6252320-Debnath_proj2

June 3, 2022

0.0.1 6252320 - Leon Debnath

[1]:
```
from sklearn.preprocessing import normalize
from skimage.util import random_noise
import matplotlib.pyplot as plt
import numpy as np
import cv2
import math
```

[2]:
```
def display(images, labels, axis='off', cmap=None, figsize=(15,10), cols=2):

    if len(images) != len(labels):
        raise Exception("images and labels don't correspond")

    fig = plt.figure(figsize=figsize)
    rows = math.ceil(len(images)/cols)

    for i in range(len(images)):
        fig.add_subplot(rows, cols, i+1)
        plt.imshow(images[i], cmap=cmap)
        plt.title(labels[i])
        plt.axis(axis)
```

1 Exercise 1 - Image Degradation and Motion Blur

1.1 Degrading & Additive Noise

1.1.1 Motion Blurring

A motion blur filter H is created where $H(u,v) = sinc(\alpha \cdot u + \beta \cdot v) \cdot e^{-j\pi(\alpha \cdot u + \beta \cdot v)}$. The variables α and β tune the amount of blur for horizontal and vertical directions respectively. To blur the image, the colour channels are split and each channel is transformed into the fourier domain seperately, as though a greyscale image. Then each channel is multiplied by $H(u,v)$ before being returned to the spacial domain.

[3]:
```
def blur_channel(c, a, b):
    """ Blurs a single colour channel (n x m numpy array) with values a and b"""
    height, width = c.shape
```

```
    # Fast fourier transform the channel
    c_fft = np.fft.fft2(c)

    # Create a mask for the image (H) in the frequency domain
    [u, v] = np.mgrid[-round(height/2):round(height/2), -round(width/2):
↪round(width/2)]
    u = 2 * u / height
    v = 2 * v / width
    h = np.sinc((u*a + v*b)) * np.exp(-1j * np.pi * (u*a + v*b))

    # Apply the mask and inverse fourier transform before returning the␣
↪absolute value
    return np.abs(np.fft.ifft2(c_fft * h))


def blur(img, a, b):
    """ Blurs an image (n x m x d numpy array) with values a and b"""
    (c1, c2, c3) = cv2.split(img)
    c1_ = blur_channel(c1, a, b)
    c2_ = blur_channel(c2, a, b)
    c3_ = blur_channel(c3, a, b)
    img_blurred = cv2.merge((c1_, c2_, c3_))
    return img_blurred / np.max(img_blurred)
```

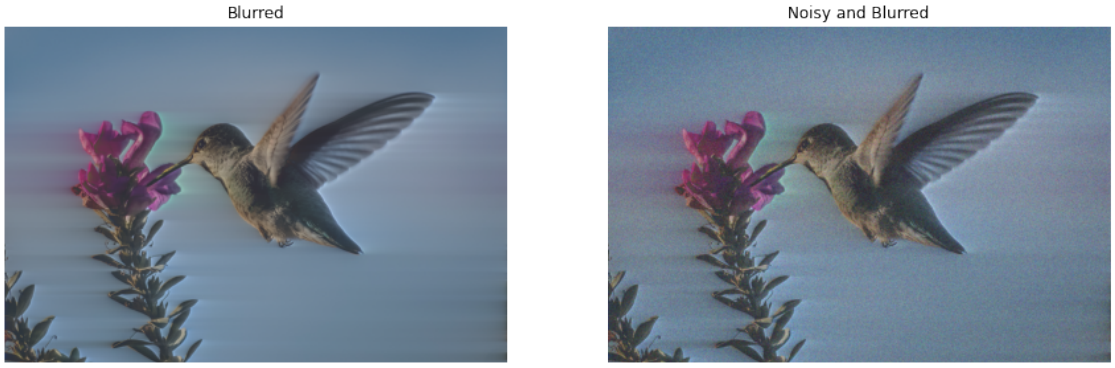
In this case the values for α and β were set to 0.15 to create the blurred images to be a realistic motion blur, but not excessively noisy.

[4]:
```
bird = cv2.cvtColor(cv2.imread('images/bird.jpg'), cv2.COLOR_BGR2RGB)
bird = bird/255

bird_blurred_h = blur(bird, 0, 0.15)
bird_blurred_v = blur(bird, 0.15, 0)

display([bird_blurred_h, bird_blurred_v], ['Horizontal Blur', 'Vertical Blur'])
```

1.1.2 Gaussian Noise

Gaussian noise was added by creating an array the same size as the image full of random numbers that fit the distribution with a $\mu = 0$ and $\sigma = 0.04$. This array was then summed with the image to create random distortions within all three colour channel. The noise manifested itself as random coloured speckling uniformly throughout the image.

[5]:
```
# Add random noise to the horizontally blurred image
bird_gaussian = random_noise(bird_blurred_h, "gaussian", mean=0, var=0.04)
display([bird_blurred_h, bird_gaussian], ['Blurred', 'Noisy and Blurred'])
```



1.1.3 Display Images

The final product conception can be observed in the three stages; in it's original form (1), blurred using the horizontal motion blur (2) and with noise added. This transformation is represented by the formula:

$$G(u, v) = F(u, v)H(u, v) + N(u, v)$$

where F is the Fourier Transform, H is the motion blur and N is the gaussian noise

[6]:
```
display([bird, bird_blurred_h, bird_gaussian], ['Original', 'Blurred', 'Noisy␣
↪and Blurred'], cols=3)
```

