations = i + 1  # Number of iterations it took to converge

    return lambda_new, x, num_iterations, eigenvalue_history

# Define matrix A and initial eigenvector guesses
A = np.array([[2, 1, 2],
              [1, 3, 4],
              [2, 2, 1]])
initial_guesses = [np.array([-1, 0.5, 3]), np.array([2, -6, 0.2])]

# Use the power method for each initial guess and plot the convergence
for guess in initial_guesses:
    lambda_power, eigenvector_power, iterations_power, history_power =
power_method(A, guess)
    plt.plot(history_power, label=f'Initial guess: {guess}')
    plt.xlabel('Iteration')
    plt.ylabel('Eigenvalue estimate')
    plt.title('Convergence of Power Method')
    plt.legend()
    plt.show()
    print(f"Converged in {iterations_power} iterations to eigenvalue
```

```
{lambda_power}")
    print(f"Corresponding eigenvector: {eigenvector_power}")

# Validate against numpy.linalg.eig
eigenvalues, eigenvectors = np.linalg.eig(A)
print("NumPy eigenvalues:", eigenvalues)
print("NumPy eigenvectors:", eigenvectors)
```




```
Converged in 12 iterations to eigenvalue 6.0291119632857
Corresponding eigenvector: [0.42697963 0.76905767 0.47564556]
```




