Lab 8 - Principal Component Analysis

Name and netid: Matthew Wei (mswei2)

Due: Nov 10, 2023 at 11:59 PM

Logistics:

Type *Markdown* and LaTeX: α^2

What You Will Need To Know For This Lab:

- Eigendecomposition
- Singular Value Decomposition
- Principal Component Analysis

Preamble (Don't change this):

In [1]:
```
from __future__ import division
%pylab inline
import numpy as np
from sklearn import neighbors
from mpl_toolkits.mplot3d import Axes3D
import random
from sklearn.decomposition import PCA
from PIL import Image
from sklearn.cluster import KMeans
import scipy.spatial.distance as dist
from matplotlib.colors import ListedColormap
```

```
%pylab is deprecated, use %matplotlib inline and import the required librarie
s.
Populating the interactive namespace from numpy and matplotlib

C:\Users\matt3\AppData\Local\Programs\Python\Python310\lib\site-packages\scip
y\__init__.py:146: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is requi
red for this version of SciPy (detected version 1.26.1
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

Enable Interactive Plots

In [28]:
```
enable_interactive=False # If you want to rotate plots, set this to True.
# When submitting your notebook, enable_interactive=False and run the whole no
# The interactive stuff can be a bit glitchy, so if you're having trouble, tur
if enable_interactive:
    # These packages allow us to rotate plots and what not.
    from IPython.display import display
    from IPython.html.widgets import interact
```

Problem 1: Visualizing Principal Components (50 points)

In this problem, you will be implementing PCA, visualizing the principal components and using it to perform dimensionality reduction.

Do not use a pre-written implementation of PCA for this problem (e.g. sklearn.decomposition.PCA). You should assume that the input data has been appropriately pre-processed to have zero-mean features.

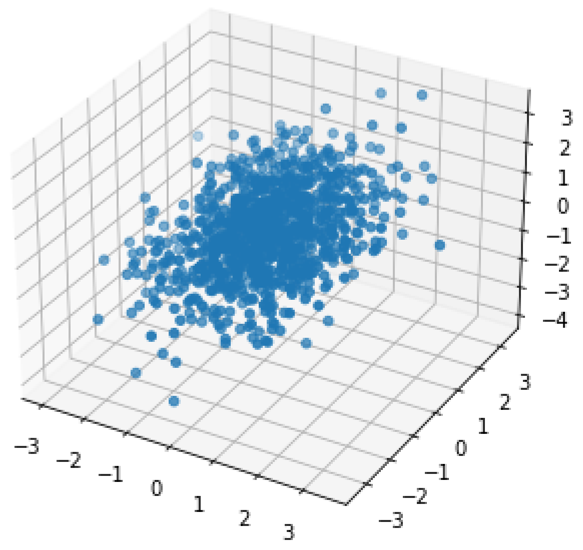
In [29]:
```
# We will generate some data.
numpy.random.seed(seed=2232017)
true_cov = np.array([[1,.2,.3],[.2,1,.6],[.3,.6,1]]) #This is the true covaria
                                                     #of the data. Do not use
data=(np.random.randn(1000,3)).dot(np.linalg.cholesky(true_cov).T)
```

First, we visualize the data using a 3D scatterplot.

Our data is stored in a variable called `data` where each row is a feature vector (with three features).

In [30]:
```
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(data[:,0],data[:,1],data[:,2])
if enable_interactive:
    @interact(elev=(-90, 90), azim=(0, 360))
    def view(elev, azim):
        ax.view_init(elev, azim)
        display(ax.figure)
```

```
C:\Users\matt3\AppData\Local\Temp\ipykernel_46452\1059038685.py:2: Matplotlib
DeprecationWarning: Axes3D(fig) adding itself to the figure is deprecated sin
ce 3.4. Pass the keyword argument auto_add_to_figure=False and use fig.add_ax
es(ax) to suppress this warning. The default value of auto_add_to_figure will
change to False in mpl3.5 and True values will no longer work in 3.6.  This i
s consistent with other Axes classes.
  ax = Axes3D(fig)
```



Write a function which implements PCA via the eigendecomposition. **(10 points)**

You will be given as input:

- A (N, d) numpy array of data (with each row as a feature vector)

Your function should return a tuple consisting of the PCA transformation matrix (which is (d, d)), and a vector consisting of the amount of variance explained in the data by each PCA feature. Note that the PCA features are ordered in decreasing amount of variance explained, by convention.

Hints:

- The function numpy.linalg.eigh (http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.eigh.html) will be useful. Note that it returns its eigenvalues in *ascending* order. `numpy.fliplr` or similar may be useful as well.
- You can calculate the covariance matrix of the data by multiplying the data matrix with its transpose in the appropriate order, and scaling it.

- Do not use numpy.cov -- we are assuming the data has zero mean beforehand, so the number of degrees of freedom is different (since the covariance estimate knows the mean in our case)

In [43]:
```
def pcaeig(data):
    cov = (1/data.shape[0])*(data.T).dot(data)
    s, W = np.linalg.eigh(cov)
    W = np.fliplr(W).T
    s = s[::-1]
    return (W,s)
```

Now, run PCA on your data. Store your PCA transformation in a variable called `W` , and the amount of variance explained by each PCA feature in a variable called `s` . Print out the principal components (i.e. the rows of `W`) along with the corresponding amount of variance explained. **(5 points)**

In [44]:
```
# Now, run PCA on your data. The PCA transformation is stored in W, while the
W,s=pcaeig(data)