1.2 Removing Noise

If H is known, removal of blur is a simple process of transforming to the Fourier domain, dividing the image by $H(u, v)$ and then returning the image back to the spacial domain. This is represented by the formula:

$$G(u, v) = \frac{F(u, v)}{H(u, v)}$$

[7]:
```
def inv_blur_channel(c, a, b):
    """ Inverse blurs a single colour channel (n x m numpy array) with values a␣
 ↪and b"""
    height, width = c.shape

    # Fast fourier transform the channel
    c_fft = np.fft.fft2(c)

    # Create a mask for the image (H) in the frequency domain
    [u, v] = np.mgrid[-round(height/2):round(height/2), -round(width/2):
 ↪round(width/2)]
    u = 2 * u / height
    v = 2 * v / width
    h = np.sinc((u*a + v*b)) * np.exp(-1j * np.pi * (u*a + v*b))

    # Apply the mask and inverse fourier transform before returning the␣
 ↪absolute value
    return np.abs(np.fft.ifft2(c_fft / h))


def inv_blur(img, a, b):
    """ Inverse blurs an image (n x m x d numpy array) with values a and b"""
    (c1, c2, c3) = cv2.split(img)
    c1_ = inv_blur_channel(c1, a, b)
    c2_ = inv_blur_channel(c2, a, b)
    c3_ = inv_blur_channel(c3, a, b)
    img_blurred = cv2.merge((c1_, c2_, c3_))
    return img_blurred / np.max(img_blurred)
```

1.2.1 Inverse Blur Filtering

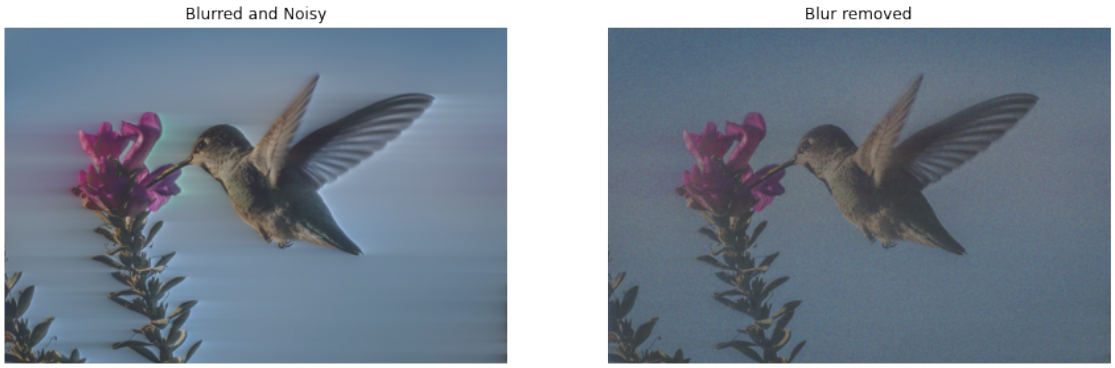
The result of applying the inverse blur is shown below, although it does a good job of removing the blur, the image is clearly not as sharp as the original.

[8]:
```
display([bird_blurred_h, inv_blur(bird_blurred_h, 0, 0.15)],['Horizontal Blur',␣
↪'Blur removed'])
```

1.2.2 Inverse Blur Filtering with Noise

When gaussian noise has been applied to the image, the inverse filtering has no effect on the noise. This is due to the fact that the noise is random and thus cannot be removed by the application of a periodic function.

[9]:
```
display([bird_blurred_h, inv_blur(bird_gaussian, 0, 0.15)], ['Blurred and␣
↪Noisy', 'Blur removed'])
```



1.2.3 Minimum Mean Squared Error (Wiener) Filter

Assuming that the noisy image ($\hat{f}$) and original image (f) are uncorrelated (such as in the case of random gaussian noise) then the expected likelihood of the error is:

$$e^2 = E\{(f - \hat{f})^2\}$$

This expands to the expression:

$$\hat{F} = \left[\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + K}\right]G(u,v)$$

Where:

$H(u,v)$ = degradation function

$H^*(u,v)$ = complex conjugate of$H(u,v)$

$|H(u,v)|^2 = H^*(u,v)H(u,v)$

The filter is applied per channel to the noisy image compared with the original image. When the K-ratio flag is set to false, the value of k is set to 1, essentially making it's effect void.

[10]:
```
def mmse_filter_channel(chl_o, chl_n, k_ratio):
    """Filter a single channel using a MMSE filter

    Parameters:
        chl_o: original image channel (m x n np.array)
        chl_n: noisy image channel (m x n np.array)
        k_ratio: boolean flag if k ratio in use
    """
    # Create the power spectrum of the original image
    o_fft = np.fft.fftshift(np.fft.fft2(chl_o))
    o_pwr_spectrum = np.abs(o_fft) ** 2

    # Create the power spectrum of the noisy image
    n_fft = np.fft.fftshift(np.fft.fft2(chl_n))
    n_pwr_spectrum = np.abs(n_fft) ** 2

    if k_ratio:
        k = np.sum(n_pwr_spectrum) / np.sum(o_pwr_spectrum)
    else:
        k = 1

    # Apply the formula for F_hat
    fft_combined = np.fft.fftshift(np.fft.fft2(chl_o + chl_n))
    img_back_fft = fft_combined / (1 + ((n_pwr_spectrum / o_pwr_spectrum) * k))
    img_back = np.fft.ifft2(img_back_fft)

    return np.abs(img_back)


def mmse_filter(img_o, img_n, k_ratio=False):

    (c1, c2, c3) = cv2.split(img_o)
    (n1, n2, n3) = cv2.split(img_n)
    c1_ = mmse_filter_channel(c1, n1, k_ratio)
    c2_ = mmse_filter_channel(c2, n2, k_ratio)
```

```
    c3_ = mmse_filter_channel(c3, n3, k_ratio)
    img = cv2.merge((c1_, c2_, c3_))
    return img / np.max(img)
```

[11]:
```
display([bird, bird_gaussian, mmse_filter(bird, bird_gaussian)], ['Original',␣
 ↪'Blurred and Noisy', 'Noise removed'],
       cols=3)
```