```
Converged in 10 iterations to eigenvalue 6.029111188004842
Corresponding eigenvector: [-0.42697924 -0.76905805 -0.47564529]
NumPy eigenvalues: [ 6.02911192  1.33625596 -1.36536788]
NumPy eigenvectors: [[-0.42697964 -0.7120039  -0.25611215]
 [-0.76905768  0.69343836 -0.62050013]
 [-0.47564554 -0.11042501  0.74120588]]
```

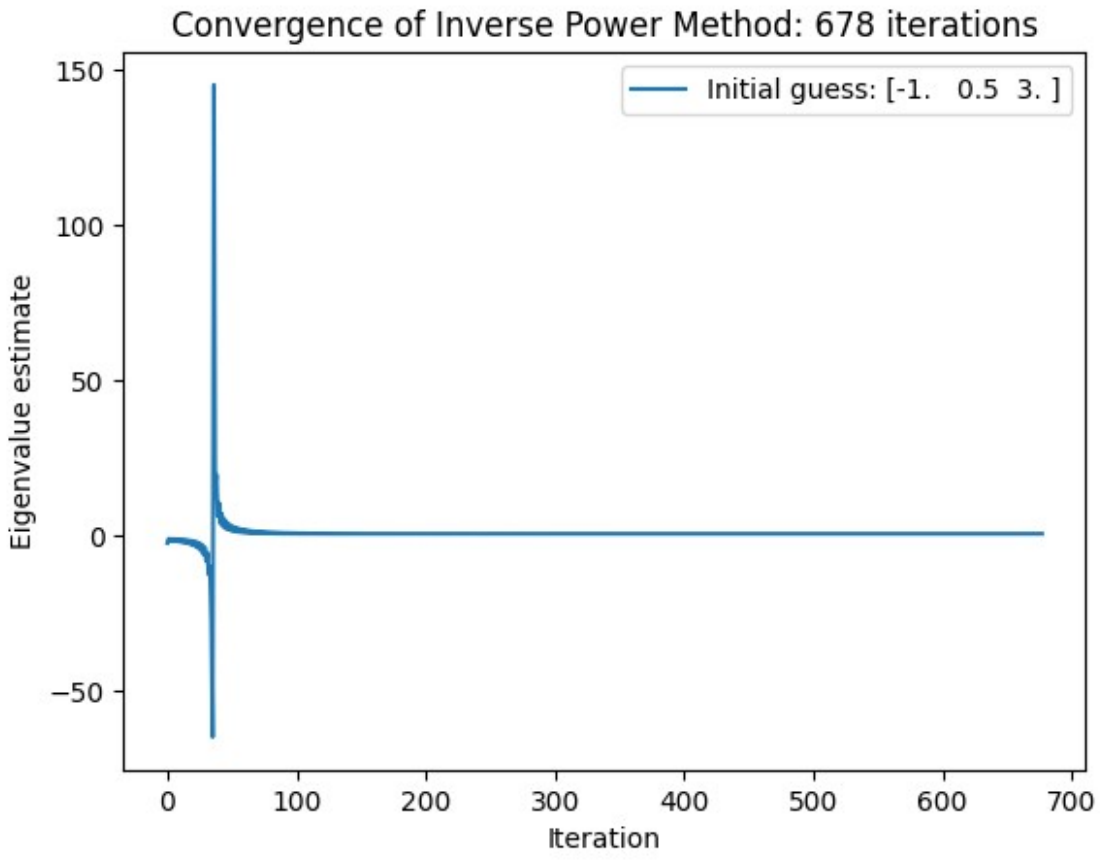
(b) Write function `inverse_power_method(A,x)`, which takes as input matrix A and a vector x, and uses inverse power method to calculate the smallest (in absolute value) eigenvalue and corresponding eigenvector. Solve for the smallest (in absolute value) eigenvalue and corresponding eigenvector for the matrix from (a). Use the same intial eigenvector guesses as (a). Report how many iterations do you need for it to converge to the smallest eigenvalue. Plot the computed/estimated eigenvalue w.r.t iterations. Report the final eigenvalue and eigenvector you get. Match your answer with the results generated by the numpy API `numpy.linalg.eig`.

Note that you only need to look at magnitudes of eigenvalues. Use an absolute tolerance of 10^{-6} between eigenvalue output of previous and current iteration as stopping criteria. You may also need to normalize the final eigenvector to match with output of numpy API `numpy.linalg.eig`. (15 points)

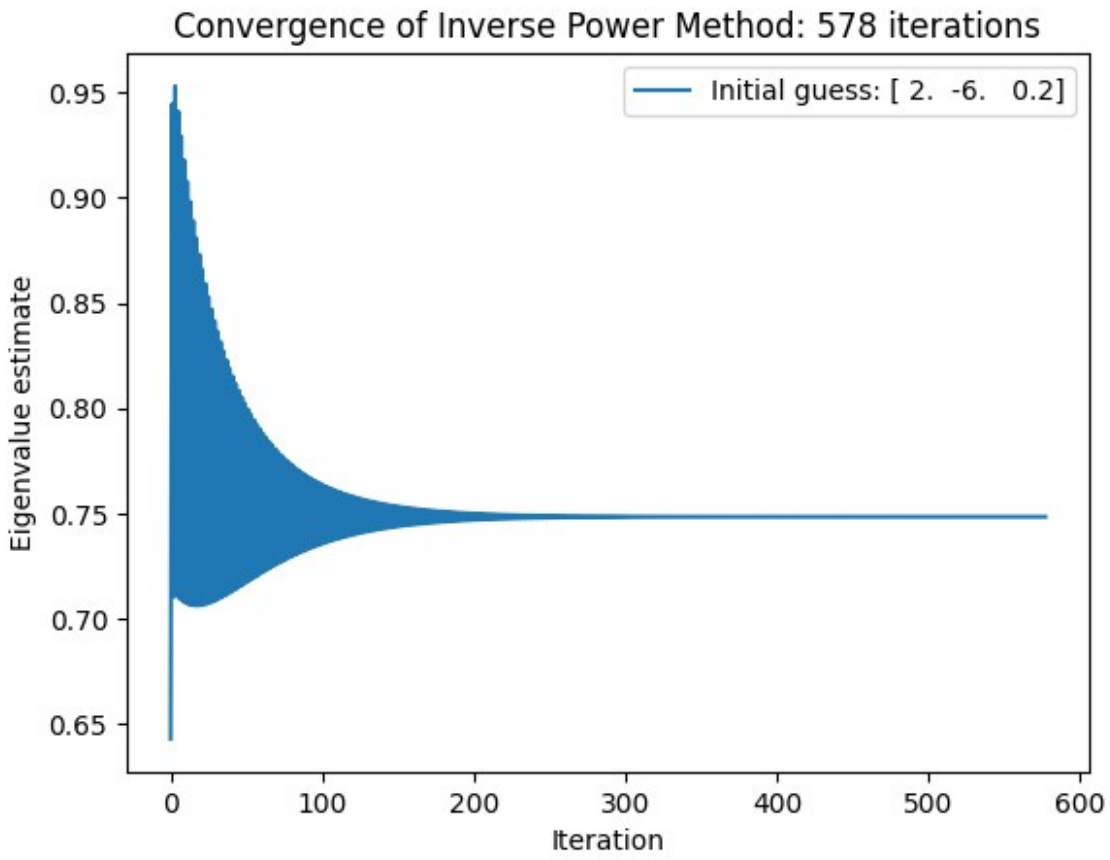
```
# !!!! YOUR CODE HERE !!!!
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg

