EECS531-A2-axs1202

April 3, 2020

0.1 EECS 531 Assignment 2

0.2 Anna Sedlackova axs1202

[207]:
```
#all the import statemenets
import numpy as np
import pandas as pd
import sys
import cv2
import skimage
import scipy
from scipy import fftpack
from scipy import ndimage
from scipy.signal import convolve
from PIL import Image
import matplotlib.pyplot as plt
from math import sqrt
from skimage.util import random_noise
from IPython.core.pylabtools import figsize
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA, FastICA
from sklearn.datasets import fetch_olivetti_faces

np.set_printoptions(threshold=sys.maxsize)
```

[208]:
```
# array storing the official coordinates of an image for exercise 1
official_coordinates = \
[[109, 137],
 [109, 138],
 [110, 136],
 [110, 137],
 [110, 138],
 [110, 139],
 [111, 136],
 [111, 137],
 [111, 138],
 [111, 139],
```

```
[112, 137],
[112, 138],
[167, 217],
[167, 218],
[168, 216],
[168, 217],
[168, 218],
[168, 219],
[169, 216],
[169, 217],
[169, 218],
[169, 219],
[170, 216],
[170, 217],
[170, 218],
[218, 222],
[218, 223],
[218, 224],
[219, 221],
[219, 222],
[219, 223],
[219, 224],
[220, 221],
[220, 222],
[220, 223],
[220, 224],
[221, 222],
[221, 223],
[221, 224],
[319, 183],
[320, 182],
[320, 183],
[320, 184],
[321, 183],
[321, 184],
[377, 344],
[377, 345],
[377, 346],
[378, 344],
[378, 345],
[378, 346],
[378, 347],
[379, 344],
[379, 345],
[379, 346],
[399, 255],
[399, 256],
```

```
 [399, 257],
 [400, 255],
 [400, 256]]

official_coordinates = np.array(official_coordinates)
```

0.2.1 Exercise 1

1.1 I have re-implemented my code from assignment 1. I have decided to implement a star detector - a kernel that detects round objects and is set to a PSF (point spread function) of the data. In terms of design, I have reused most of my code from the previous homework and added thresholding. Whenever the sum of the result of multiplying the kernel with a patch of the image is higher than the threshold we replace that patch of the image and append those coordinates to a list that is returned. I return both the list of coordinates as well as the new image.

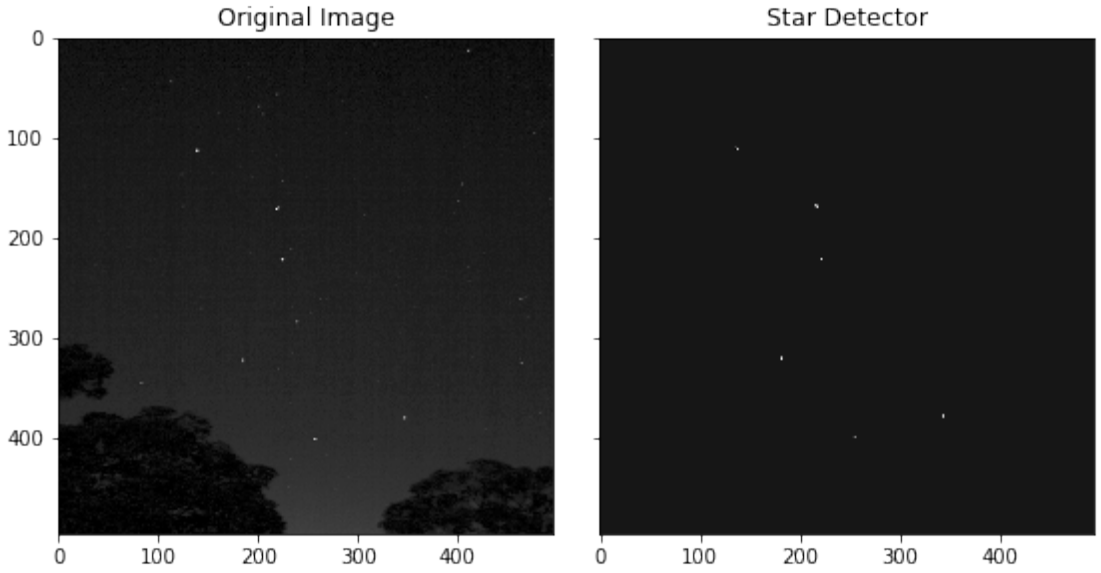
[209]:
```
def feature_detector(kernel, image_array, treshold):
    """
    Applies a kernel to an image array.

    Parameters:
    kernel (list of lists): 2D kernel that modifies the image
    image_array (list of lists): input image as an 2D grayscale
    treshold (int): only returns the sum if higher than the treshold value

    Returns:
    int: modified image array, convolved with the kernel
    list: list of coordinates that were above the treshold
    """
    # Load an image based on the given path
    # image = Image.open(image_path).convert('L')
    image_array = np.asarray(image_array)

    coordinates = []

    kernel = np.asarray(kernel)

    #make a new image array
    offset = len(kernel) // 2

    new_image_array = np.zeros(shape = (image_array.shape[0] - 2 * offset,
                                        image_array.shape[1] - 2 * offset))


    for i in range(offset, image_array.shape[0] - offset):
            for j in range(offset, image_array.shape[1] - offset):
```

```
                #Place the kernel anchor on top of a determined pixel,
                #with the rest of the kernel overlaying the corresponding
                #local pixels in the image
                patch = image_array[i - offset : i - offset + kernel.shape[0],
                                    j - offset : j - offset + kernel.shape[0]]

                #Multiply the kernel coefficients by the corresponding image
                #pixel values and sum the result
                product = np.multiply(patch, kernel)
                sum = np.sum(product)

                #Place the result to the location of the anchor
                new_image_array[i - offset][j - offset] = max(sum, treshold)

                if (sum > treshold):
                    local_coordinates = [i - offset, j - offset]
                    coordinates.append(local_coordinates)

    coordinates = np.array(coordinates)
    new_image = new_image_array.astype('uint8')

    return new_image, coordinates
```

[210]:
```
def display_detector(kernel, image_path, treshold):
    """
    Applies a linear filter to an array.

    Parameters:
    kernel (list of lists): 2D kernel that modifies the image
    image_array (list of lists): input image as an 2D grayscale

    Returns:
    int: modified image array, convolved with the kernel
    """
    # Load an image based on the given path

    image = Image.open(image_path).convert('L')
    image = image.resize((500, 500), 1)
    image_array = np.array(image)

    new_image = feature_detector(kernel, image_array, treshold)[0]

    # display results
    fig, ax = plt.subplots(ncols=2, sharex=True, sharey=True,
                           figsize=(8, 4))

    ax[0].imshow(image_array, cmap=plt.cm.gray)
```

```
    ax[0].set_title('Original Image')

    ax[1].imshow(new_image, cmap=plt.cm.gray)
    ax[1].set_title('Star Detector')

    plt.tight_layout()
    plt.show()
```

[211]:
```
kernel =  [[1., 2., 3., 2., 1.],
           [2., 3., 5., 3., 2.],
           [3., 5., 8., 5., 3.],
           [2., 3., 5., 3., 2.],
           [1., 2., 3., 2., 1.]]

treshold = 5400

display_detector(kernel,'images/stars2.jpg', treshold)
```

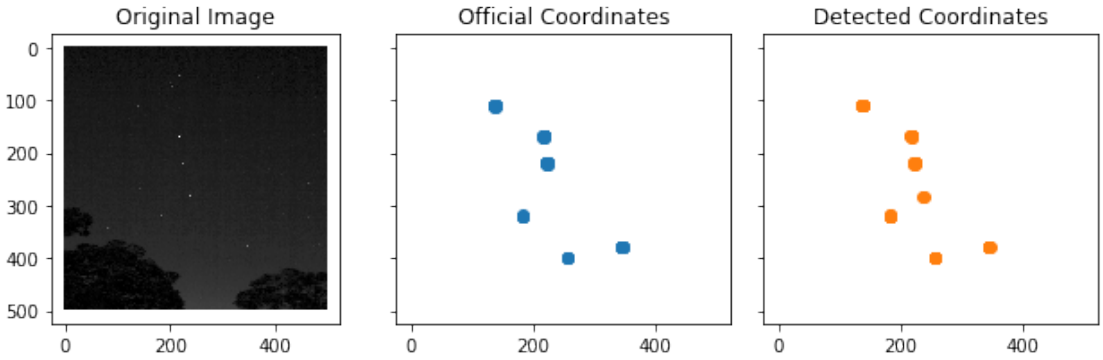


1.2 I have generated the true feature locations manually. The results are generated on two separate plots - one with the true features and one with the detected ones.