The addition of the K ratio as a specified constant that is added to all terms of $|H(u,v)|^2$ adapts the formula to:

$$\hat{F} = \left[\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + S_\eta(u,v)/S_f(u,v)}\right]G(u,v)$$

Where:

$S_\eta(u,v) = |N(u,v)|^2 =$ power spectrum of the noise

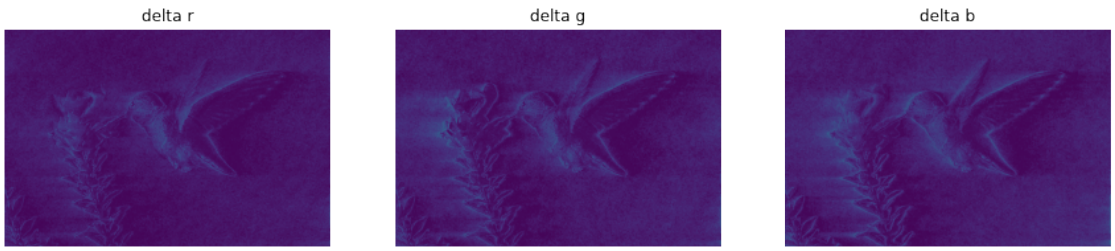
$S_f(u,v) = |F(u,v)|^2 =$ power spectrum of the undegraded image

[12]:
```
bird_wo_k = mmse_filter(bird, bird_gaussian)
bird_w_k = mmse_filter(bird, bird_gaussian, k_ratio=True)
display([bird_wo_k, bird_w_k],['Without K ratio', 'With K ratio'],␣
 ↪figsize=(17,23))
```

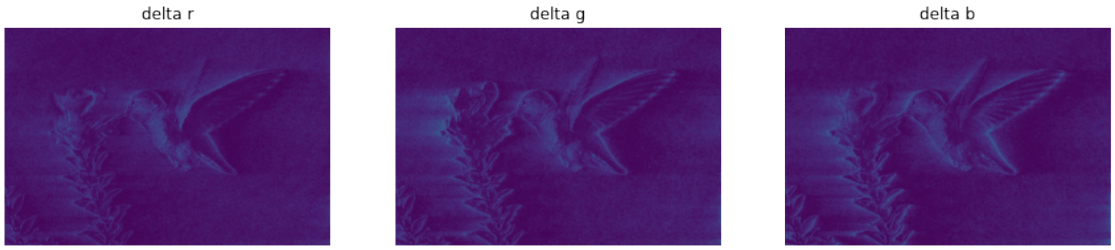
The application of a K ratio shows a slight, if not particularly noticable improvement on the original image, however when the difference for each channel is found between the original image and the MMSE filtered images, a clear difference is apparent, most notably in the blue channel.

[13]:
```
x, y = 800, 650
display([bird_wo_k[x+0:x+500, y+0:y+500], bird_w_k[x+0:x+500, y+0:y+500]],␣
↪['Without', 'With'])
```

[14]:
```
r = np.abs(bird[:, :, 0] - bird_wo_k[:, :, 0])
g = np.abs(bird[:, :, 1] - bird_wo_k[:, :, 1])
b = np.abs(bird[:, :, 2] - bird_wo_k[:, :, 2])
display([r, g, b], ['delta r', 'delta g', 'delta b'], cols=3)
```



[15]:
```
r = np.abs(bird[:, :, 0] - bird_w_k[:, :, 0])
g = np.abs(bird[:, :, 1] - bird_w_k[:, :, 1])
b = np.abs(bird[:, :, 2] - bird_w_k[:, :, 2])
display([r, g, b], ['delta r', 'delta g', 'delta b'], cols=3)
```



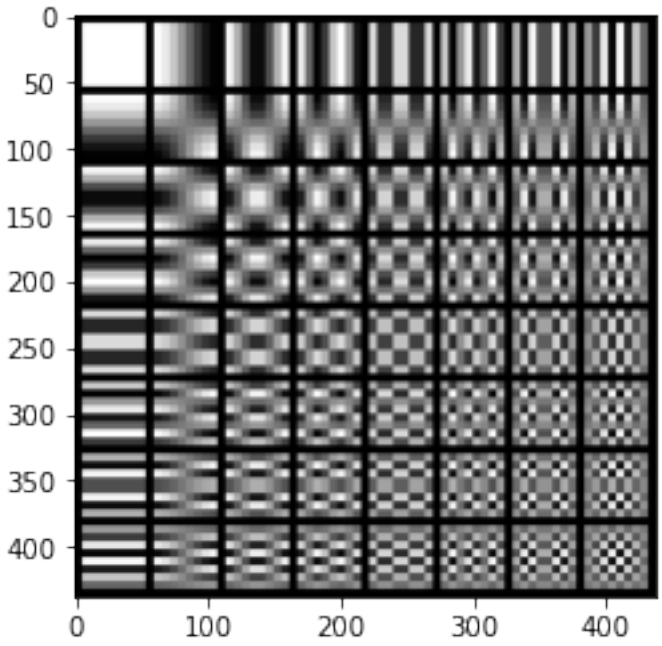
2 Exercise 2 - Hide a Secret Message in an Image DCT

[16]:
```
from scipy import fftpack
from numpy import pi
from numpy import r_

bike = cv2.cvtColor(cv2.imread('images/motogp.jpeg'), cv2.COLOR_BGR2GRAY)
display([bike],['A grayscale image without encoded message'], cmap='gray')
```

2.1 Watermark Insertion

Two functions were used to apply a blockwise Discrete Cosine Transform (DCT) to an image with a variable blcok size. In this case the blocksize of 8 was chosen, returning a 8x8 matrix of coeficients that made up the original image. The top left [0,0] coeficient indicates the value of the DC component (the brightness of the overall block) while every other coeficient holds the value of the contribution of that particular combination of vertical and horizontal cosine waves to the original image, with the lower right [7,7] block showing a checkerbaord pattern that sees a peak of a wave at every other pixel, and a trough at the pixels between peaks.