# Print out the principal components + the amount of variance they explain
for i in range(W.shape[1]):
    print(i+1,end='')
    print('-th principal component: ', W[i,:], "\t Variance:",s[i])
```

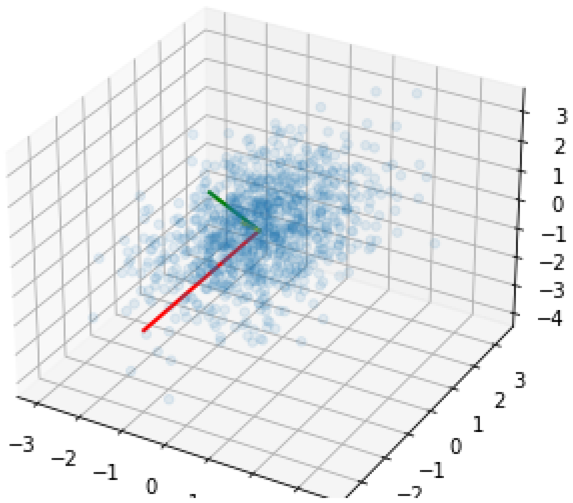
```
1-th principal component:  [-0.37919732 -0.6615397  -0.64697343]        Vari
ance: 1.7740242208390087
2-th principal component:  [-0.91750747  0.35945511  0.17021185]        Vari
ance: 0.8145914895124162
3-th principal component:  [-0.119956   -0.65814683  0.74327203]        Vari
ance: 0.386639067521256
```

We can visualize the principal components on top of our data. The first principal component is in red, and captures the most variance. The second principal component is in green, while the last principal component is in yellow.

We generated our data from am *elliptical distribution*, so it should be easy to visualize these components as the axes of the data (which looks like an ellipsoid).

In [33]:
```
figb = plt.figure()
axb = Axes3D(figb)
axb.scatter(data[:,0],data[:,1],data[:,2],alpha=0.1)
c=['r-','g-','y-']
for var, pc,color in zip(s, W,c):
    axb.plot([0, 2*var*pc[0]], [0, 2*var*pc[1]], [0, 2*var*pc[2]], color, lw=2
if enable_interactive:
    @interact(elev=(-90, 90), azim=(0, 360))
    def view(elev, azim):
        axb.view_init(elev, azim)
        display(axb.figure)
```