def inverse_power_method(A, x, tol=1e-6, max_iterations=10000):
    """
    Inverse power method for approximating the smallest (in absolute
value) eigenvalue
    and corresponding eigenvector.

    Parameters:
    A (ndarray): A square matrix.
    x (ndarray): The initial guess for the eigenvector.
    tol (float): The tolerance for convergence.
    max_iterations (int): The maximum number of iterations.

    Returns:
    mu (float): The approximated smallest eigenvalue.
    x (ndarray): The corresponding normalized eigenvector.
    num_iterations (int): The number of iterations to converge.
    eigenvalue_history (list): History of eigenvalue approximations
for plotting.
    """
    n = A.shape[0]
    x = x / np.linalg.norm(x)  # Normalize the initial vector
    mu_old = 0
    eigenvalue_history = []
```

```
    # Precompute the LU decomposition of A for solving linear systems
    lu_and_piv = scipy.linalg.lu_factor(A)

    for i in range(max_iterations):
        # Solve the linear system A*y = x for y
        y = scipy.linalg.lu_solve(lu_and_piv, x)

        # Normalize the vector y to obtain the new eigenvector
estimate
        x = y / np.linalg.norm(y)

        # Calculate the Rayleigh quotient as the eigenvalue estimate
        mu_new = np.dot(x.T, np.dot(A, x))

        # Store the history of eigenvalue approximations for later
analysis
        eigenvalue_history.append(1 / mu_new)

        # Check for convergence based on the change in estimated
eigenvalues
        if np.abs(1 / mu_new - mu_old) < tol:
            break

        mu_old = 1 / mu_new  # Update the old eigenvalue estimate

    num_iterations = i + 1
    mu = 1 / mu_new  # The estimated eigenvalue is the reciprocal of
the Rayleigh quotient

    return mu, x, num_iterations, eigenvalue_history

# Define the matrix A
A = np.array([[2, 1, 2],
              [1, 3, 4],
              [2, 2, 1]])

# Initial eigenvector guesses
initial_guesses = [np.array([-1, 0.5, 3]), np.array([2, -6, 0.2])]

# Apply the inverse power method and plot the results
for guess in initial_guesses:
    mu_inverse, eigenvector_inverse, iterations_inverse,
history_inverse = inverse_power_method(A, guess)
    plt.figure()
    plt.plot(history_inverse, label=f'Initial guess: {guess}')
    plt.xlabel('Iteration')
    plt.ylabel('Eigenvalue estimate')
    plt.title(f'Convergence of Inverse Power Method:
{iterations_inverse} iterations')
```

```
    plt.legend()
    plt.show()
    print(f"Initial guess {guess} converged in {iterations_inverse}
iterations to eigenvalue {mu_inverse} and eigenvector
{eigenvector_inverse}")

# Compare with numpy's eigenvalue calculation
eigenvalues, eigenvectors = np.linalg.eig(A)
smallest_eigenvalue_index = np.argmin(np.abs(eigenvalues))
print(f"NumPy smallest eigenvalue:
{eigenvalues[smallest_eigenvalue_index]}, eigenvector:
{eigenvectors[:, smallest_eigenvalue_index]}")
```




```
Initial guess [-1.   0.5  3. ] converged in 678 iterations to
eigenvalue 0.748359127585521 and eigenvector [-0.71200438  0.69343798
-0.11042432]
```