[17]:
```
plt.imshow(cv2.cvtColor(cv2.imread('images/DCT-8x8.png'), cv2.COLOR_BGR2RGB));
```



[18]:
```
def dct2(a):

    return fftpack.dct( fftpack.dct( a.T, norm='ortho' ).T, norm='ortho' )


def dct(img, block_size=8):

    dct = np.zeros(img.shape)

    for i in r_[:img.shape[0]: block_size]:
        for j in r_[:img.shape[1]: block_size]:
            dct[i: i + block_size, j: j + block_size] = dct2(img[i:i+␣
 ↪block_size, j: j+ block_size])
```

```
    return dct
```

A second two functions were used to compute the inverse DCT from an 8x8 coefficent matrix, returning the original values.

[19]:
```
def idct2(a):

    return fftpack.idct( fftpack.idct( a.T , norm='ortho').T,norm='ortho')


def idct(img_dct, block_size=8):

    img = np.zeros(img_dct.shape)

    for i in r_[:img.shape[0]: block_size]:
        for j in r_[:img.shape[1]: block_size]:
            img[i:(i + block_size), j:(j + block_size)] = idct2( img_dct[i:(i +␣
 ↪block_size),j:(j + block_size)])

    return img
```

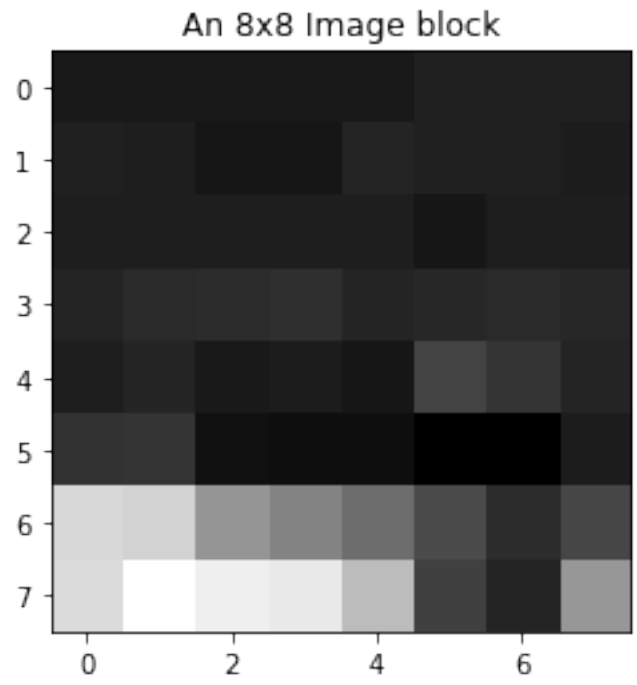
[20]:
```
def apply_threshold(img_dct, threshold):

    return img_dct * (abs(img_dct) > (threshold * np.max(img_dct)))
```

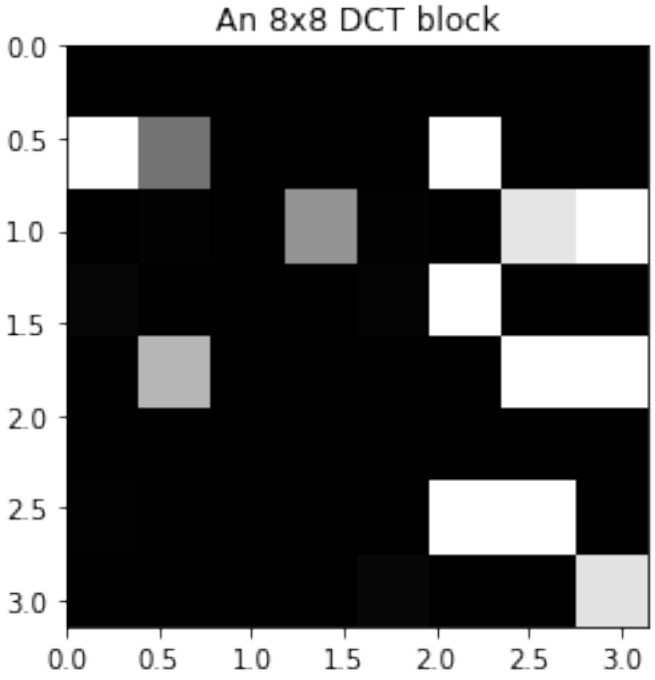
[21]:
```
def filter_k(block, k):
    """ Return the K (absolute) largest values in the same location as found in␣
 ↪the block """
    oput = np.zeros(block.shape)
    block_abs = np.abs(block)

    for i in range(k):
        r, c = np.unravel_index(np.argmax(block_abs, axis=None), block_abs.
 ↪shape)
        oput[r,c] = block[r,c]
        block_abs[r,c] = 0;

    return oput


def filter_k_highest(img_dct, k, block_size=8):

    oput = np.zeros(img_dct.shape)

    for i in r_[:img_dct.shape[0]: block_size]:
        for j in r_[:img_dct.shape[1]: block_size]:
```

```
            oput[i:(i + block_size), j:(j + block_size)] = filter_k(img_dct[i:
↪(i + block_size),j:(j + block_size)], k)

    return oput
```

2.1.1 Compute the 2-D DCT

[22]:
```
x = 55
y = 275
block_size=8

bike_dct = dct(bike)

# Extract a block from image
plt.figure()
plt.imshow(bike[x: x + block_size, y: y + block_size], cmap='gray')
plt.title(f"An {block_size}x{block_size} Image block")

# Display the dct of that block
plt.figure()
plt.imshow(bike_dct[x: x + block_size, y: y + block_size],
           cmap='gray',
           vmax=np.max(bike_dct) * 0.01,
           vmin=0,
           extent=[0,pi,pi,0])

plt.title(f"An {block_size}x{block_size} DCT block");
```