```
figure will change to False in mpl3.5 and True values will no longer work
in 3.6.  This is consistent with other Axes classes.
  axb = Axes3D(figb)
```



If done correctly, the red line should be longer than the green line which should be longer than the yellow line.

Now, you will implement functions to generate PCA features.

Write a function which implements dimension reduction via PCA. It takes in three inputs:

- A (N, d) numpy array, `data`, with each row as a feature vector
- A (d, d) numpy array, `W`, the PCA transformation matrix (e.g. generated from `pcaeig` or `pcasvd`)
- A number `k`, which is the number of PCA features to retain

It should return a (N, k) numpy array, where the i-th row contains the PCA features corresponding to the i-th input feature vector. **(5 points)**

In [78]:
```
def pcadimreduce(data,W,k):
    W = W[0:k]
    pca = np.zeros((data.shape[0], W.shape[0]))
    for i in range(data.shape[0]):
        pca[i] = np.dot(W, data[i])
    return pca
```

Write a function which reconstructs the original features from the PCA features. It takes in three inputs:

- A (N, k) numpy array, `pcadata` , with each row as a PCA feature vector (e.g. generated from `pcadimreduce`)
- A (d, d) numpy array, `W` , the PCA transformation matrix (e.g. generated from `pcaeig` or `pcasvd`)
- A number `k` , which is the number of PCA features

It should return a (N, d) numpy array, where the i-th row contains the reconstruction of the original i-th input feature vector (in `data`) based on the PCA features contained in `pcadata` . **(5 points)**

In [79]:
```
def pcareconstruct(pcadata,W,k):
    W = W[0:k].T
    pca = np.zeros((pcadata.shape[0], W.shape[0]))
    for i in range(pcadata.shape[0]):
        pca[i] = np.dot(W, pcadata[i])
    return pca
```

As a sanity check, if you take $k = 3$, perform dimensionality reduction then reconstruction, you should get the original data back:

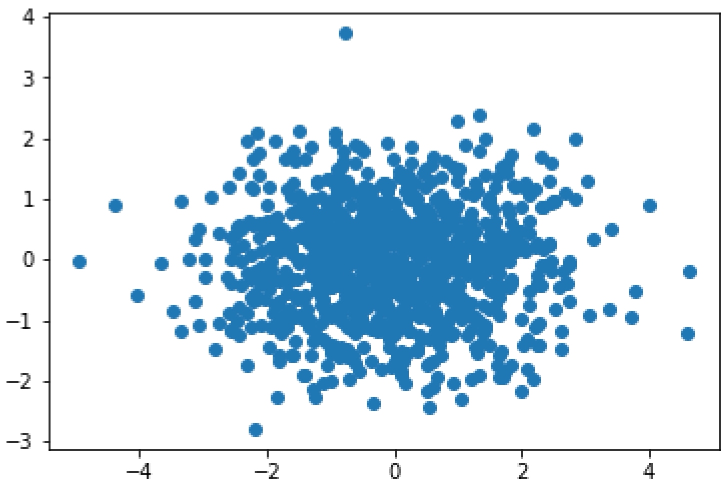
In [80]:
```
# Reconstructed data using all the principal components
reduced_data=pcadimreduce(data,W,3)
reconstructed_data=pcareconstruct(reduced_data,W,3)

print ("This should be small:",np.max(np.abs(data-reconstructed_data)))
```

```
This should be small: 2.220446049250313e-15
```

One use of PCA is to help visualize data. The 3-D plots above are a bit hard to read on a 2-D computer screen or when printed out.

Use PCA to to reduce the data to two dimensions. Visualize the first two PCA features with a scatter plot. Also, construct an approximation of the original features using the first two principal components into a (N, d) array called `reconstructed_data` .**(10 points)**

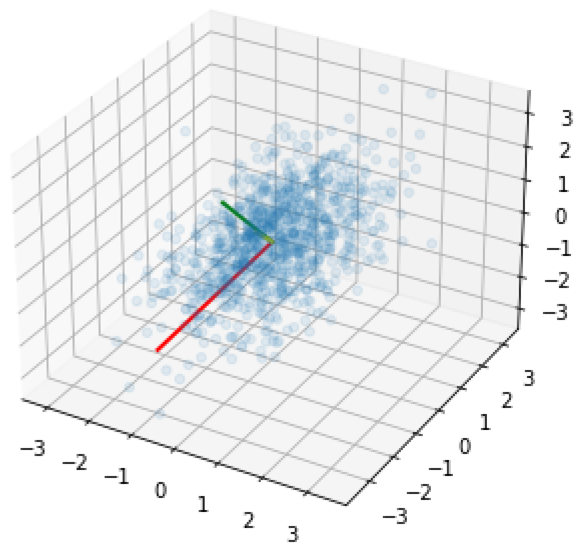
In [81]:
```
#Put your code here
reduced_data=pcadimreduce(data,W,2)
scatter(reduced_data[:,0],reduced_data[:,1])
reconstructed_data=pcareconstruct(reduced_data,W,2)
```



We can now visualize the data using two principal components in the original feature space.

In [82]:
```
figc = plt.figure()
axc = Axes3D(figc)
axc.scatter(reconstructed_data[:,0],reconstructed_data[:,1],reconstructed_data
c=['r-','g-','y-']
for var, pc,color in zip(s, W,c):
    axc.plot([0, 2*var*pc[0]], [0, 2*var*pc[1]], [0, 2*var*pc[2]], color, lw=2

if enable_interactive:
    @interact(elev=(-90, 90), azim=(0, 360))
    def view(elev, azim):
        axc.view_init(elev, azim)
        display(axc.figure)
```