```
Initial guess [ 2.  -6.   0.2] converged in 578 iterations to
eigenvalue 0.7483601022960851 and eigenvector [ 0.71200342 -0.69343875
0.1104257 ]
NumPy smallest eigenvalue: 1.336255963212595, eigenvector: [-0.7120039
0.69343836 -0.11042501]
```

HW3 - Q4: Face Recognition with Eigenfaces (10 points)

Keywords: Principal Component Analysis (PCA), Eigenvalues and Eigenvectors

About the dataset: *Labeled Faces in the Wild* dataset consists of face photographs designed for studying the problem of unconstrained face recognition. The original dataset contains more than 13,000 images of faces collected from the web.

Agenda:

- In this programming challenge, you will be performing face recognition on the *Labeled Faces in the Wild* dataset using PyTorch.
- First, you will do Principal Component Analysis (PCA) on the image dataset. PCA is used for dimentionality reduction which is a type of unsupervised learning.
- You will be applying PCA on the dataset to extract the principal components (Top k *eigenvalues*).
- As you will see eventually, the reconstruction of faces from these *eigenvalues* will give us the *eigen-faces* which are the most representative features of most of the images in the dataset.
- Finally, you will train a simple PyTorch Neural Network model on the modified image dataset.
- This trained model will be used prediction and evaluation on a test set.

Note:

- Run all the cells in order.
- **Do not edit** the cells marked with !!DO NOT EDIT!!
- Only **add your code** to cells marked with !!!! YOUR CODE HERE !!!!
- Do not change variable names, and use the names which are suggested.

```
# !!DO NOT EDIT!!
# loading the dataset directly from the scikit-learn library (can take
about 3-5 mins)
import numpy as np
from sklearn.datasets import fetch_lfw_people
dataset = fetch_lfw_people(min_faces_per_person=80)

# each 2D image is of size 62 x 47 pixels, represented by a 2D array.
# the value of each pixel is a real value from 0 to 255.
count, height, width = dataset.images.shape
print('The dataset type is:',type(dataset.images))
print('The number of images in the dataset:',count)
print('The height of each image:',height)
```

```
print('The width of each image:',width)

# sklearn also gives us a flattened version of the images which is a
vector of size 62 x 47 = 2914.
# we can directly use that for our exercise
print('The shape of data is:',dataset.data.shape)
```

```
The dataset type is: <class 'numpy.ndarray'>
The number of images in the dataset: 1140
The height of each image: 62
The width of each image: 47
The shape of data is: (1140, 2914)
```

For optimum performance, we have only considered people who have more than 80 images. This restriction notably reduces the size of the dataset. Now let us look at the labels of the people present in the dataset

```
# !!DO NOT EDIT!!
# create target label - target name pairs
targets = [(x,y) for x,y in zip(range(len(np.unique(dataset.target))),
dataset.target_names)]
print('The target labels and names are:\n', targets)
```

```
The target labels and names are:
 [(0, 'Colin Powell'), (1, 'Donald Rumsfeld'), (2, 'George W Bush'),
(3, 'Gerhard Schroeder'), (4, 'Tony Blair')]
```

(a) Preprocessing: Using the `train_test_split` API from sklearn, split the data into train and test dataset in the ratio 3:1. Use `random_state=42`.

For better performance, it is recommended to normalize the features which can have different ranges with huge values. As all our features here are in the range [0,255], it is not explicitly needed here. However, it is a good exercise. Use the `StandardScaler` class from sklearn and use that to normalize X_train and X_test. Validate and show your result by printing the first 5 columns of 5 images of X_train (This result can vary from pc to pc). (1.5 points)

```
# !!DO NOT EDIT!!
X = dataset.data
y = dataset.target

#######
# !!!! YOUR CODE HERE !!!!
# Import necessary libraries
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    dataset.data, dataset.target, test_size=0.25, random_state=42
)

# Initialize a StandardScaler instance
scaler = StandardScaler()

# Fit the scaler on the training data and transform both the training
and test data
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Print the first 5 columns of 5 images from X_train to validate
print(X_train[:5, :5])

