Homework 3

This notebook contains all code for Homework 3 of DSC 210.

- Question 3: Power method (20 points)
- Question 4: Face Recognition with Eigenfaces (30 points + 10 bonus points)

Notes:

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

Question 3: Power method (20 points)

Part (a): `power_method(A, x)`

Write function `power_method(A,x)`, which takes as input matrix A and a vector $\mathbf{x}$, and uses the power method to calculate eigenvalues and eigenvectors.

Get the largest **(in absolute value)** eigenvalue and the corresponding eigenvector for matrix A using the above function.

$$A = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 3 & 2 \\ 2 & 4 & 1 \end{bmatrix}$$

Start with intial eigenvector guesses: `[-1, 0.5, 3]` and `[2,-6,0.2]`. For each of the vectors, iterate until convergence.

(i) Plot how the eigenvalue changes w.r.t. iterations.

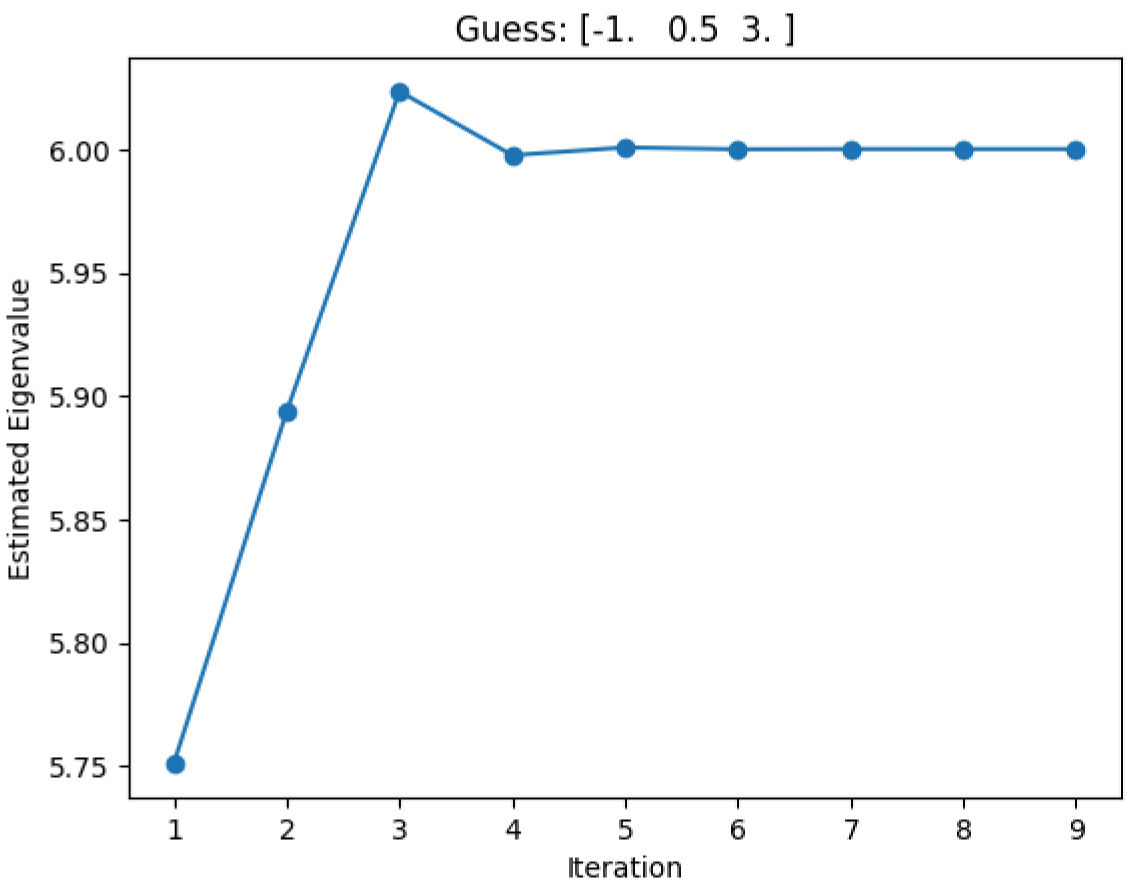
(ii) Report the number of steps it took to converge, for both the eigenvalue and eigenvector.