```
C:\Users\matt3\AppData\Local\Temp\ipykernel_46452\534669976.py:2: MatplotlibD
eprecationWarning: Axes3D(fig) adding itself to the figure is deprecated sinc
e 3.4. Pass the keyword argument auto_add_to_figure=False and use fig.add_axe
s(ax) to suppress this warning. The default value of auto_add_to_figure will
change to False in mpl3.5 and True values will no longer work in 3.6.  This i
s consistent with other Axes classes.
  axc = Axes3D(figc)
```



If done correctly, you should see no component of the data along the third principal direction, and the data should lie in a plane. This may be easier to see with the Interactive Mode on.

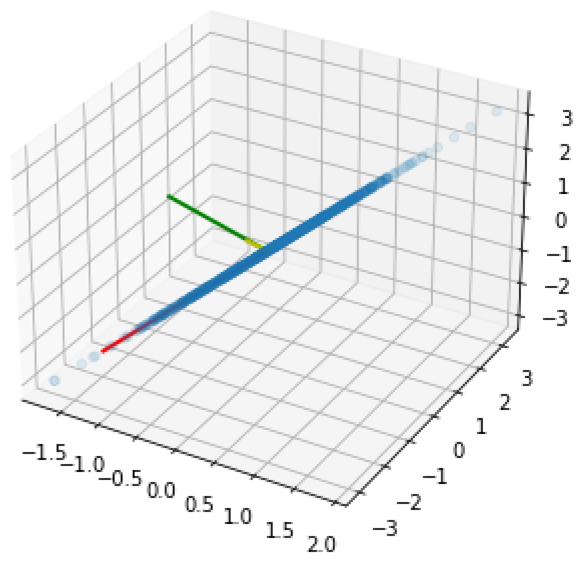
Use PCA to reduce the data to one dimension and store the one dimensional PCA feature in `reduced_data_1` . Construct an approximation of the original features using the first principal component into a (N, d) array called `reconstructed_data_1` . **(5 points)**

In [83]:
```
#Put your code here
reduced_data_1 = pcadimreduce(data,W,1)
reconstructed_data_1 = pcareconstruct(reduced_data_1,W,1)
```

We can now visualize this in the original feature space.

In [84]:
```
figd = plt.figure()
axd = Axes3D(figd)
axd.scatter(reconstructed_data_1[:,0],reconstructed_data_1[:,1],reconstructed_
c=['r-','g-','y-']
for var, pc,color in zip(s, W,c):
    axd.plot([0, 2*var*pc[0]], [0, 2*var*pc[1]], [0, 2*var*pc[2]], color, lw=2

if enable_interactive:
    @interact(elev=(-90, 90), azim=(0, 360))
    def view(elev, azim):
        axd.view_init(elev, azim)
        display(axd.figure)
```

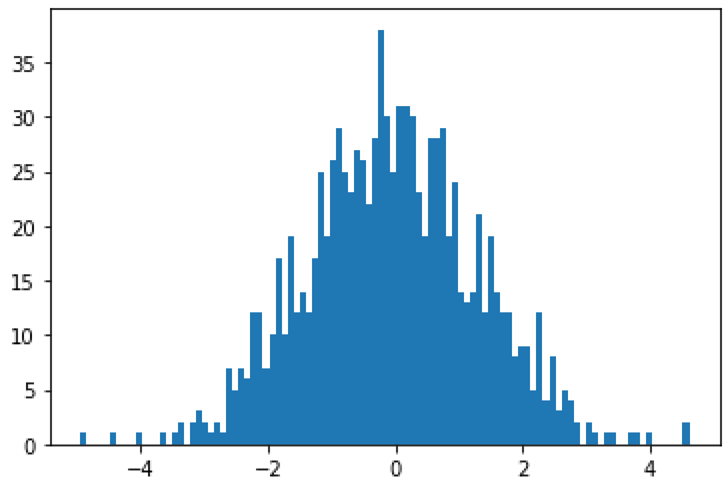
```
C:\Users\matt3\AppData\Local\Temp\ipykernel_46452\496516323.py:2: MatplotlibD
eprecationWarning: Axes3D(fig) adding itself to the figure is deprecated sinc
e 3.4. Pass the keyword argument auto_add_to_figure=False and use fig.add_axe
s(ax) to suppress this warning. The default value of auto_add_to_figure will
change to False in mpl3.5 and True values will no longer work in 3.6.  This i
s consistent with other Axes classes.
  axd = Axes3D(figd)
```



If done correctly, you should see no component of the data along the second and third principal direction, and the data should lie along a line. This may be easier with the Interactive Mode on.

We can also visualize the PCA feature as a histogram:

In [85]:
```
n, bins, patches = hist(reduced_data_1,100)
```



Finally, you will implement PCA via the SVD. **(5 points)**

You will be given as input:

- A (N, d) numpy array of data (with each row as a feature vector)

Your function should return a tuple consisting of the PCA transformation matrix, and a vector consisting of the amount of variance explained in the data by each PCA feature. Note that the PCA features are ordered in decreasing amount of variance explained.

Hints:

- The function numpy.linalg.svd (http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.svd.html) will be useful. Use the full SVD (default).
- Be careful with how the SVD is returned in `numpy.linalg.svd`

In [86]:
```
def pcasvd(data):
    SVD = linalg.svd(data)
    V = SVD[2]
    D = SVD[1]**2/data.shape[0]
    return V, D
```

If your PCA implementation via the SVD is correct (and your Eigendecomposition implementation is correct), principal components should match between the SVD and PCA implementations (up to sign, i.e. the i-th principal component may be the negative of the i-th principal component from the eigendecomposition approach).

Verify this by printing out the principal components and the corresponding amount of variance explained. You will not get any credit if the principal components (up to sign) and variances do not match the eigendecomposition. **(5 points)**

In [87]:
```
# Now, run PCA on your data. The PCA transformation is stored in Wsvd, while t
Wsvd,ssvd=pcasvd(data)