# output variable names -  X_train, X_test, y_train, y_test
#######
```

```
[[-1.0548956  -1.1314403  -1.1468822  -0.85341436 -0.7309732 ]
 [ 2.6098442   2.4664047   1.6027023   0.7667316   0.23976761]
 [-1.0626272  -1.0687327  -1.1071484  -1.1075549   1.3457758 ]
 [-0.1735025  -0.2221809  -0.7018628  -1.3775792  -1.3436539 ]
 [ 0.05844303  0.00513394 -0.03433381 -0.20217922 -0.0466805 ]]
```

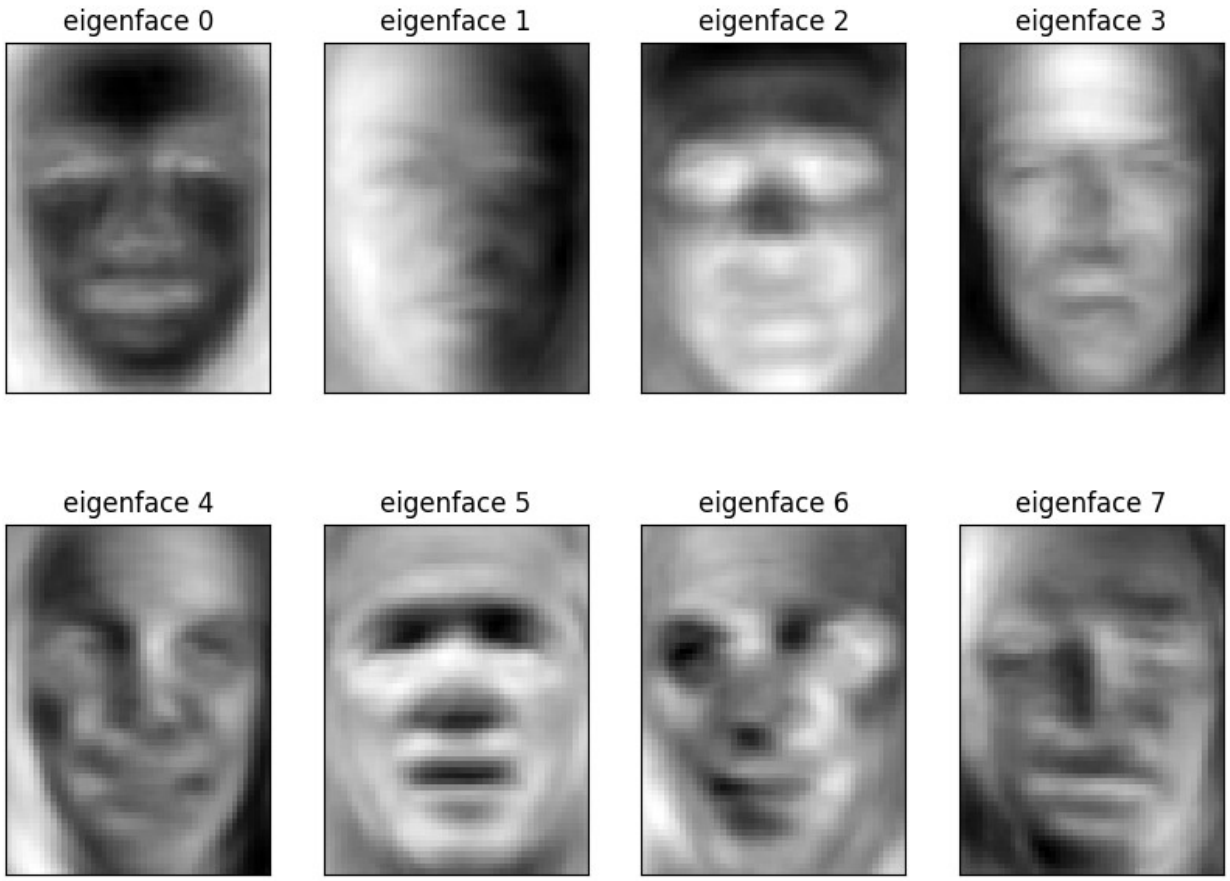
(b) Dimentionality reduction : In this section, use the `PCA` API from sklearn to extract the top 100 principal components of the image matrix and fit it on the training dataset. We can then visualize some of the top few components as an image (eigenfaces). (1.5 points)

```
#######
# !!!! YOUR CODE HERE !!!!
# initialize PCA API from sklearn with n_components. Also set
svd_solver="randomized" and whiten=True in the initialization
parameters.
from sklearn.decomposition import PCA