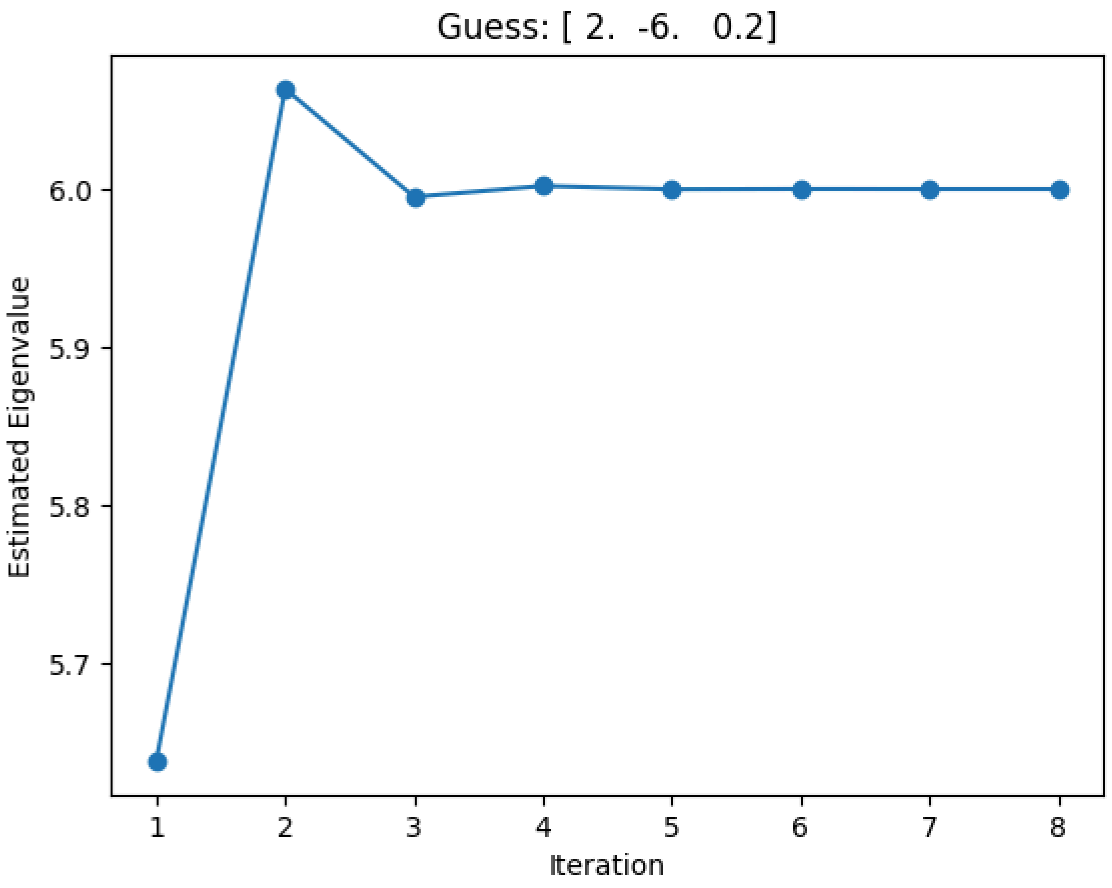
(iii) Report the final eigenvalue and eigenvector. Match your output with the results generated by the numpy API: `numpy.linalg.eig`

Note: 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`. (10 points)

In [22]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

```python
A = np.array([[2, 2, 1], [1, 3, 2], [2, 4, 1]])
# !!!! YOUR CODE HERE !!!!
def power_method(A, x):
    original_x = x.copy()
    eigenvalues = []
    steps = 0

    while True:
        x_new = A @ x / np.linalg.norm(A @ x)

        eigenvalue = x_new.T @ A @ x_new / (x_new.T @ x_new)
        eigenvalues.append(eigenvalue)

        if steps > 0 and np.allclose(eigenvalues[-1], eigenvalues[-2], atol=1e-6) a
            break

        x = x_new
        steps += 1

    plt.plot(range(1, steps + 2), eigenvalues, marker="o")
    plt.xlabel("Iteration")
    plt.ylabel("Estimated Eigenvalue")
    plt.title(f"Guess: {original_x}")
    plt.show()
    return eigenvalue, x_new

u1_guess = np.array([-1, 0.5, 3])
u2_guess = np.array([2, -6, 0.2])
eigenvalue1, eigenvector1 = power_method(A, u1_guess)
eigenvalue2, eigenvector2 = power_method(A, u2_guess)
print("Estimated Eigenvalue 1:", eigenvalue1)
print("Estimated Eigenvector 1:", eigenvector1)
print("Estimated Eigenvalue 2:", eigenvalue2)
print("Estimated Eigenvector 2:", eigenvector2)
print("Actual Eigenvalues and Eigenvectors:")
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```


Guess: [-1.   0.5 3.]
6.00
5.95
5.90
5.85
5.80
5.75
Estimated Eigenvalue
1
2
3
4
5
6
7
8
9
Iteration


Guess: [2. -6.   0.2]
6.0
5.9
5.8
5.7
Estimated Eigenvalue
1
2
3
4
5
6
7
8
Iteration