# Print out the principal components + the amount of variance they explain
for i in range(Wsvd.shape[1]):
    print(i+1,end='')
    print('-th principal component: ', W[i,:], "\t Variance:",s[i])
```

```
1-th principal component:  [-0.37919732 -0.6615397  -0.64697343]        Vari
ance: 1.7740242208390087
2-th principal component:  [-0.91750747  0.35945511  0.17021185]        Vari
ance: 0.8145914895124162
3-th principal component:  [-0.119956   -0.65814683  0.74327203]        Vari
ance: 0.386639067521256
```

Problem 2: PCA for Data Compression (30 points)

In [ ]:
```

```

In this part of the lab, we will look at eigenfaces for compression using the Olivetti faces dataset (http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html).

In [89]:
```
# First, we load the Olivetti dataset
from sklearn.datasets import fetch_olivetti_faces


oli = fetch_olivetti_faces()
# Height and Width of Images are in h,w. You will need to reshape them to this
h=64
w=64
X = oli.data

X_t=X[-1]
X=X[:-1]

#This centering is unnecessary. it just makes the pictures a bit more readable

X_m=np.mean(X,axis=0)
X=X-X_m # center them
X_t=X_t-X_m

# The data set is in X. You will compress the image X_t.
```

We can visualize the Olivetti Faces:

We will be making use of Scikit-Learn's PCA (http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html) functionality.

Three functions will be useful for this problem :

- PCA.fit : Finds the requested number of principal components.
- PCA.transform : Apply dimensionality reduction (returns the PCA features)
- PCA.inverse_transform : Go from PCA features to the original features (Useful for visualizing)

You will also find the following useful:

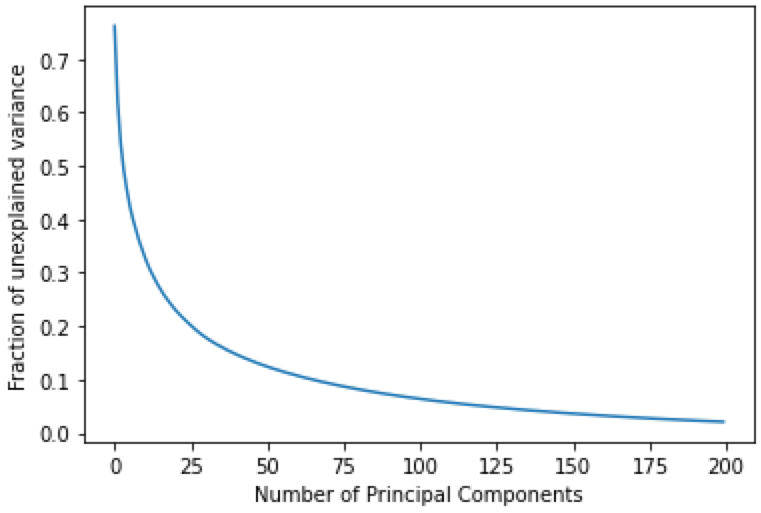
- PCA.explained_variance_ratio_ : Percentage of variance explained by each of the principal components