# Initialize PCA with the top 100 components
pca = PCA(n_components=100, svd_solver='randomized', whiten=True)

# Fit PCA on the training data
pca.fit(X_train)

# Now you can visualize some of the top eigenfaces if required
```

```
# output variable name -  pca
#######

PCA(n_components=100, svd_solver='randomized', whiten=True)
```

Now we will plot the most representative eigenfaces:

```
# !!DO NOT EDIT!!
# Helper function to plot
import matplotlib.pyplot as plt
def plot_gallery(images, titles, height, width, n_row=2, n_col=4):
    plt.figure(figsize=(2* n_col, 3 * n_row))
    plt.subplots_adjust(bottom=0, left=0.01, right=0.99, top=0.90,
hspace=0.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((height, width)),
cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())

# !!DO NOT EDIT!!
# get the 100 eigen faces and reshape them to original image size
which is 62 x 47 pixels
n_components = 100
eigenfaces = pca.components_.reshape((n_components, height, width))

# plot the top 8 eigenfaces
eigenface_titles = ["eigenface %d" % i for i in
range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, height, width)

plt.show()
```



(c) Face reconstruction: In this section, we will reconstruct an image from its point projected on the principal component basis. Project the first three faces on the eigenvector basis using PCA models trained with varying number of principal components. Using the projected points, reconstruct the faces, and visualize the images. Your final output should be a 3×5 image matrix, where the rows are the data points, and the columns correspond to original image and reconstructed image for n_components$\in [10, 100, 150, 500]$. (3.5 points)

```
#######
# !!!! YOUR CODE HERE !!!!
# Reconstruct faces for different numbers of principal components
n_components_list = [10, 100, 150, 500]
reconstructed_faces = []

for n_components in n_components_list:
    # Initialize a new PCA model
    pca_temp = PCA(n_components=n_components)
```

```
    # Fit the model and transform the data
    X_train_pca = pca_temp.fit_transform(X_train)

    # Inverse transform the data to reconstruct the faces
    X_train_inv = pca_temp.inverse_transform(X_train_pca)

    # Append the reconstructed faces to the list
    reconstructed_faces.append(X_train_inv[0])

# You would then need to write code to visualize these reconstructed
faces in a 3x5 matrix

#######
```

(d) Prediction: In this section, we will train a neural network classifier in **PyTorch** on the transformed dataset. This classifier will help us with the face recognition task. Complete each of the steps below.

For PyTorch reference see documentation. (3.5 points)

```
# !!DO NOT EDIT!!
# define imports here
import torch
import torch.nn as nn
```

Before we start training, we need to transform the training and test dataset to reduced forms (100 dimensions) using the pca function defined in (b).

we will also need to move the train and test dataset to torch tensors in order to work with pytorch.

```
#######
# !!!! YOUR CODE HERE !!!!
# 1. project X_train and X_test on orthonormal basis using the PCA API
initialized in part (b).

import torch
from torch import nn
from torch.optim import SGD

# Convert the dataset to torch tensors
X_train_pca_torch = torch.tensor(pca.transform(X_train)[:,:100],
dtype=torch.float32)
X_test_pca_torch = torch.tensor(pca.transform(X_test)[:,:100],
```

```
dtype=torch.float32)
y_train_torch = torch.tensor(y_train, dtype=torch.long)
y_test_torch = torch.tensor(y_test, dtype=torch.long)

# Define the neural network architecture
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(100, 1024)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(1024, 5)
        self.log_softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.log_softmax(x)
        return x

# Instantiate the model, loss function, and optimizer
model = MLP()
criterion = nn.NLLLoss()
optimizer = SGD(model.parameters(), lr=1e-1)

# Train the model (you will need to add the code to loop over epochs)
for epoch in range(501):
    optimizer.zero_grad()
    output = model(X_train_pca_torch)
    loss = criterion(output, y_train_torch)
    loss.backward()
    optimizer.step()
    if epoch % 50 == 0:
        print(f'Epoch: {epoch} Loss: {loss.item()}')

# After training, you can predict and evaluate the model on the test
data

# 2. now convert X_train_pca, X_test_pca, y_train and y_test to
torch.tensor. For y_train and y_test, set dtype=torch.long