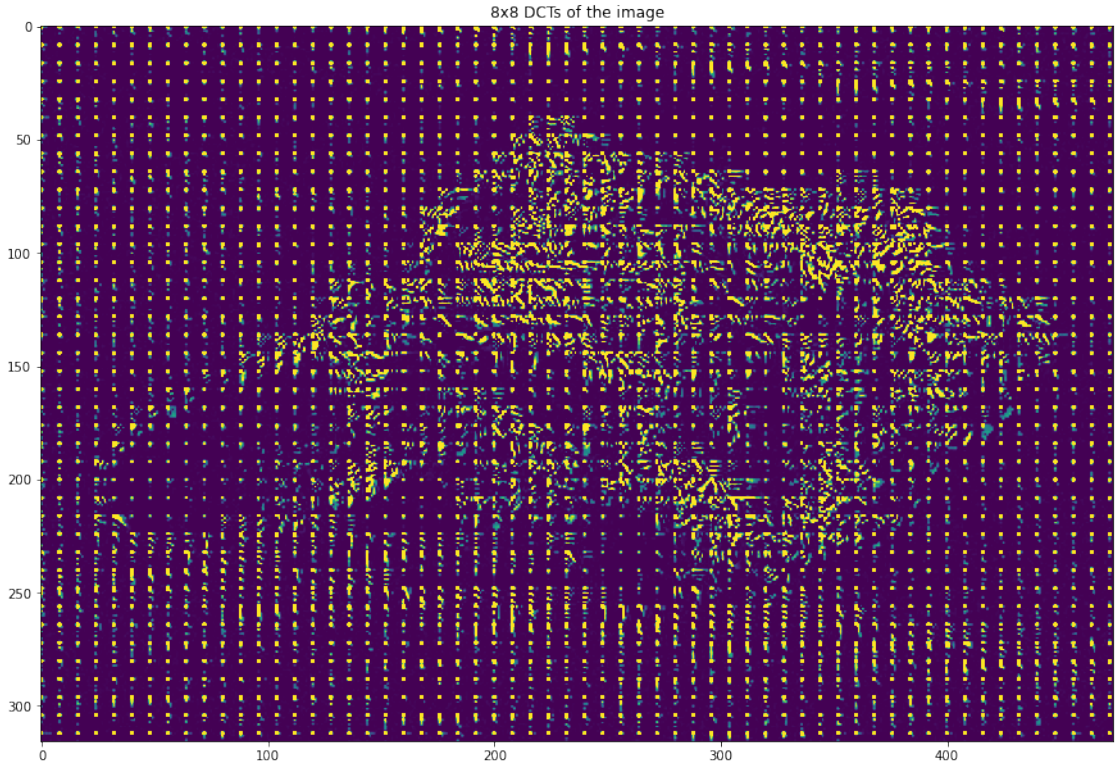




Showing the entire image DCT gives an indication of where high frequency block are within the image. Areas with high density of yellow correspond to areas of change on the motorbike, such as

the rider, wheel rims and nose of the bike. The background, where many parts of the image are almost all one colour, show only the DC component, or a very few low frequency components near the [0,0] position of the matrix.

[23]:
```
plt.figure(figsize=(15,15))
plt.imshow(bike_dct, vmax = np.max(bike_dct)*0.01, vmin = 0)
plt.title( "8x8 DCTs of the image");
```