Plot the fraction of **unexplained** variance on `X` by PCA retaining the first k principal components, where $k = 1, \ldots, 200$. Note that this is a scree plot (normalized by the total variance).

`numpy.cumsum` may be useful for this. **(10 points)**

In [92]:
```
# Put your code here
for k in range(1, 201):
    model = PCA(n_components = k)
    model.fit(X)
unexp = 1 - np.cumsum(model.explained_variance_ratio_)
figure()
plot(unexp)
xlabel('Number of Principal Components')
ylabel('Fraction of unexplained variance')
```

Out[92]:
```
Text(0, 0.5, 'Fraction of unexplained variance')
```



Based on the Scree plot, propose a reasonable number of principal components to keep, in order to perform dimensionality reduction. Justify your choice. **(5 points)**

There is a range of correct answers (but you need to justify yours!).

Type *Markdown* and LaTeX: α^2

A reasonable number to choose is 125 components. Before 125, the changes are drastic. Around 125 is where it starts to flatten out or the change of unexplained variances decreases. At 125, it looks like it is around 95 and at 200, it also looks like it is around 95.

Visualize the first 5 principal components as well as the 30th, 50th and 100th principal components, which are called *eigenfaces* in this context. Assuming your PCA object is called `pca` , the eigenfaces are contained in `pca.components_` , where each row is a principal component.

The following code may be useful:

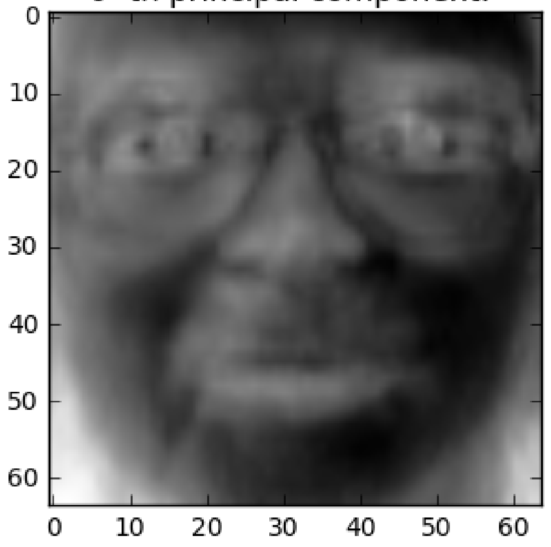
```
figure()
imshow( image , cmap = cm.Greys_r)
```

where image is the appropriately reshaped principal component (to `h` rows and `w` columns). What can you say about later eigenfaces compared to earlier ones? **(5 points)**

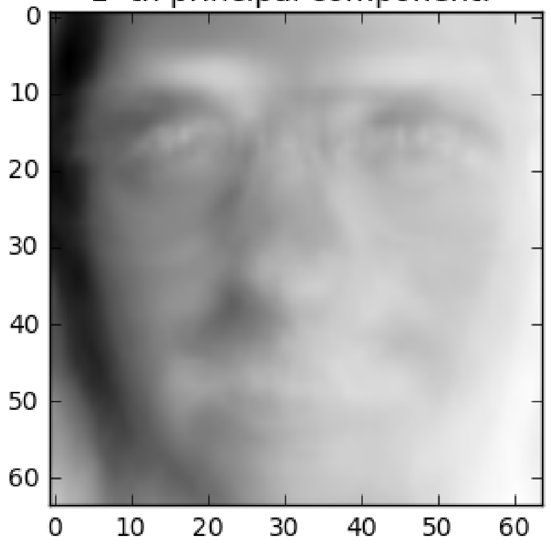
The later eigenfaces seem to be more contrast and able to detect more features. The edges get more defined and it is able to separate the features more.

In [20]:
```
fn=np.asarray([0,1,2,3,4,29,49,99])