```
Estimated Eigenvalue 1: 6.000000515918747
Estimated Eigenvector 1: [0.46156625 0.59344262 0.65938036]
Estimated Eigenvalue 2: 6.000001525411784
Estimated Eigenvector 2: [-0.46156589 -0.59344306 -0.65938021]
Actual Eigenvalues and Eigenvectors:
Eigenvalues: [ 6.  1. -1.]
Eigenvectors: [[ 4.61566331e-01  8.94427191e-01  5.18104078e-17]
 [ 5.93442426e-01 -4.47213595e-01 -4.47213595e-01]
 [ 6.59380473e-01  3.01974496e-16  8.94427191e-01]]
```

Part (b): `inverse_power_method(A, x)`

Write function `inverse_power_method(A,x)`, which takes as input matrix A and a vector $\mathbf{x}$, 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).

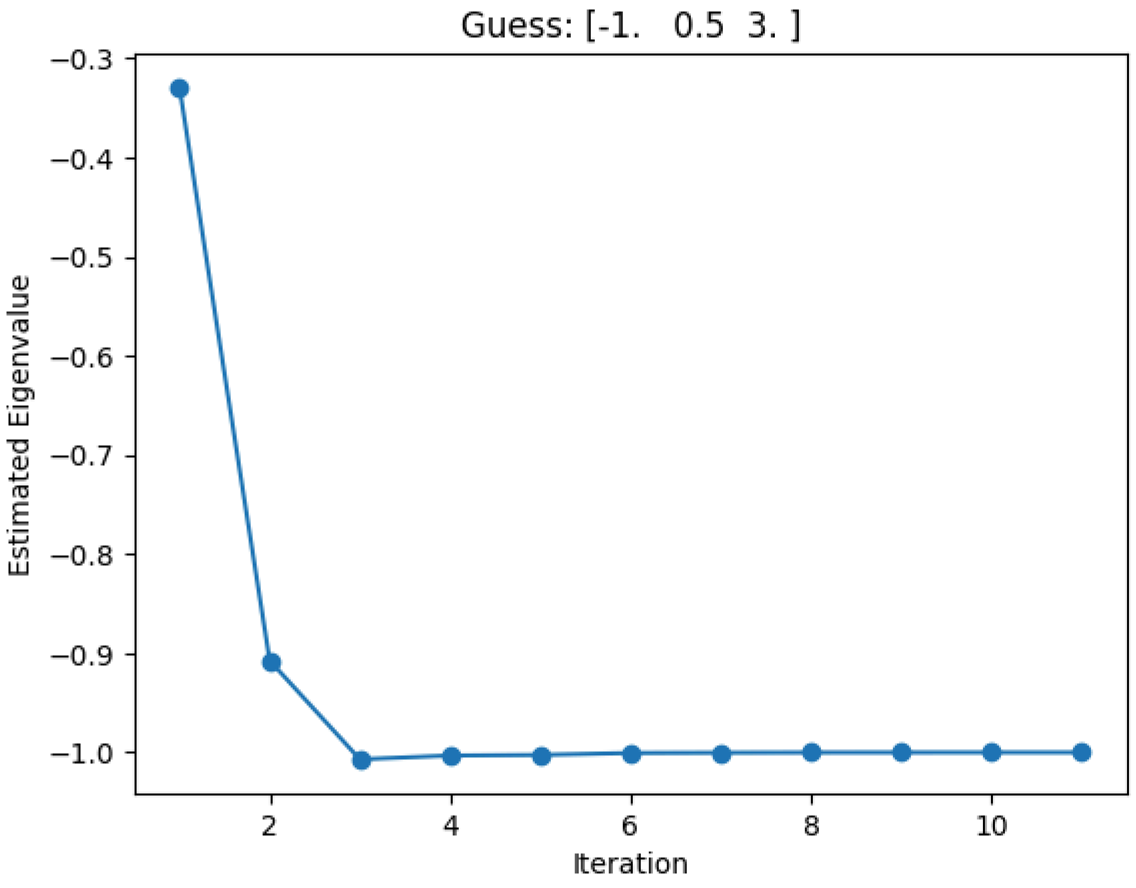
(i) Plot the computed/estimated eigenvalue with respect to iterations.

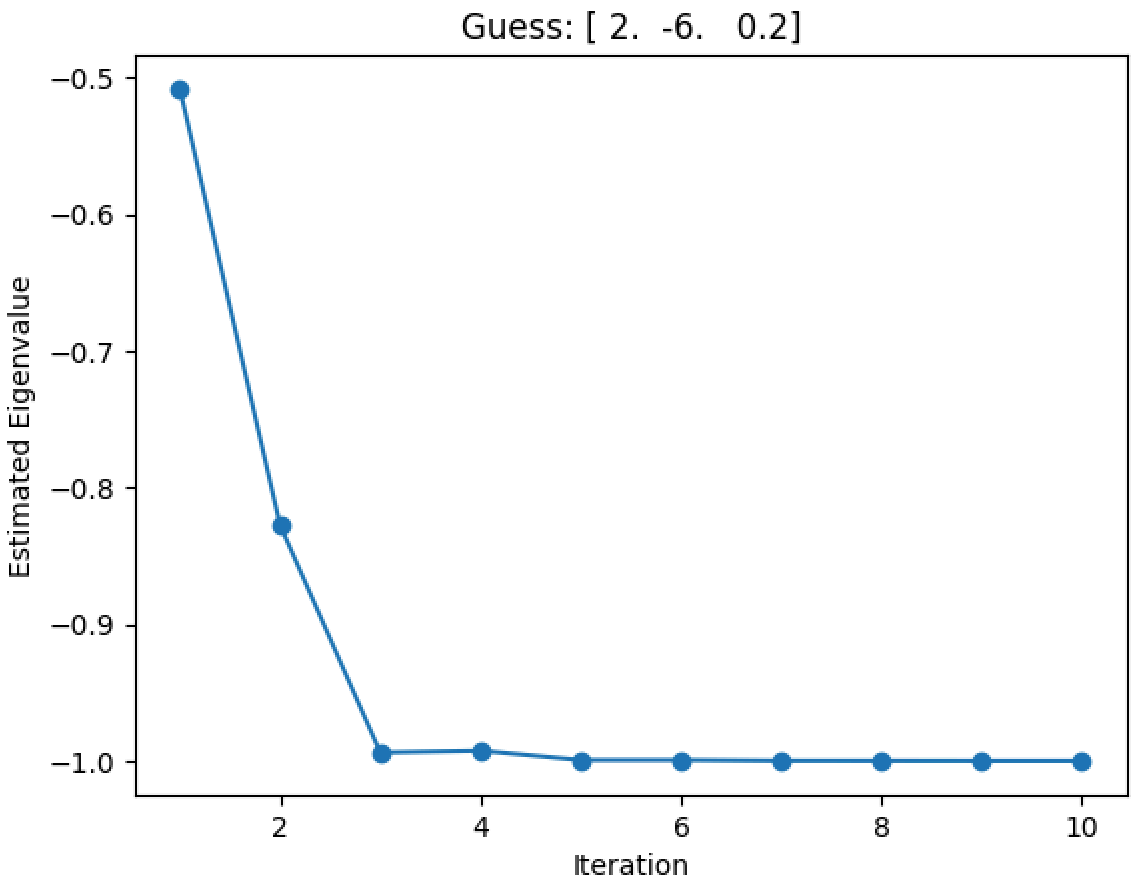
(ii) Report how many iterations do you need for it to converge to the smallest eigenvalue.

(iii) Report the final eigenvalue and eigenvector you get. Match your answer with the results generated by the **numpy** API `numpy.linalg.eig`.

Note: 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`. (10 points)

In [67]:
```python
# !!!! YOUR CODE HERE !!!!
def inverse_power_method(A, x, shift=2):
    original_x = x.copy()
    eigenvalues = []
    steps = 0