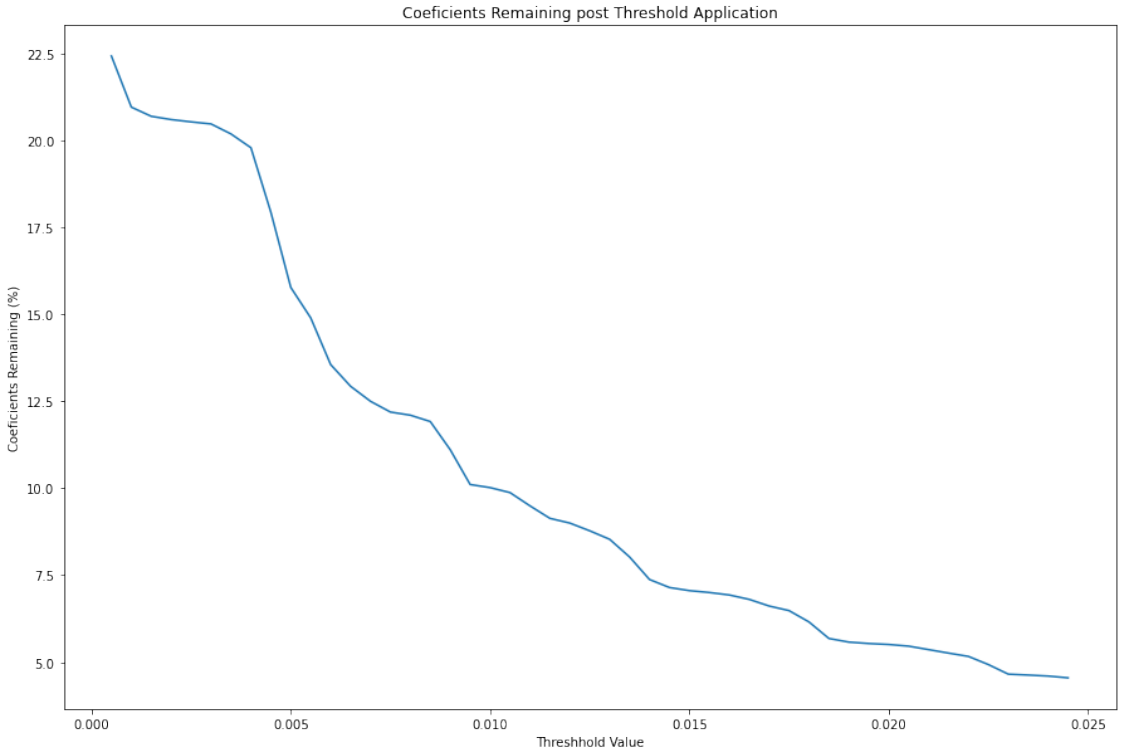
2.1.2 Choose the K-Largest Coeficients

Using a single value threshold allows for an understanding of how much loss occurs within the output image quality relative to the number of values stored. As the threshold is reduced, the percentage of coeficients reduces rapidly, hoever the plateus in the graph below also indicate points where little to no loss can be achieved while still reducing the number of coefficients (and thus storeage size) of the final image.

[24]:
```
x = [x * 0.0005 for x in range(1, 50)]
thresholds = {}
coeficients = {}

for key in x:
    thresholds[key] = apply_threshold(bike_dct, key)
```

```
    coeficients[key] = 100 * np.sum(thresholds[key] != 0.0 ) / (bike.shape[0] *␣
↪bike.shape[1] * 1.0)
```

[25]:
```
plt.figure(figsize=(15,10))
plt.plot(list(coeficients.keys()), list(coeficients.values()))
plt.title('Coeficients Remaining post Threshold Application')
plt.xlabel('Threshhold Value')
plt.ylabel('Coeficients Remaining (%)');
```



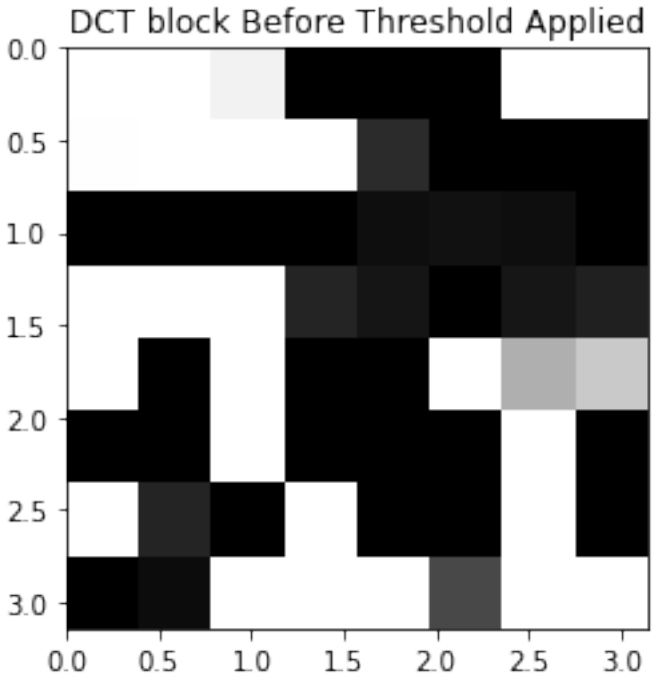
The images below show the final image after coefficients have been removed at different levels. Note that, after removing circa 13% of the image's coefficients (a threshold of 0.005) the image has had a negligible reduction in quality to the human eye. The degredation at higher thresholds (above 0.015) is clear in the high pixellation and visible 8x8 blocks.

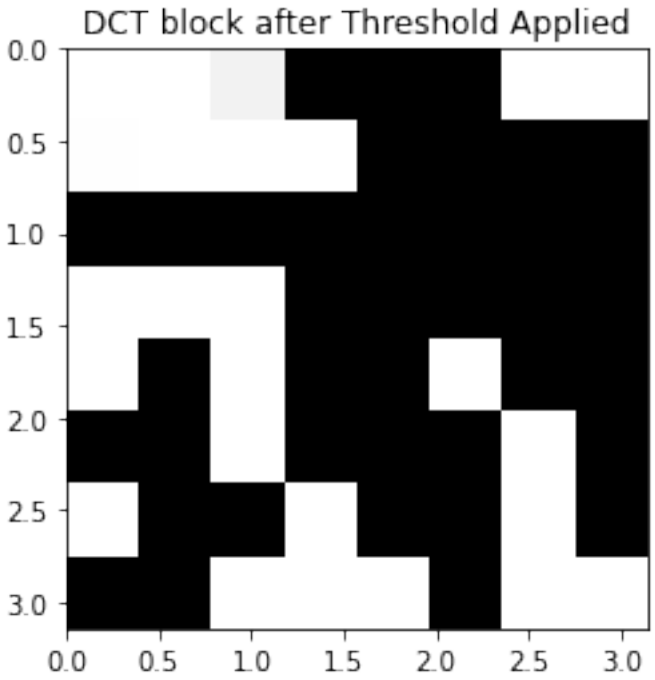
[26]:
```
transformed = idct(apply_threshold(bike_dct, 0.001))

display([bike,
         idct(apply_threshold(bike_dct, 0.005)),
         idct(apply_threshold(bike_dct, 0.01)),
         idct(apply_threshold(bike_dct, 0.015)),
         idct(apply_threshold(bike_dct, 0.02)),
         idct(apply_threshold(bike_dct, 0.025))],
        ['Original',
```

```
         'Threshold = 0.005',
         'Threshold = 0.01',
         'Threshold = 0.015',
         'Threshold = 0.02',
         'Threshold = 0.025'],
        figsize=(15,15),
        cmap='gray')
```

A close up view of one block shows the output before and after the threshold is applied, with the lowest values (in grey) set to zero (shown in black).