[212]:
```
def get_coordinates(kernel, image_path, treshold):
    """
    Retrieves and graphs the coordinates

    Parameters:
    kernel (list of lists): 2D kernel that modifies the image
```

```
    image_path (string): link to an input image
    treshold (int): only returns the sum if higher than
    the treshold value
    """
    # Load an image based on the given path

    image = Image.open(image_path).convert('L')
    image = image.resize((500, 500), 1)
    image_array = np.asarray(image)

    coordinates = feature_detector(kernel, image_array, treshold)[1]

    y, x = official_coordinates.T
    y2, x2 = coordinates.T

    # display results
    fig, ax = plt.subplots(ncols=3, sharex=True, sharey=True,
                           figsize=(9, 3))

    ax[0].imshow(image_array, cmap=plt.cm.gray)
    ax[0].set_title('Original Image')

    ax[1].scatter(x,y)
    ax[1].set_title('Official Coordinates')

    ax[2].scatter(x2,y2, c='#ff7f0e')
    ax[2].set_title('Detected Coordinates')

    plt.tight_layout()
    plt.show()
```

[213]:
```
kernel =  [[1., 2., 3., 2., 1.],
           [2., 3., 5., 3., 2.],
           [3., 5., 8., 5., 3.],
           [2., 3., 5., 3., 2.],
           [1., 2., 3., 2., 1.]]

treshold = 5400

get_coordinates(kernel,'images/stars2.jpg', treshold)
```



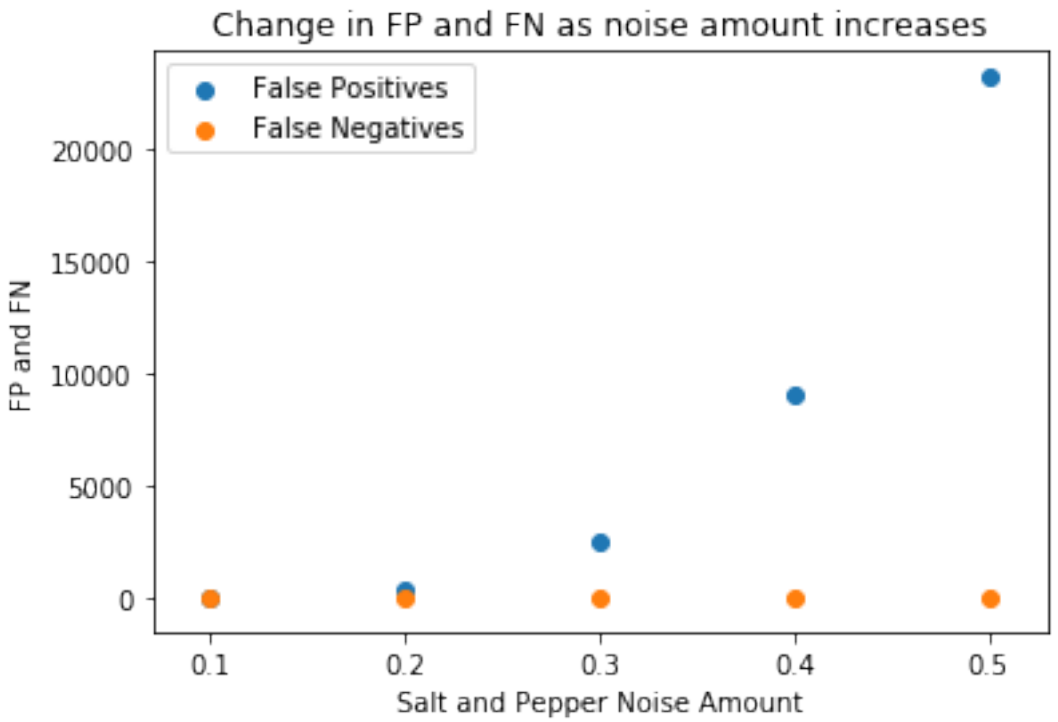
As can be seen in the plots above, the function works fairly well in detecting the star objects in the image.

1.3 I have decided to add increasing amounts of noise until 0.5 using the salt and pepper noise function.

[214]:
```
def graph_noise(kernel, image_path, treshold, noise_array):
    """
    Applies a varying amount of noise to an images and graphs them.

    Parameters:
    kernel (list of lists): 2D kernel that modifies the image
    image_path (string): link to an input image
    treshold (int): only returns the sum if higher than the treshold value
    noise_array(list): list of percent int values for noise
    """

    # Load an image based on the given path
    image = Image.open(image_path).convert('L')
    image = image.resize((500, 500), 1)

    image_array = np.array(image)

    FPs = []
    FNs = []

    for i in range(0, len(noise_array)):

        # Add salt-and-pepper noise to the image.
        noise_img = random_noise(image_array, mode='s&p',
                                 amount=noise_array[i])
        noise_img = np.array(255*noise_img, dtype = 'uint8')

        # Convert the images and plot them
```

```
        new_image, coordinates = feature_detector \
        (kernel, noise_img, treshold)

        coordinates = np.asarray(coordinates)
        oc = np.array(official_coordinates)

        coordinates[:,1] = 1000 * coordinates[:,1]
        oc[:,1] = 1000 * oc[:,1]

        c2 = np.add(coordinates[:,1], coordinates[:,0])
        oc2 = np.add(oc[:,1], oc[:,0])

        # Get the number of false positive and false negatives
        FP = len(np.setdiff1d(c2, oc2))
        FN = len(np.setdiff1d(oc2, c2))

        FPs.append(FP)
        FNs.append(FN)

    # display results
    plt.scatter(noise_array, FPs, label = "False Positives")
    # plotting the line 2 points
    plt.scatter(noise_array, FNs, label = "False Negatives")
    plt.xlabel('Salt and Pepper Noise Amount')
    # Set the y axis label of the current axis.
    plt.ylabel('FP and FN')
    # Set a title of the current axes.
    plt.title('Change in FP and FN as noise amount increases')
    # show a legend on the plot
    plt.legend()
    # Display a figure.
    plt.show()
```

[215]:
```
# Let's test this on few different images
kernel =  [[1., 2., 3., 2., 1.],
           [2., 3., 5., 3., 2.],
           [3., 5., 8., 5., 3.],
           [2., 3., 5., 3., 2.],
           [1., 2., 3., 2., 1.]]

treshold = 8000

graph_noise(kernel, 'images/stars2.jpg',treshold, [0.1, 0.2, 0.3, 0.4, 0.5])
```



The false negatives (objects that are stars but where not identified) are almost minimal, while the number of false positives (objects that are not stars but are identified as starts) grow exponentially with increasing amounts of noise.