    B = A - shift * np.eye(A.shape[0])

    while True:
        y = np.linalg.solve(B, x)

        eigenvalue = (x @ y) / (x @ x)
        eigenvalues.append(eigenvalue)

        if steps > 0 and np.allclose(eigenvalues[-1], eigenvalues[-2], atol=1e-6):
            break

        x = y / np.linalg.norm(y)
        steps += 1
    plt.plot(range(1, steps + 2), eigenvalues, marker="o")
    plt.xlabel("Iteration")
    plt.ylabel("Estimated Eigenvalue")
    plt.title(f"Guess: {original_x}")
    plt.show()
    return eigenvalue, x

u1_guess = np.array([-1, 0.5, 3])
u2_guess = np.array([2, -6, 0.2])
eigenvalue1, eigenvector1 = inverse_power_method(A, u1_guess)
eigenvalue2, eigenvector2 = inverse_power_method(A, u2_guess)
print("Estimated Eigenvalue 1:", eigenvalue1)
print("Estimated Eigenvector 1:", eigenvector1)
print("Estimated Eigenvalue 2:", eigenvalue2)
print("Estimated Eigenvector 2:", eigenvector2)
print("Actual Eigenvalues and Eigenvectors:")
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```





```
Estimated Eigenvalue 1: -1.000003151522974
Estimated Eigenvector 1: [-8.94431020e-01  4.47205938e-01  2.07434890e-05]
Estimated Eigenvalue 2: -0.9999941928438382
Estimated Eigenvector 2: [-8.94421512e-01  4.47224953e-01 -3.68376612e-05]
Actual Eigenvalues and Eigenvectors:
Eigenvalues: [ 6.  1. -1.]
Eigenvectors: [[ 4.61566331e-01  8.94427191e-01  5.18104078e-17]
 [ 5.93442426e-01 -4.47213595e-01 -4.47213595e-01]
 [ 6.59380473e-01  3.01974496e-16  8.94427191e-01]]
```

Question 4: Face Recognition with Eigenfaces (30 points + 10 bonus points)

Goal: Perform face recognition on the *Labeled Faces in the Wild* dataset using PyTorch.

Dataset Information: *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.

Tasks:

- First, perform Principal Component Analysis (PCA) on the image dataset.
- Using PCA, extract the Top k principal components (*eigenvalues*).

- 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.
- **BONUS**: Finally, train a simple PyTorch Neural Network model on the modified image dataset. This trained model will be used for 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.

In [28]:
```
# !!DO NOT EDIT!!
# Loading the dataset directly from the scikit-learn library (can take about 3-5 mi
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
# 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

In [29]:
```
# !!DO NOT EDIT!!
# create target label - target name pairs
targets = [(x,y) for x,y in zip(range(len(np.unique(dataset.target))), dataset.targ
print('The target labels and names are:\n', targets)
```

```
The target labels and names are:
 [(0, np.str_('Colin Powell')), (1, np.str_('Donald Rumsfeld')), (2, np.str_('George
W Bush')), (3, np.str_('Gerhard Schroeder')), (4, np.str_('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, 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 features of 5 images of `X_train` (This result can vary from pc to pc). (10 points)

In [30]:
```
# !!DO NOT EDIT!!
X = dataset.data
y = dataset.target
```

In [ ]:
```
#######
# !!!! YOUR CODE HERE !!!!

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_st

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print(X_train_scaled[: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) Dimensionality reduction :

Use the `PCA` API from `sklearn` to extract the top 100 principal components of the image matrix and fit it on the training dataset.

Visualize some of the top few components as an image (eigenfaces). (10 points)

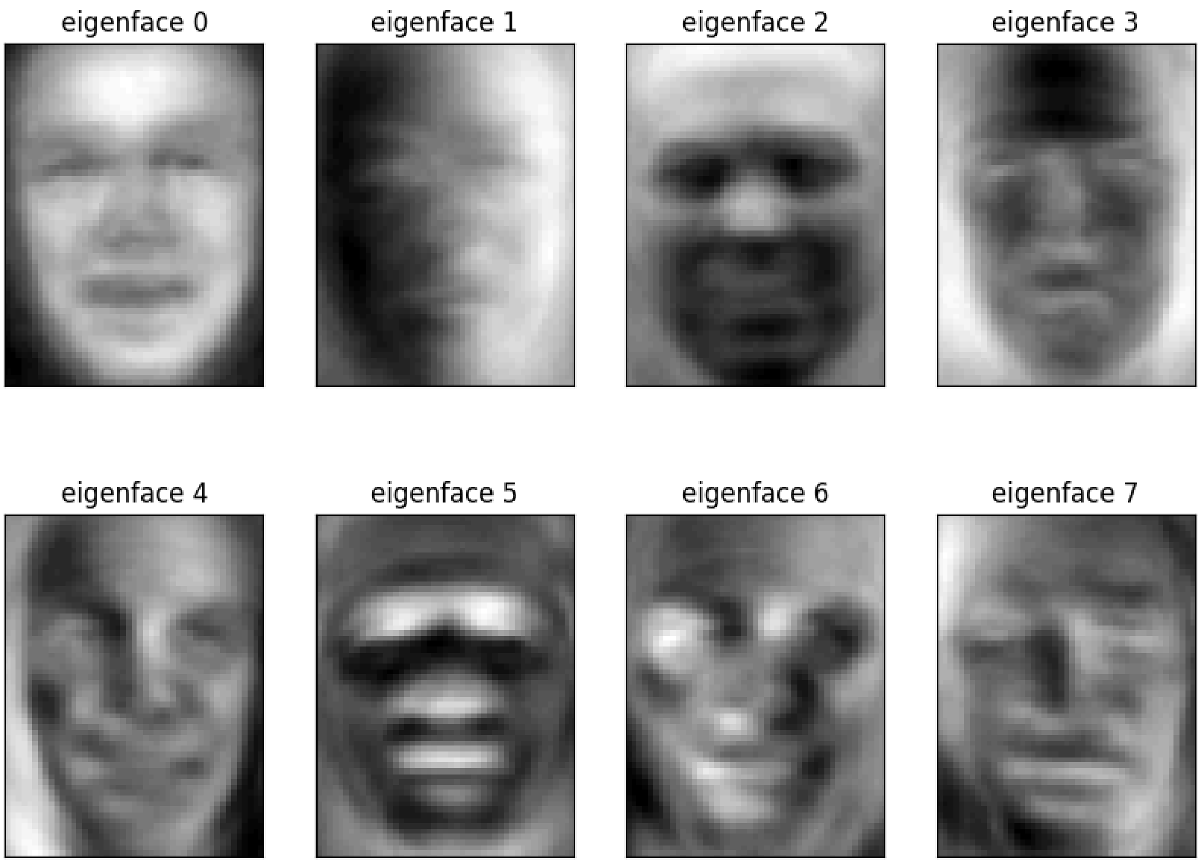
In [36]:
```
#######
# !!!! YOUR CODE HERE !!!!
# initialize PCA API from sklearn with n_components. Also set svd_solver="randomize
```