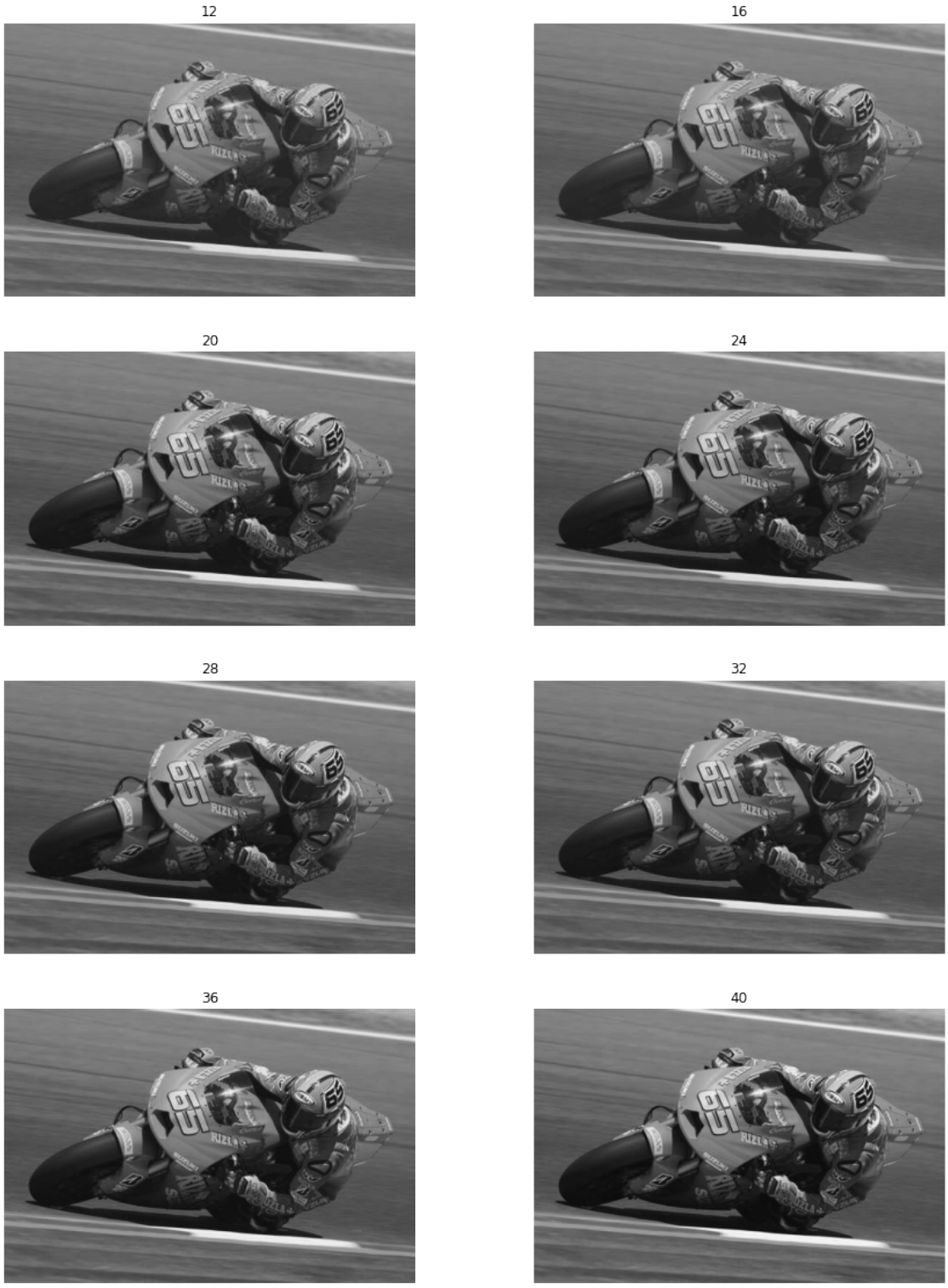
[27]:
```
x_pos = 100
y_pos = 200
block_size=8

bike_dct = dct(bike)

# Display the dct of a block before the threshold is applied
plt.figure()
plt.imshow(bike_dct[x_pos: x_pos + block_size, y_pos: y_pos + block_size],
           cmap='gray',
           vmax=np.max(bike_dct) * 0.01,
           vmin=0,
           extent=[0,pi,pi,0])
plt.title(f"DCT block Before Threshold Applied");

bike_dct_t = apply_threshold(bike_dct, 0.0085)

# Display the dct of the block after the application of the threshold
plt.figure()
plt.imshow(bike_dct_t[x_pos: x_pos + block_size, y_pos: y_pos + block_size],
           cmap='gray',
           vmax=np.max(bike_dct) * 0.01,
           vmin=0,
           extent=[0,pi,pi,0])

plt.title(f"DCT block after Threshold Applied");
```





To encode a watermark in the image, the threshold must be set at a certain value per block. This value was experimentally found by increasing the integer value of coefficients remaining in each block and observing if there was a visible deterioration of the image quality. A value of 32 was selected as a value slightly higher than the human observable "identical" image, due to the fact that some deterioration would be likely while adding the watermark.

[28]:
```
K = [n for n in range(12,42,4)]

images = {}

for k in K:
    images[k] = idct(filter_k_highest(bike_dct, k))

values = list(images.values())
keys = list(images.keys())
display(values, keys, cmap='gray', figsize=(15, 20))
```



[29]:
```
display([bike, idct(filter_k_highest(bike_dct, 42))], ['Original', '42 Highest␣
↪Coefficients'], cmap='gray')
```

2.1.3 Create a watermark

The watermark was created using a gaussian distribution of 42 random numbers with a $\mu = 0$ and $\sigma = 0.1$ These values were experimentally found to be low enough not to cause undue noise within the image while still remaining detectable post watermarking.