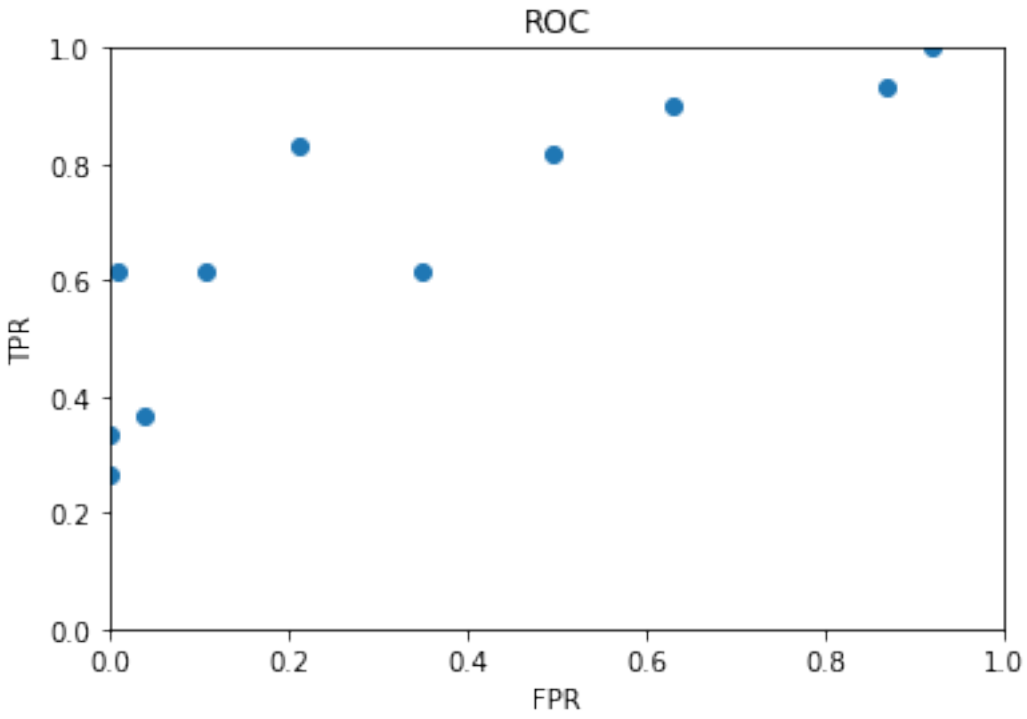
1.4 I have plotted the ROC curve with 10 different noise levels and varying thresholds.

[216]:
```
def graph_ROC(kernel, image_path, treshold, noise_array):
    """
    Draws ROC curve.

    Parameters:
    kernel (list of lists): 2D kernel that modifies the image
    image_path (string): link to an input image
    treshold (int): only returns the sum if higher than the
    treshold value
    noise_array(list): list of percent int values for noise
    """

    # Load an image based on the given path
    image = Image.open(image_path).convert('L')
    image = image.resize((500, 500), 1)

    image_array = np.array(image)
```

```
FPs = []
FNs = []
TPs = []
TPRs = []
FPRs = []

for i in range(0, len(noise_array)):

    # Add salt-and-pepper noise to the image.
    noise_img = random_noise(image_array, mode='s&p',
                             amount=noise_array[i])
    noise_img = np.array(255*noise_img, dtype = 'uint8')

    # Convert the images and plot them
    new_image, coordinates = feature_detector \
    (kernel, noise_img, treshold[i])

    coordinates = np.asarray(coordinates)
    oc = np.array(official_coordinates)

    coordinates[:,1] = 1000 * coordinates[:,1]
    oc[:,1] = 1000 * oc[:,1]

    c2 = np.add(coordinates[:,1], coordinates[:,0])
    oc2 = np.add(oc[:,1], oc[:,0])

    # Get the number of false positive and false negatives
    FP = len(np.setdiff1d(c2, oc2))
    FN = len(np.setdiff1d(oc2, c2))
    TP = len(np.intersect1d(oc2, c2))

    FPs.append(FP)
    FNs.append(FN)
    TPs.append(TP)

    TPR = TP/len(oc2)
    FPR = FP/(500*500 - len(oc2))

    TPRs.append(TPR)
    FPRs.append(FPR)

# display results
plt.scatter(FPRs, TPRs)
plt.ylim(0, 1)
plt.xlim(0, 1)
plt.xlabel('FPR')
```

```
    plt.ylabel('TPR')
    plt.title('ROC')
    plt.show()
```

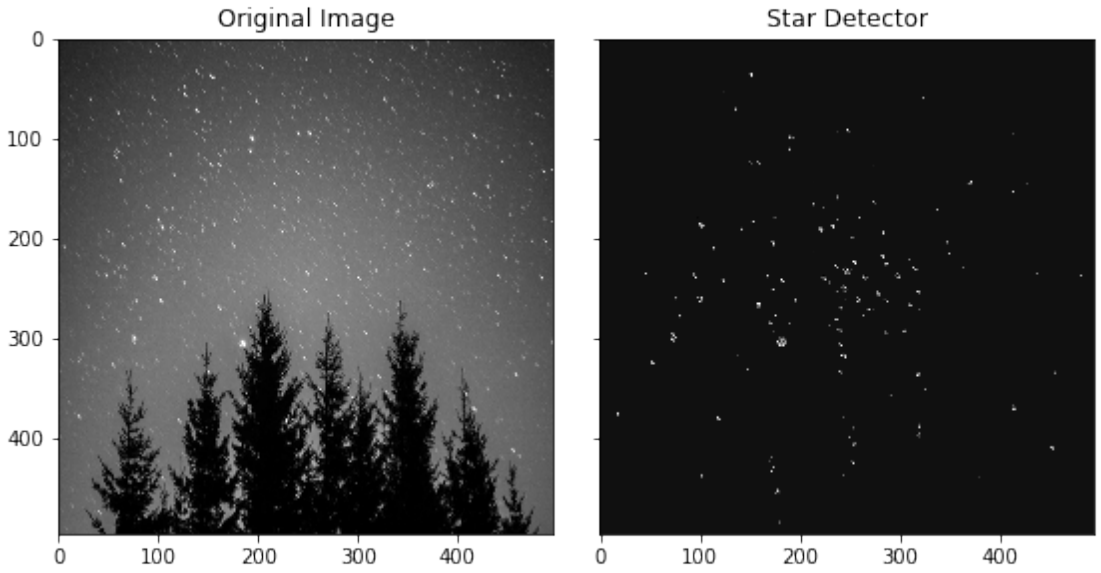
[239]:
```
# Let's test this on few different images
kernel =  [[1., 2., 3., 2., 1.],
           [2., 3., 5., 3., 2.],
           [3., 5., 8., 5., 3.],
           [2., 3., 5., 3., 2.],
           [1., 2., 3., 2., 1.]]

treshold = [7000, 7000, 7000, 7000, 7000,
            7000, 7000, 7000, 7000, 6000, 6000]

graph_ROC(kernel, 'images/stars2.jpg',treshold,
          [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
```