```
from sklearn.decomposition import PCA
n_components = 100
pca_model = PCA(n_components=n_components, svd_solver="randomized", whiten=True)
pca = pca_model.fit(X_train_scaled)

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

Now we will plot the most representative eigenfaces:

In [37]:
```
# !!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(())
```

In [38]:
```
# !!DO NOT EDIT!!
# get the 100 eigen faces and reshape them to original image size which is 62 x 47
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:

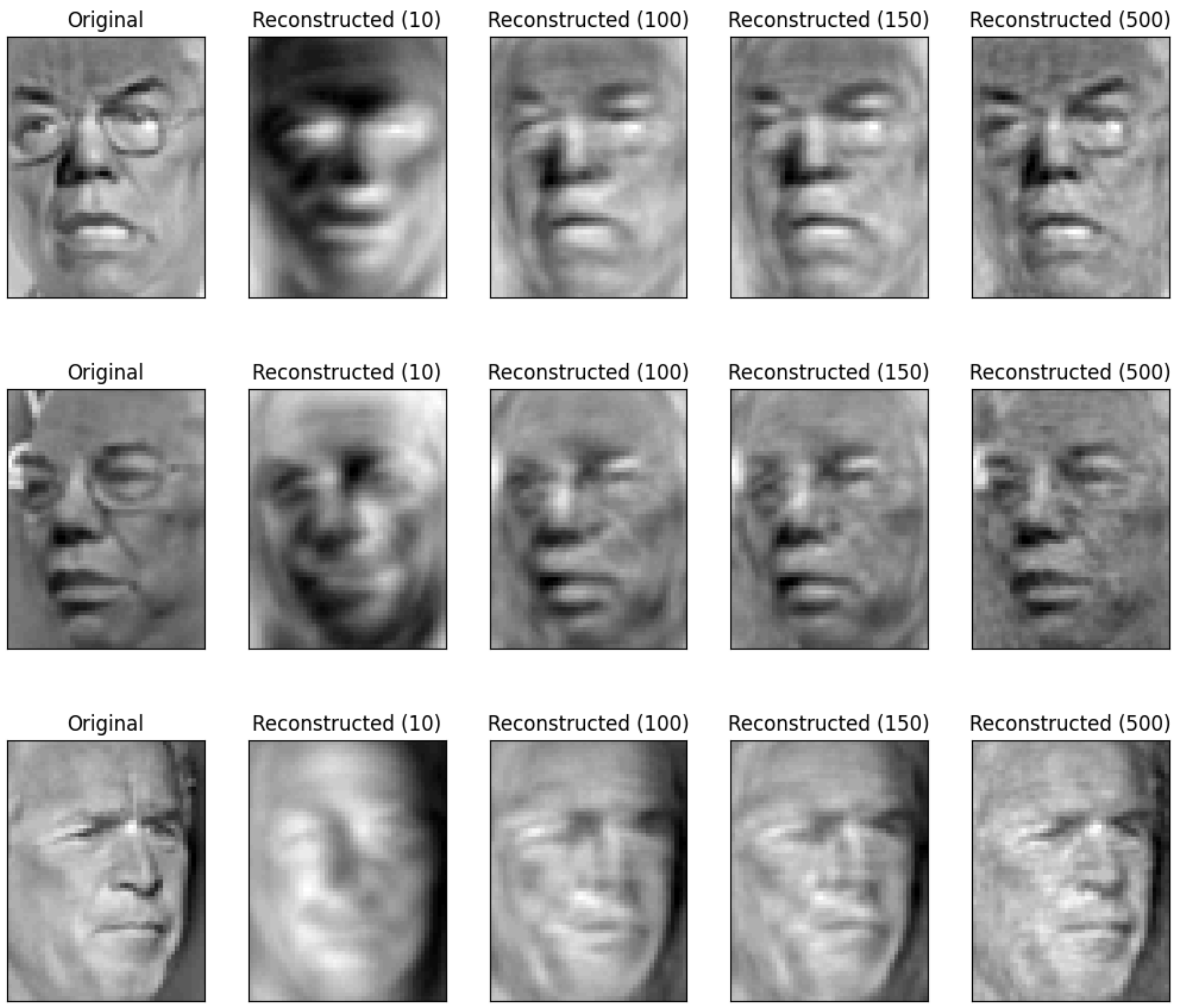
Reconstruct an image from its point projected on the principal component basis.

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 $= [10, 100, 150, 500]$. (10 points)

In [40]:
```
#######
# !!!! YOUR CODE HERE !!!!