figure(figsize=(8,16))
for i in range(fn.size):
    subplot(4,2,i+1)
    title("{} -th principal component: ".format(fn[i]))
    imshow((pca.components_[fn[i]]).reshape((h,w)),cmap=cm.Greys_r)
```

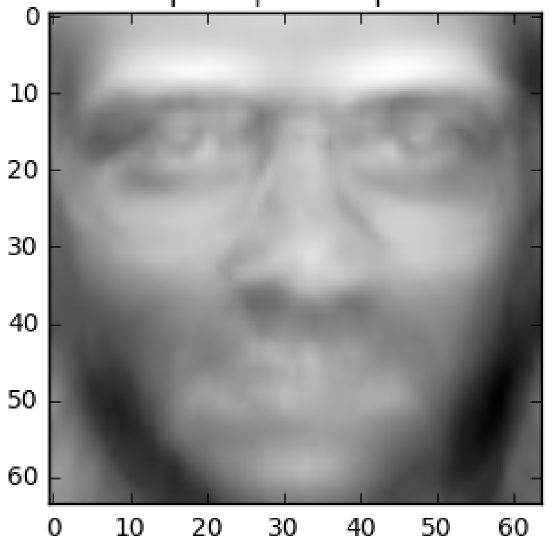
0 -th principal component:



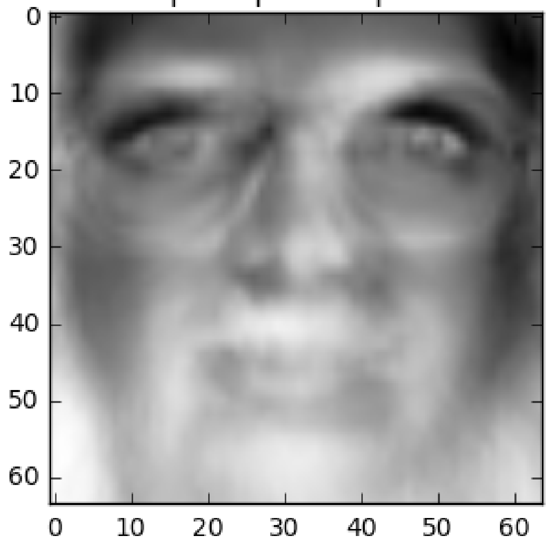
1 -th principal component:



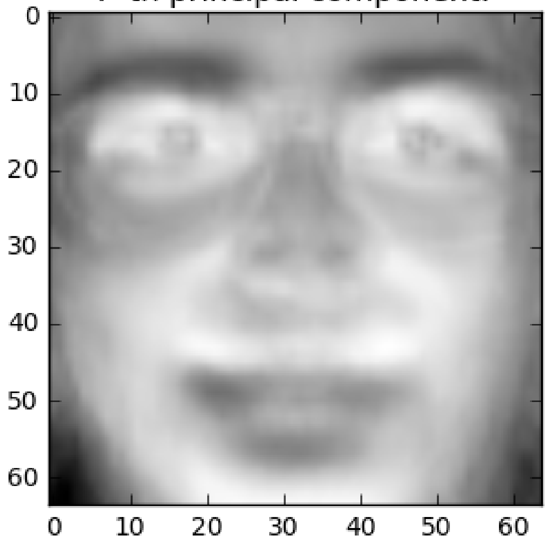
2 -th principal component:



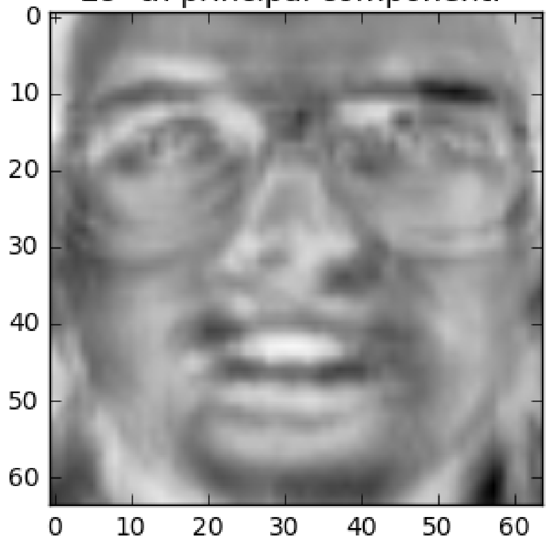
3 -th principal component:



4 -th principal component:

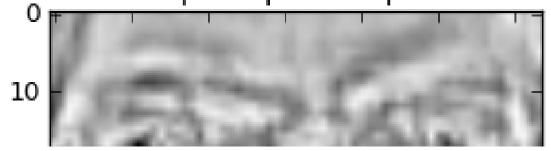


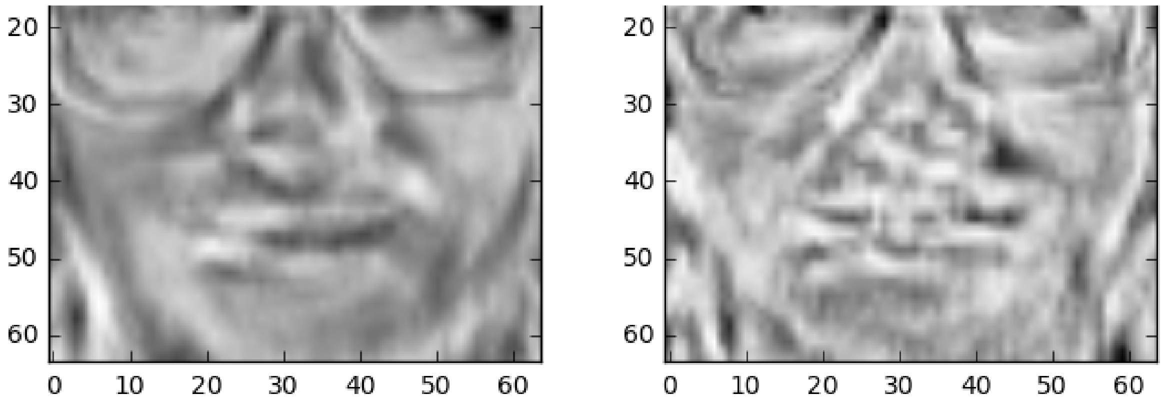
29 -th principal component:



49 -th principal component:

99 -th principal component:



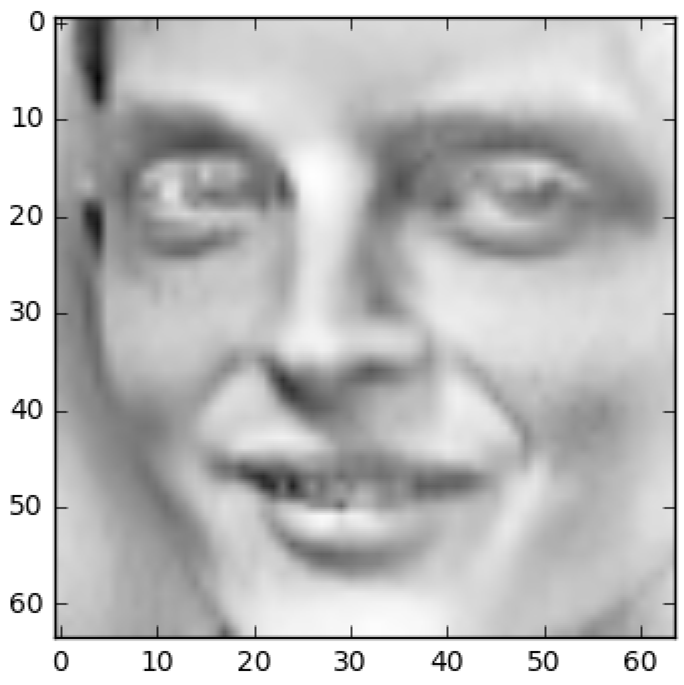


Later eigenfaces capture more detail as compared to earlier ones (e.g. they're specific to some guy).

Now, you will compress an image, `X_t` , using PCA.

In [21]:
```
# This is what X_t looks like:
imshow((X_t).reshape((h,w)),cmap=cm.Greys_r)
```

Out[21]:
```
<matplotlib.image.AxesImage at 0xcf729b0>
```



Display the image in `X_t` 's approximation using the first `i` principal components (learned from `X`) where i=1,10,20,...,100 (i.e. in increments of 10), then 120,140,160,180,200 (i.e. in increments of 20).

Do this by the following procedure:

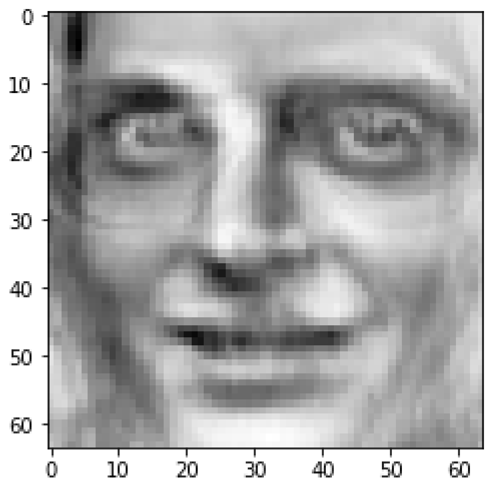
1. Determine the PCA transformation (i.e. fit) on `X` .
2. Transform `X_t` to the PCA features determined by `X` .
3. Retain the first `i` PCA features of the transformed `X_t` (set the others to zero).

4. Transform the result of step 3 back to the original feature space.

(5 points)

In [97]:
```
# Put your code here.
def reconst(pc_comp):
    pca = PCA(n_components = i)
    pca.fit(X)
    image = np.reshape(pca.inverse_transform(pca.transform(X_t.reshape(1,-1)))
    return image

image = reconst(1)
figure(1)
suptitle('k = 1');
imshow( image, cmap = cm.Greys_r);

for i in range(10,101,10):
    image = reconst(i)
    figure(i)
    suptitle("k = " + str(i));
    imshow( image, cmap = cm.Greys_r);

for i in range(120, 201, 20):
    image = reconst(i)
    figure(i)
    suptitle("k = " + str(i));
    imshow( image, cmap = cm.Greys_r);
```

k = 120



k = 140

How many principal components would you keep to compress the image? Why? (You may be qualitative or quantitative) **(5 points)**

I would keep 120 principal components because the jump from 100 to 120 is the last big jump in noticable features. From 120 onwards, not much change in the image can be detected qualitativly. It also is similar to my answer above with choosing 125 as that is where the change