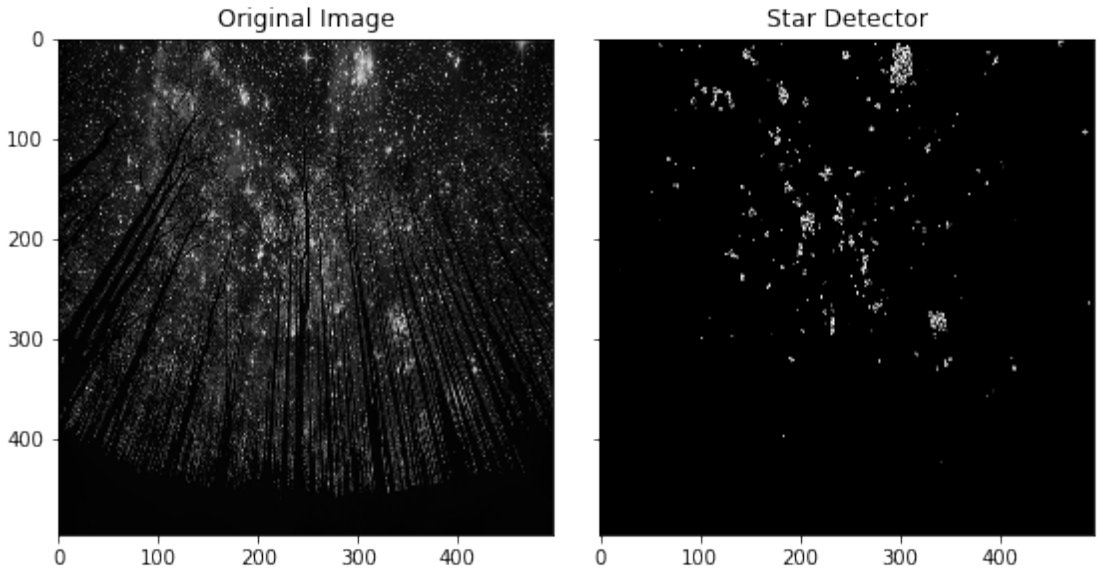
As can be seen in the figure above we get fairly high area under the ROC, which means that with varying amounts of tresholding our function performs well at star detection.

1.5 I wanted to look at some more complex images of the night sky to see how well the algorithm performs. As long as the threshold value is relatively correct, the performance is very good.

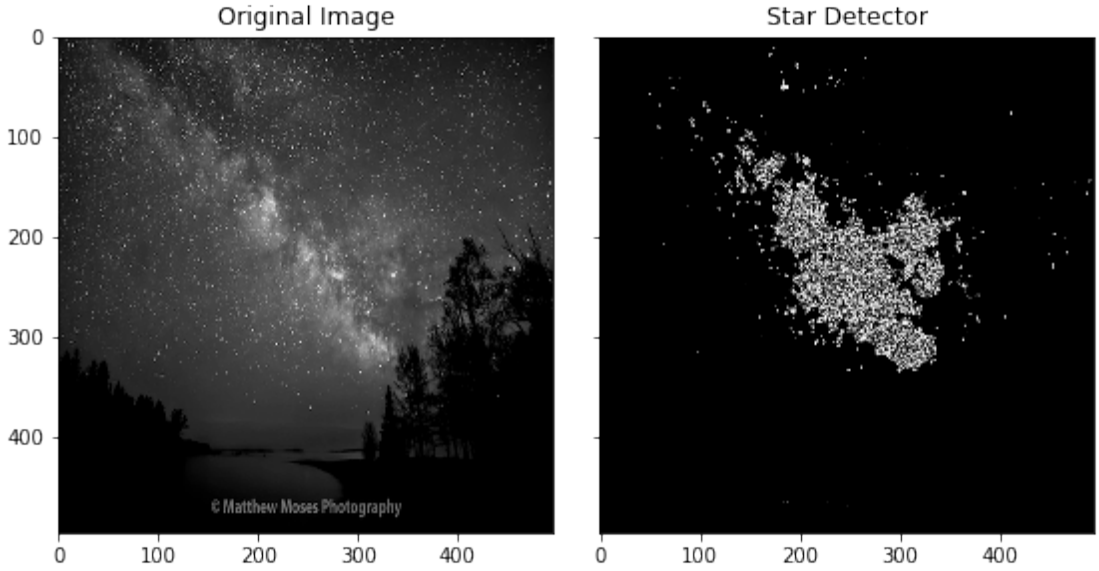
[218]:
```
kernel =  [[1., 2., 3., 2., 1.],
           [2., 3., 5., 3., 2.],
           [3., 5., 8., 5., 3.],
           [2., 3., 5., 3., 2.],
           [1., 2., 3., 2., 1.]]

treshold = 10000

display_detector(kernel, 'images/sky3.jpg', treshold)
```



[219]:
```
kernel =  [[1., 2., 3., 2., 1.],
           [2., 3., 5., 3., 2.],
           [3., 5., 8., 5., 3.],
           [2., 3., 5., 3., 2.],
           [1., 2., 3., 2., 1.]]

treshold = 6400

display_detector(kernel, 'images/sky1.jpeg', treshold)
```



[220]:
```
kernel =  [[1., 2., 3., 2., 1.],
           [2., 3., 5., 3., 2.],
           [3., 5., 8., 5., 3.],
           [2., 3., 5., 3., 2.],
           [1., 2., 3., 2., 1.]]

treshold = 6400

display_detector(kernel, 'images/sky4.png', treshold)
```



The function does; however, struggle with large patches of light such as galaxies on the night sky. This can be seen in the picture above.

0.2.2 Exercise 2

2.1 An image can be represented as follows:

$$I = w * S \tag{1}$$

Where I is the image represented as a function of the coefficients w and S. Here I investigate the basis functions for a 16 x 16 discrete Fourier transformation.

DCT is used in space-frequency analyses to represent image on its amplitude responses to a set of spatial frequencies. Instead of representing an image as a set of pixel values, we consider it to be an array of amplitude response coefficients. Each coefficient corresponds to the response of the input to a particular spatial frequency.

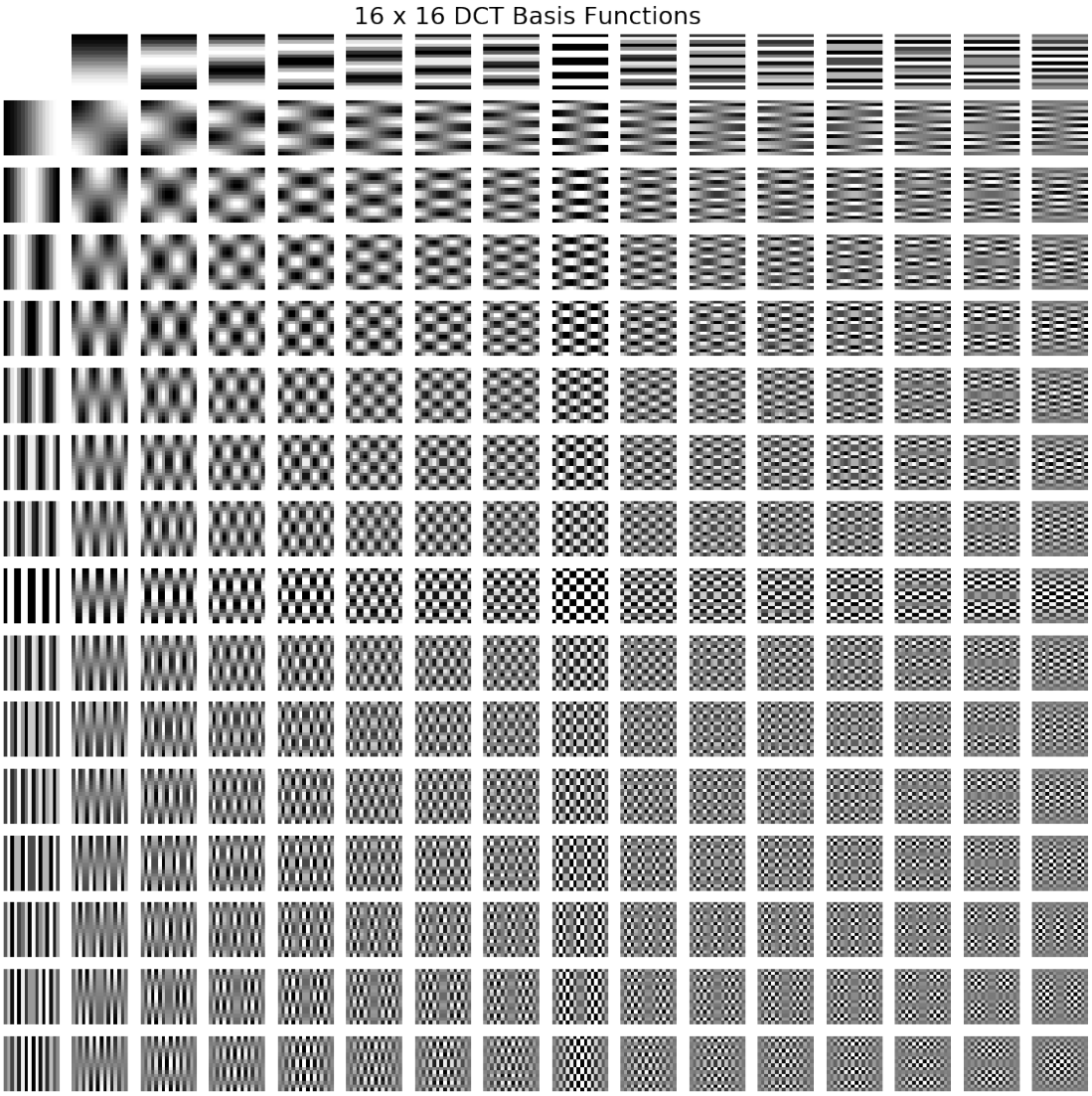
Let's first create a function that computes the basis functions of an m by n matrix.