n_components_list = [10, 100, 150, 500]
reconstructed_images = []
for n_components in n_components_list:
    pca_model = PCA(n_components=n_components, svd_solver="randomized", whiten=True
    pca = pca_model.fit(X_train_scaled)

    projected = pca.transform(X_test_scaled[:3])
    reconstructed = pca.inverse_transform(projected)
    reconstructed_images.append(reconstructed)
```

```
final_images = []
for i in range(3):
    final_images.append(X_test_scaled[i])
    for j in range(len(n_components_list)):
        final_images.append(reconstructed_images[j][i])

image_titles = []
for i in range(3):
    image_titles.append("Original")
    for n_components in n_components_list:
        image_titles.append(f"Reconstructed ({n_components})")
plot_gallery(final_images, image_titles, height, width, n_row=3, n_col=5)
plt.show()

#######
```



(d) Prediction (Bonus):

Train a neural network classifier in **PyTorch** on the transformed dataset. Complete each of the steps below.

Note: For PyTorch reference see documentation. (10 points)

In [47]:
```
# !!DO NOT EDIT!!
# define imports here
import torch
import torch.nn as nn
```

In [48]:
```
print(torch.cuda.is_available())
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
True
```

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

Move the train and test dataset to torch tensors in order to work with pytorch.

In [53]:
```
#######
# !!!! YOUR CODE HERE !!!!
# 1. project X_train and X_test on orthonormal basis using the PCA API initialized

n_components = 100
pca_model = PCA(n_components=n_components, svd_solver="randomized", whiten=True)
pca = pca_model.fit(X_train_scaled)

X_train_pca = pca.transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

# 2. now convert X_train_pca, X_test_pca, y_train and y_test to torch.tensor. For y

X_train_pca_torch = torch.tensor(X_train_pca, dtype=torch.float32).to(device)
X_test_pca_torch = torch.tensor(X_test_pca, dtype=torch.float32).to(device)
y_train_torch = torch.tensor(y_train, dtype=torch.long).to(device)
y_test_torch = torch.tensor(y_test, dtype=torch.long).to(device)

# output variable names -  X_train_pca_torch, X_test_pca_torch, y_train_torch, y_te
#######
```

In [54]:
```
#######
# !!!! YOUR CODE HERE !!!!
# 3. We will implement a simple multilayer perceptron (MLP) in pytorch with one hid
# 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
    self.fc1 = nn.Linear(100, 1024)
    self.fc2 = nn.Linear(1024, 5)

    # DO: initialize relu activation function
    self.relu = nn.ReLU()

    # DO: initialize LogSoftmax
    self.logsoftmax = nn.LogSoftmax(dim=1)

  def forward(self, x):
    # DO: define the feedforward algorithm of the model and return the final output
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    x = self.logsoftmax(x)
    return x

#######
```

In [55]:

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

model = MLP().to(device)

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

criterion = nn.NLLLoss().to(device)

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

optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

#######

# !!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: 1.660
Epoch: 50, Loss: 0.443
Epoch: 100, Loss: 0.224
Epoch: 150, Loss: 0.136
Epoch: 200, Loss: 0.091
Epoch: 250, Loss: 0.066
Epoch: 300, Loss: 0.050
Epoch: 350, Loss: 0.040
Epoch: 400, Loss: 0.033
Epoch: 450, Loss: 0.027
Epoch: 500, Loss: 0.023
```