# output variable names -  X_train_pca_torch, X_test_pca_torch,
y_train_torch, y_test_torch
#######

Epoch: 0 Loss: 1.5808149576187134
Epoch: 50 Loss: 0.43134286999702454
Epoch: 100 Loss: 0.21753352880477905
Epoch: 150 Loss: 0.13241726160049438
```

```
Epoch: 200 Loss: 0.08917930722236633
Epoch: 250 Loss: 0.06448309123516083
Epoch: 300 Loss: 0.04915391281247139
Epoch: 350 Loss: 0.03900125250220299
Epoch: 400 Loss: 0.03192029893398285
Epoch: 450 Loss: 0.026773637160658836
Epoch: 500 Loss: 0.022902820259332657

#######
# !!!! YOUR CODE HERE !!!!
# 3. We will implement a simple multilayer perceptron (MLP) in pytorch
with one hidden layer.
# Using this neural network model, we will train on the transformed
dataset.
class MLP(torch.nn.Module):
  def __init__(self):
    super(MLP, self).__init__()
    # Initalize various layers of MLP as instructed below
    # DO: initialze two linear layers: 100 -> 1024  and 1024-> 5

    # DO: initialize relu activation function

    # DO: initialize LogSoftmax

  def forward(self, x):
    # DO: define the feedforward algorithm of the model and return the
final output

#######

#######
# !!!! YOUR CODE HERE !!!!
# 4. create an instance of the MLP class here

# 5. define loss (use negative log likelihood loss: torch.nn.NLLLoss)

# 6. define optimizer (use torch.optim.SGD (Stochastic Gradient
Descent)).
# Set learning rate to 1e-1 and also set model parameters

#######

# !!DO NOT EDIT!!
# 7. train the classifier on the PCA-transformed training data for 500
epochs
# This part is already implemented.
# Go through each step carefully and understand what it does.
for epoch in range(501):
  # reset gradients
  optimizer.zero_grad()
```

```
  # predict
  output=model(X_train_pca_torch)

  # calculate loss
  loss=criterion(output, y_train_torch)

  # backpropagate loss
  loss.backward()

  # performs a single gradient update step
  optimizer.step()

  if epoch%50==0:
    print('Epoch: {}, Loss: {:.3f}'.format(epoch, loss.item()))
```

```
Epoch: 0, Loss: 0.023
Epoch: 50, Loss: 0.020
Epoch: 100, Loss: 0.017
Epoch: 150, Loss: 0.016
Epoch: 200, Loss: 0.014
Epoch: 250, Loss: 0.013
Epoch: 300, Loss: 0.012
Epoch: 350, Loss: 0.011
Epoch: 400, Loss: 0.010
Epoch: 450, Loss: 0.009
Epoch: 500, Loss: 0.009
```

```
# !!DO NOT EDIT!!
# predict on test data
predictions = model(X_test_pca_torch) # gives softmax logits
y_pred = torch.argmax(predictions, dim=1).numpy() # get the labels
from prdictions: nx5 -> nx1

# !!DO NOT EDIT!!
# here, we will print the multi-label classification report:
precision, recall, f1-score etc.
from sklearn.metrics import classification_report
target_names=[y for x,y in targets]
print(classification_report(y_test, y_pred,
target_names=target_names))

# let us validate some of the predictions by plotting images
# display some of the results
def title(y_pred, y_test, target_names, i):
    pred_name = target_names[y_pred[i]].rsplit(" ", 1)[-1]
    true_name = target_names[y_test[i]].rsplit(" ", 1)[-1]
    return "predicted: %s\ntrue:      %s" % (pred_name, true_name)

prediction_titles = [
    title(y_pred, y_test, target_names, i) for i in
range(y_pred.shape[0])
```

```
]

plot_gallery(X_test, prediction_titles, height, width)
```

	precision	recall	f1-score	support
Colin Powell	0.90	0.88	0.89	64
Donald Rumsfeld	0.80	0.75	0.77	32
George W Bush	0.90	0.95	0.92	127
Gerhard Schroeder	1.00	0.83	0.91	29
Tony Blair	0.82	0.85	0.84	33
accuracy			0.89	285
macro avg	0.88	0.85	0.87	285
weighted avg	0.89	0.89	0.89	285