[221]:
```
def get_all_basis_functions(m, n):
    basis_matrix = np.zeros((m, n, m*n))
    x = 0

    for i in range(m):
        for j in range(n):

            if i == 0:
                a = 1 / np.sqrt(m)
            else:
                a = np.sqrt(2 / m)

            if j == 0:
                b = 1 / np.sqrt(n)
            else:
                b = np.sqrt(2 / n)

            for k in range(m):
                for l in range(n):
                    c = np.cos((np.pi * (2 * k + 1) * i) / (2 * m))
                    d = np.cos((np.pi * (2 * l + 1) * j) / (2 * n))

                    result = a * b * c * d

                    basis_matrix[l, k, x] = result

            x = x + 1
```

```
    return basis_matrix
```

Now let's graph all the basis function using Matplotlib.

[222]:
```
functions = get_all_basis_functions(16, 16)

plt.figure(figsize=(20, 20))
for i in range(0, 256):
    plt.subplot(16, 16, i + 1)
    image = functions[:, :, i]
    plt.axis('off')
    plt.imshow(image, cmap='binary')

plt.suptitle('16 x 16 DCT Basis Functions', y = 0.9, size = 25)
plt.show();
```

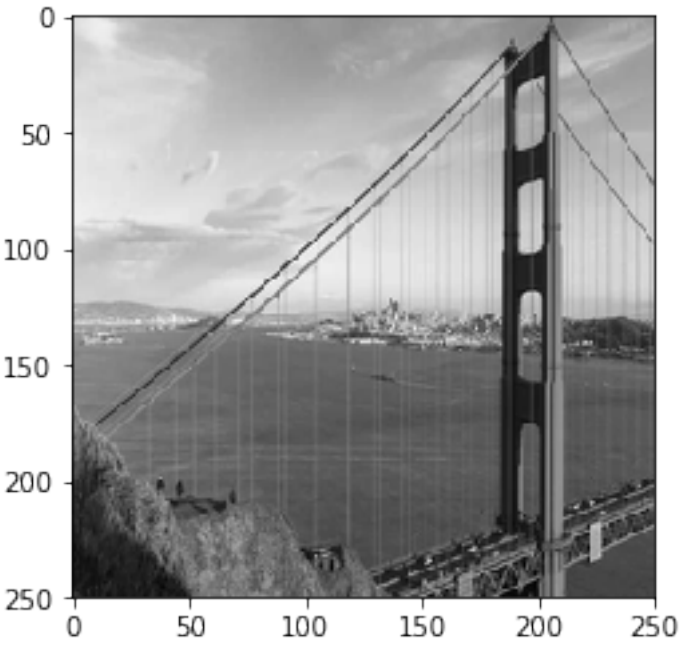


2.2 Let's first define some functions needed in the analysis before to reconstruct images and retrieve the DCT coefficients and back into an image using IDCT.

[223]:
```
def get_dct(image):
    return fftpack.dct(fftpack.dct(image.T, norm = 'ortho').T, norm = 'ortho')

def get_idct(coefficients):
    return fftpack.idct(fftpack.idct(coefficients.T, norm = 'ortho').T, norm =␣
 ↪'ortho')

def get_reconstructed_image(raw):
    img = raw.astype('uint8')
    return img
def n_max(arr, n):
    indices = arr.ravel().argsort()[-n:]
    indices = (np.unravel_index(i, arr.shape) for i in indices)
    return [(arr[i], i) for i in indices]
```

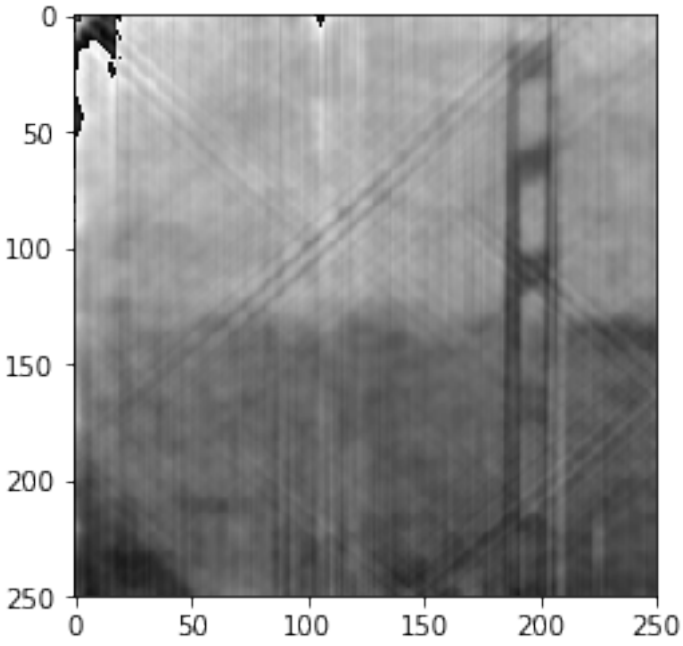
[224]:
```
image = Image.open('images/sf.jpg').convert('L')
image = image.resize((250, 250), 1)
image = np.array(image, dtype=np.float)
plt.imshow(image, cmap='gray')
```

[224]:
```
<matplotlib.image.AxesImage at 0x1262e59e8>
```



[225]:
```
dct_matrix = get_dct(image)

coordinates = n_max(dct_matrix, 500)
dct_matrix[dct_matrix < coordinates[0][0]] = 0


new_image_array = np.zeros(shape = (image.shape[0],
                                    image.shape[1]))

new_image_array = get_idct(dct_matrix)

new_image = get_reconstructed_image(new_image_array)
plt.imshow(new_image, cmap='gray')
```

[225]:
```
<matplotlib.image.AxesImage at 0x1264d0c88>
```



We can reconstruct the image fairly well using only 500 functions with the highest coefficients out of 62,500 total functions.