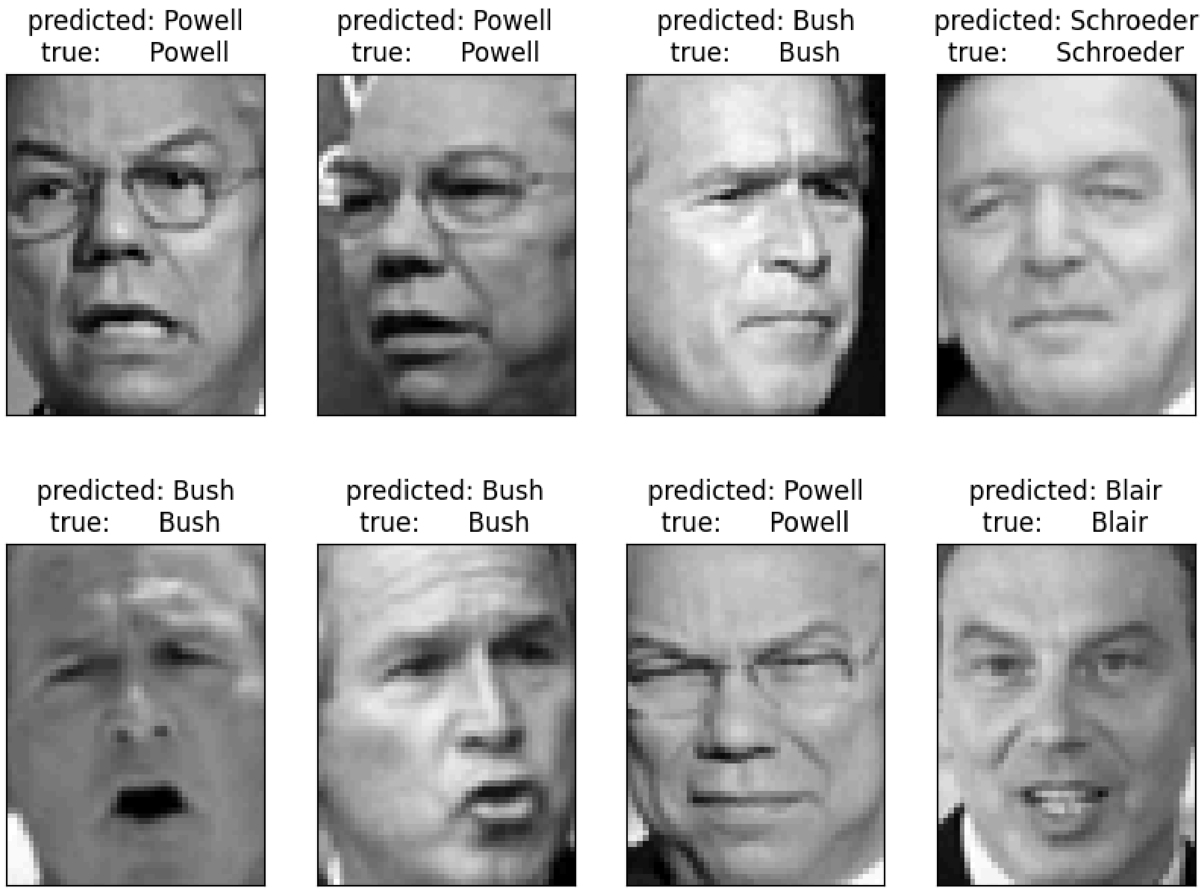
In [60]:

```python
# !!DO NOT EDIT!!
# predict on test data
predictions = model(X_test_pca_torch).to(device) # gives softmax logits
y_pred = torch.argmax(predictions, dim=1).cpu().numpy() # get the labels from prdic
```

In [61]:

```python
# !!DO NOT EDIT!!
# here, we will print the multi-label classification report: precision, recall, f1-
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      0.89      0.90        64
  Donald Rumsfeld       0.83      0.78      0.81        32
    George W Bush       0.92      0.95      0.93       127
Gerhard Schroeder       0.96      0.90      0.93        29
      Tony Blair        0.85      0.85      0.85        33

         accuracy                           0.90       285
        macro avg       0.89      0.87      0.88       285
     weighted avg       0.90      0.90      0.90       285
```


predicted: Powell
true: Powell
predicted: Powell
true: Powell
predicted: Bush
true: Bush
predicted: Schroeder
true: Schroeder
predicted: Bush
true: Bush
predicted: Bush
true: Bush
predicted: Powell
true: Powell
predicted: Blair
true: Blair