The two-dimensional discrete cosine transform (DCT) represents an image as a sum of sinusoids. The two-dimensional DCT of M x N matrix A is defined such as:

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} cos \frac{\pi(2m+1)p}{2M} cos \frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \leq p \leq M-1 \\ 0 \leq q \leq N-1 \end{matrix}$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \leq p \leq M-1 \end{cases} \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \leq q \leq N-1 \end{cases}$$

DCT can be inverted such that:

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \leq p \leq M-1 \end{cases} \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \leq q \leq N-1 \end{cases}$$

Which means that any matrix that is M X N can be written as a sum of MN basis functions defined by:

$$\alpha_p \alpha_q cos \frac{\pi(2m+1)p}{2M} cos \frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \leq p \leq M-1 \\ 0 \leq q \leq N-1 \end{matrix}$$

0.2.3 Exercise 3

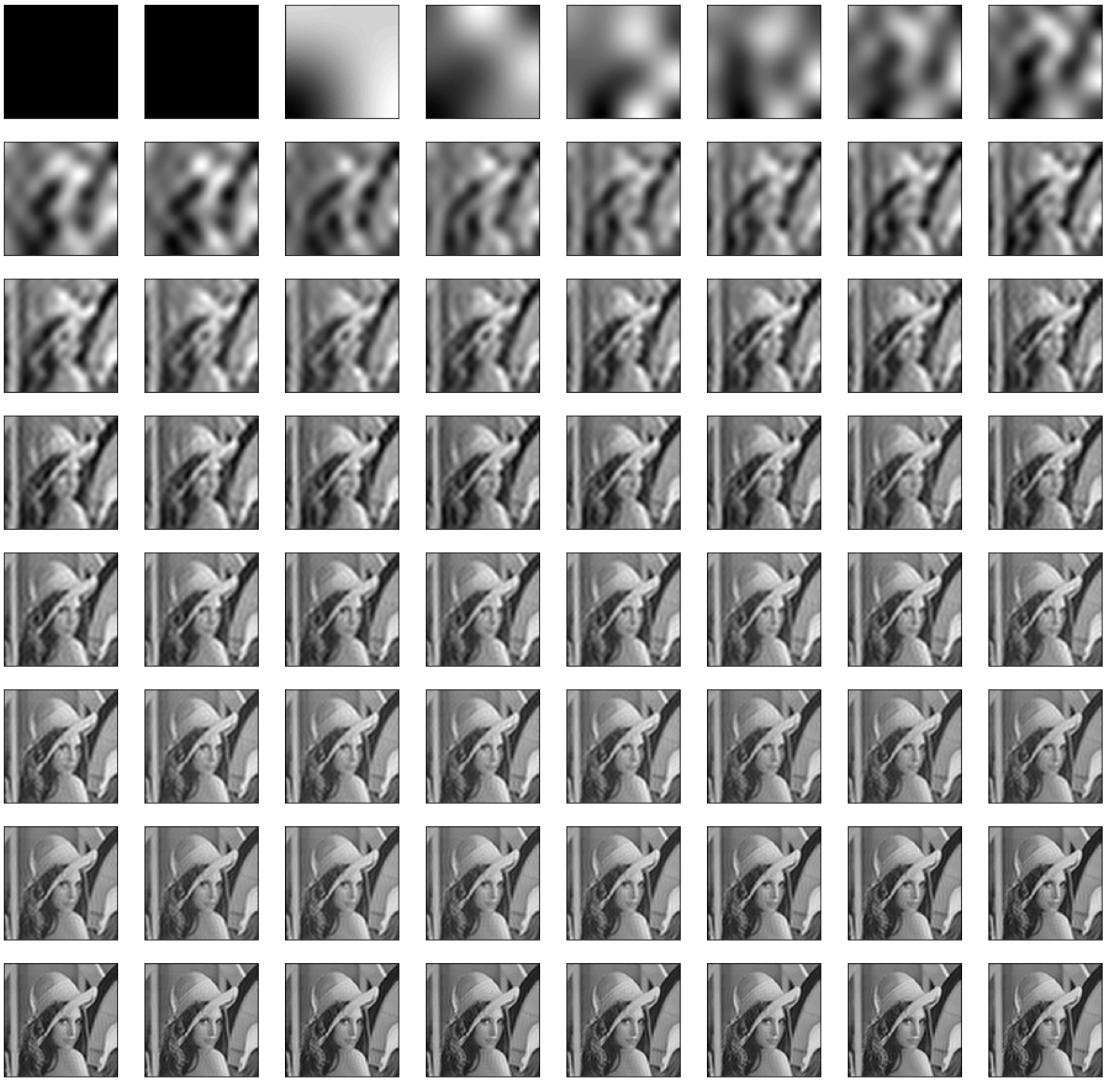
3.1 We will reuse the Scipy DCT and IDCT functions from exercise 2. Now, let's create a function to zero out components of an image.

[226]:
```
def zero_out_components (image_path):
    image = Image.open(image_path).convert('L')
    image = image.resize((250, 250), 1)
    image = np.array(image, dtype=np.float)

    dct_size = image.shape[0]
    dct = get_dct(image)
    reconstructed_images = []

    for i in range(dct_size):
        dct_copy = dct.copy()
        dct_copy[i:, :] = 0
        dct_copy[:, i:] = 0

        r_image = get_idct(dct_copy)
        reconstructed_image = get_reconstructed_image(r_image)

        reconstructed_images.append(reconstructed_image)

    return reconstructed_images
```

Low Pass Filter We can view the low-frequency images by only showing the first k components out of the reconstructed images array. These preserve the low frequencies, and at the same time attenuate high frequencies.

Let's try with k = 64.

[227]:
```
recon_arr = zero_out_components('images/lady.png')

fig = plt.figure(figsize=(16, 16))
for i in range(64):
    plt.subplot(8, 8, i + 1)
    plt.imshow(recon_arr[i], cmap=plt.cm.gray)
    plt.grid(False);
    plt.xticks([]);
    plt.yticks([]);
```



High Pass Filter Similarly for high frequency images, we can repeat the function but we zero out the low coefficients.

Let's again try with k = 64 taken from the back.

[228]:
```
recon_arr = zero_out_components('images/lady.png')

fig = plt.figure(figsize=(16, 16))
for i in range(186, 250):
    plt.subplot(8, 8, i - 186 + 1)
    plt.imshow(recon_arr[i], cmap=plt.cm.gray)
    plt.grid(False);
    plt.xticks([]);
    plt.yticks([]);
```

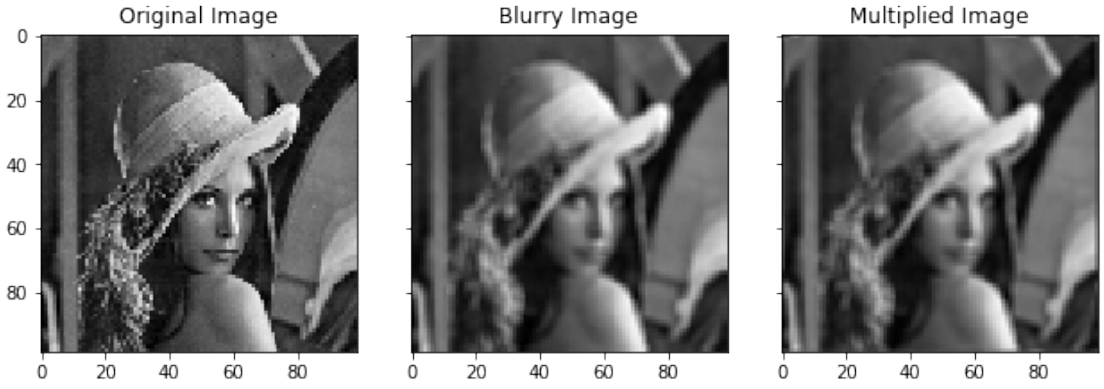
3.2 Multiplication in the frequency domain is equivalent to convolution in the spatial domain. Let's demonstrate it with the bluring filter.

Convolving Sharpening Kernel with an Image

[229]:
```
img = cv2.imread('images/lady.png')[:, :, 0]
img = cv2.resize(img, (99, 99))

blurry_kernel = np.array([[1/9, 1/9, 1/9],
                          [1/9,  1/9, 1/9],
                          [1/9, 1/9, 1/9]])

filtered_image = cv2.filter2D(img, -1, blurry_kernel)


# display results
fig, ax = plt.subplots(ncols=2, sharex=True, sharey=True,
                       figsize=(8, 4))

ax[0].imshow(img, cmap=plt.cm.gray)
ax[0].set_title('Original Image')

ax[1].imshow(filtered_image, cmap=plt.cm.gray)
ax[1].set_title("Bluring Filter")

plt.tight_layout()
plt.show()
```

Multiplying Transforms of Kernel and Image Now let's multiply the transforms of image with the kernel and then apply inverse transform.

[230]:
```
# size we are matching
sz = img.shape

# pad the kernel
sz = (sz[0] - blurry_kernel.shape[0], sz[1] - blurry_kernel.shape[1])  # total␣
↪amount of padding
padded_kernel = np.pad(blurry_kernel, (((sz[0]+1)//2, sz[0]//2), ((sz[1]+1)//2,␣
↪sz[1]//2)), 'constant')
padded_kernel = fftpack.ifftshift(padded_kernel)

# multiply them together an inverse to reconstruct the original
recon_image = np.real(fftpack.ifft2(fftpack.fft2(img) * fftpack.
↪fft2(padded_kernel)))

# display results
fig, ax = plt.subplots(ncols=3, sharex=True, sharey=True,
                       figsize=(9, 3))

ax[0].imshow(img, cmap=plt.cm.gray)
ax[0].set_title('Original Image')

ax[1].imshow(filtered_image, cmap=plt.cm.gray)
ax[1].set_title("Blurry Image")

ax[2].imshow(recon_image, cmap=plt.cm.gray)
ax[2].set_title("Multiplied Image")

plt.tight_layout()
plt.show()
```



We managed to retrieve the exact same image! However, this did not work without padding the kernel correctly.

0.2.4 Exploration

For the exploration section, I wanted to study the principal components of the fetch_olivetti_faces dataset. I will attempt to show that individual components can be approximated by the sum of the first k principal components.

Each image has a size of 64 x 64, represented by a 4096 element vector. All of these dimensions might not be necessary to capture the variations between the faces, hence let's investigate this using PCA.

[231]:
```
faces_info = fetch_olivetti_faces()
faces = faces_info['data']
```

Let's visualize the first 25 faces.

[232]:
```
faces = faces.reshape((400, 64, 64))
plt.figure(figsize=(12, 12))

for i in range(25):
    face = faces[i]
    plt.subplot(5, 5, i + 1)
    plt.imshow(face, cmap='gray')
    plt.axis('off')

plt.suptitle('First 25 Olivetti Faces', y = 0.95, size = 28)
plt.show();
```

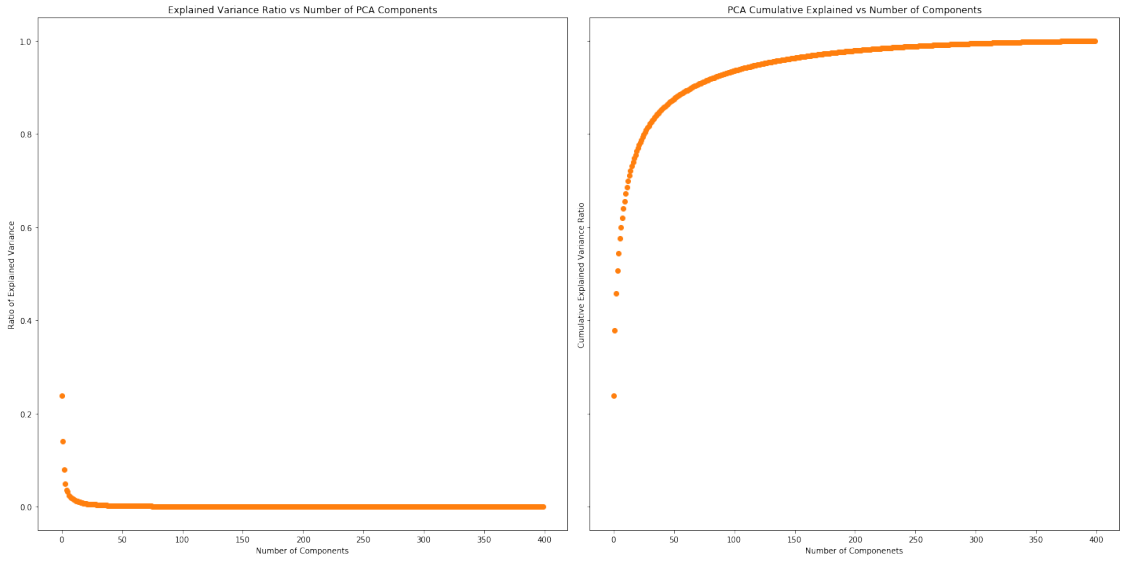
Now let's perform PCA such that we can get the images as 400 entry vectors.

[233]:
```
faces = faces.reshape((400, 64*64))
pca = PCA()
trans = pca.fit_transform(faces)
```

Now let's investigate what the ratio of explained variance and cumulative ratio of explained variance is in the 400 vectors.

[234]:
```
# display results
fig, ax = plt.subplots(ncols=2, sharex=True, sharey=True,
                       figsize=(20, 10))
```

```
ax[0].plot(list(range(len(pca.explained_variance_ratio_))),
           pca.explained_variance_ratio_, 'bo', c='#ff7f0e')
ax[0].set_xlabel('Number of Components')
ax[0].set_ylabel('Ratio of Explained Variance')
ax[0].set_title('Explained Variance Ratio vs Number of PCA Components')

ax[1].plot(list(range(len(np.cumsum(pca.explained_variance_ratio_)))),
           np.cumsum(pca.explained_variance_ratio_), 'bo', c='#ff7f0e')
ax[1].set_xlabel('Number of Componenets')
ax[1].set_ylabel('Cumulative Explained Variance Ratio')
ax[1].set_title("PCA Cumulative Explained vs Number of Components")

plt.tight_layout()
plt.show()
```



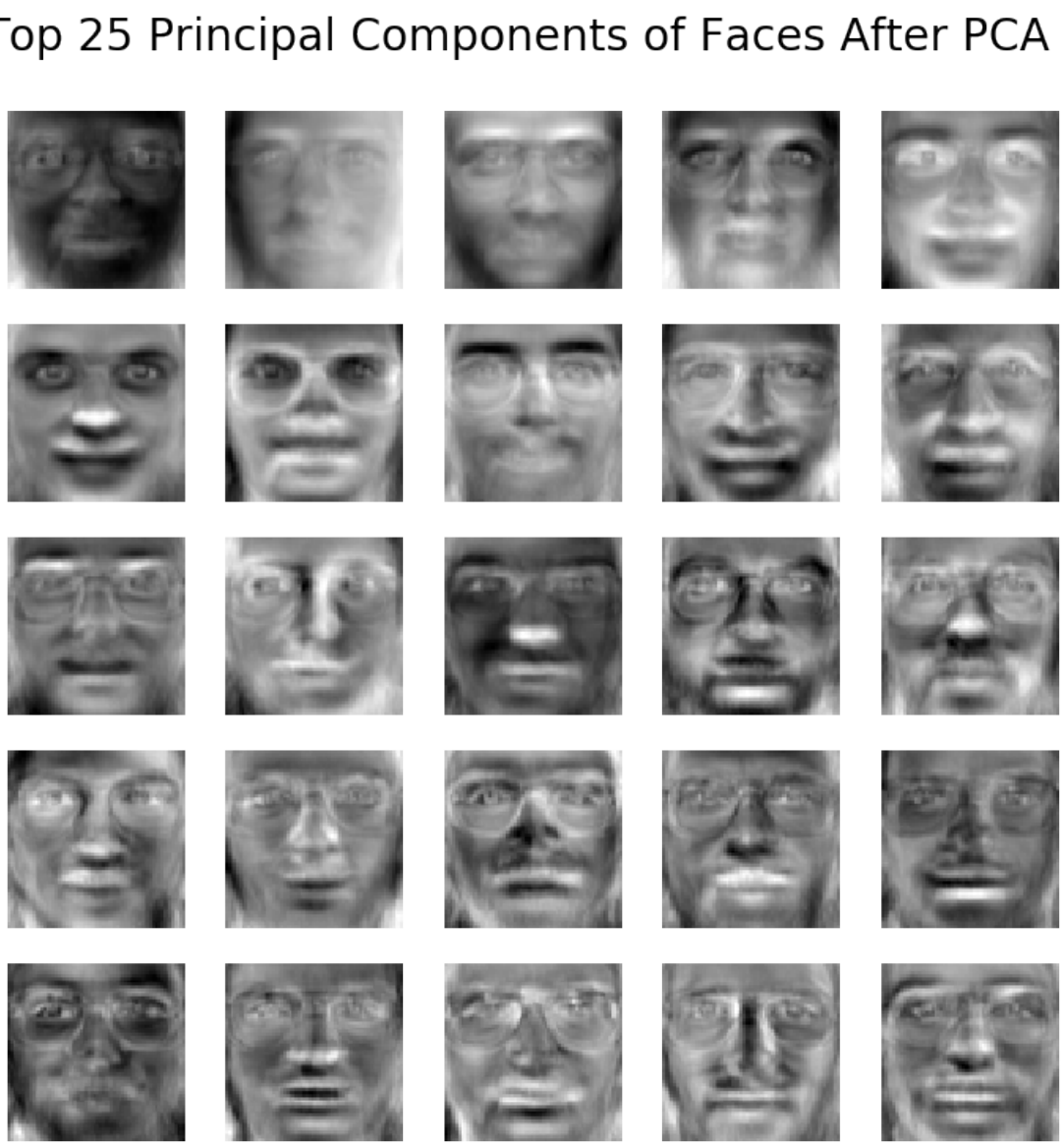
We can see that, from the first graph, the first few components capture much more variance than the rest. This is even more apparent from the graph on the right. In fact, in order to capture 99% of variances we only need 259 components as can be seen from the computation below.

[235]:
```
np.where(np.cumsum(pca.explained_variance_ratio_)
 > 0.99)[0][0]
```

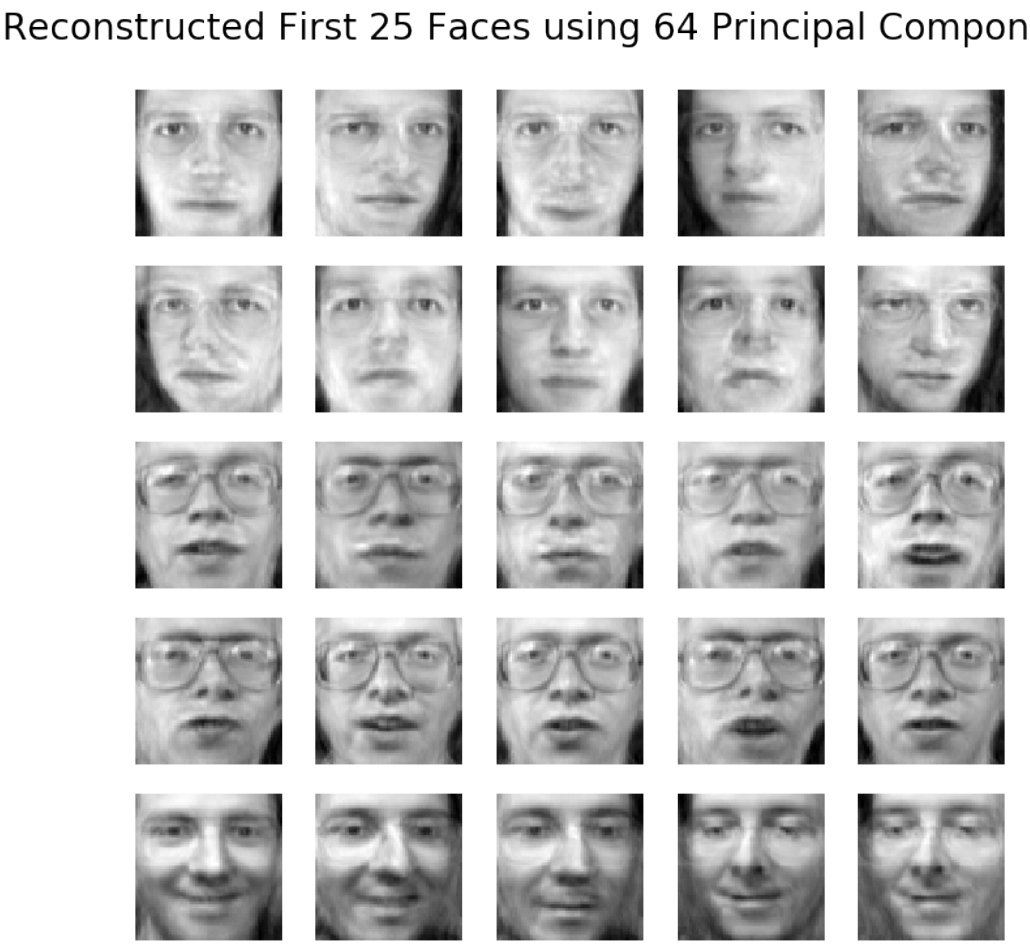
[235]: 259

Now let's try to view the actual principal components and view the top 25 after performing PCA with 64. Each components are size 64 by 64 (4096). These are not the transformed images from before but components that constitute faces.

[236]:
```
pca_reduced = PCA(n_components=64)
faces_reduced = pca_reduced.fit_transform(faces.reshape((400, 4096)))
plt.figure(figsize=(12, 12))

for i in range(25):
    plt.subplot(5, 5, i + 1)
    pc_face = pca_reduced.components_[i, :].reshape((64, 64))
    plt.imshow(pc_face, cmap='gray');
    plt.axis('off')

plt.suptitle('Top 25 Principal Components of Faces After PCA', y = 0.95, size =␣
↪28);
plt.show();
```



[237]:
```
reverted = pca_reduced.inverse_transform(faces_reduced).reshape((400, 64, 64))

plt.figure(figsize=(12, 12))

for i in range(25):
    face = reverted[i]
    plt.subplot(5, 5, i + 1)
    plt.imshow(face, cmap='gray')
    plt.axis('off')

plt.suptitle('Reconstructed First 25 Faces using 64 Principal Components', y =␣
↪0.95, size = 28)
plt.show();
```



Conclusion After doing PCA on the Olivetti data set we have managed to retain nearly 90% of the variation in the faces while we kept the number of parameters necessary to specify the images by 82%. From the reconstruction we can see that we can get away with only using the top 64 components. This method works well in decreasing the number of features/dimensions while retaining most of the necessary information to reconstruct various faces.

The extra computations required for the conclusion can be seen below.

[238]:
```
ic = 400 * 4096
nc = (64 * 4096) + (400 * 64)
print('% variance by 64 components: ', np.sum(pca_reduced.
 ↪explained_variance_ratio_))
print('initial number of params: ', ic)
print('reduced number of params:', nc)
print('% reduced ',100 * (1 - nc/ic))
```

```
% variance by 64 components:  0.89702946
initial number of params:  1638400
reduced number of params: 287744
% reduced  82.4375
```