[30]:
```
k = 42
mu = 0
sigma = 1
alpha = 0.1

w = np.random.default_rng().normal(size=(k,))
print(w)
```

```
[-0.43601915 -0.52544269 -0.7731568  -0.47052952 -0.79368084  0.2432324
 -0.31785281 -0.6790909  -0.51275442  1.35770306  0.84934923 -0.13257402
  0.66159191 -0.32213324 -0.20068897  3.58616898  0.61432963  1.20517928
  0.27707263  0.02267599 -0.10332188 -0.82513593 -2.64922759  0.44034384
 -1.45546344 -0.37809708  0.21102097 -0.64885948 -1.8550106  -1.14444417
 -1.84902213 -0.81022399  0.45660421  0.32594782  0.18446504 -1.85798111
 -1.05549581 -0.3584888  -1.07422633 -0.96687317 -1.24047131 -0.05716732]
```

2.1.4 Embed a Watermark

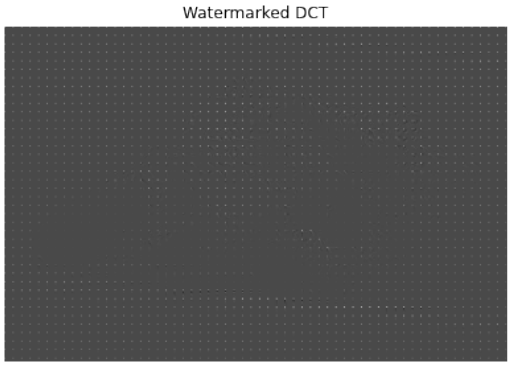
The watermark was embedded by unravelling and sorting the 64 elements of the block in decending order of absolute value. The DC component (at position [0,0]) was removed from the list so that it was not modified by the watermark

[31]:
```
def watermark_block(block, w):

    # Create a copy of the block to keep track of which elements have been used
    oput = np.zeros(block.shape)
    block_abs = np.abs(block)

    # Remove the DC component from the absolute block so that it is not used
```

```
    block_abs[0,0] = 0

    # For every element in the watermark 'w'
    for w_i in w:

        # Select the next highest value and get it's row and column
        r, c = np.unravel_index(np.argmax(block_abs, axis=None), block_abs.
 ↪shape)

        # Set the output to the the sum of the original value and scaled␣
 ↪watermark
        oput[r,c] = block[r,c] * (1 + alpha * w_i)

        # Set the element to 0 so it is not selected again
        block_abs[r,c] = 0;

    # Insert the original DC component into the output
    oput[0,0] = block[0,0]

    return oput


def watermark(img_dct, w, block_size=8):

    oput = np.zeros(img_dct.shape)

    for i in r_[:img_dct.shape[0]: block_size]:
        for j in r_[:img_dct.shape[1]: block_size]:
            oput[i:(i + block_size), j:(j + block_size)] =␣
 ↪watermark_block(img_dct[i:(i + block_size),j:(j + block_size)], w)

    return oput
```

2.1.5 Create the Watermarked DCT

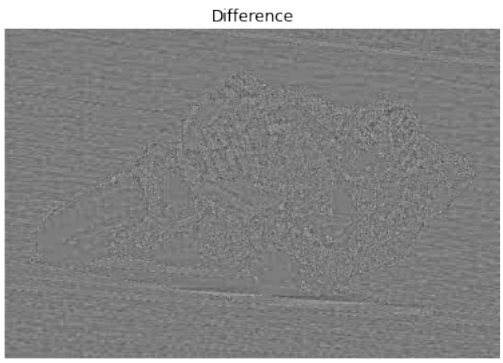
[32]:
```
bike_w = watermark(bike_dct_t, w)
display([bike_w, idct(bike_w)],['Watermarked DCT', 'Watermarked Image'],␣
 ↪cmap='gray')
```



2.1.6 Comparison

The watermarked image, when placed side by side with the original, shows a slight fading where noise has slightly distorted the image, however it is very difficult to detect and would not be noticable if the original was not avialable to make a comparison. The difference taken between the two images shows that there is a clear watermark embedded into the second image.

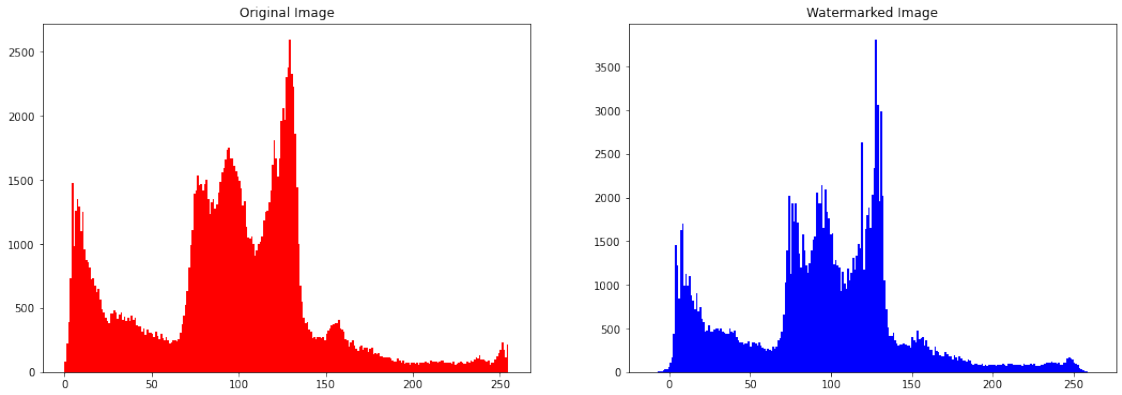
[33]:
```
bike_w = idct(bike_w)
bike_delta = np.subtract(bike, bike_w)
display([bike, bike_w, bike_delta], ['Original Image', 'Watermarked Image',␣
 ↪'Difference'], cmap='gray')
```



2.1.7 Histogram

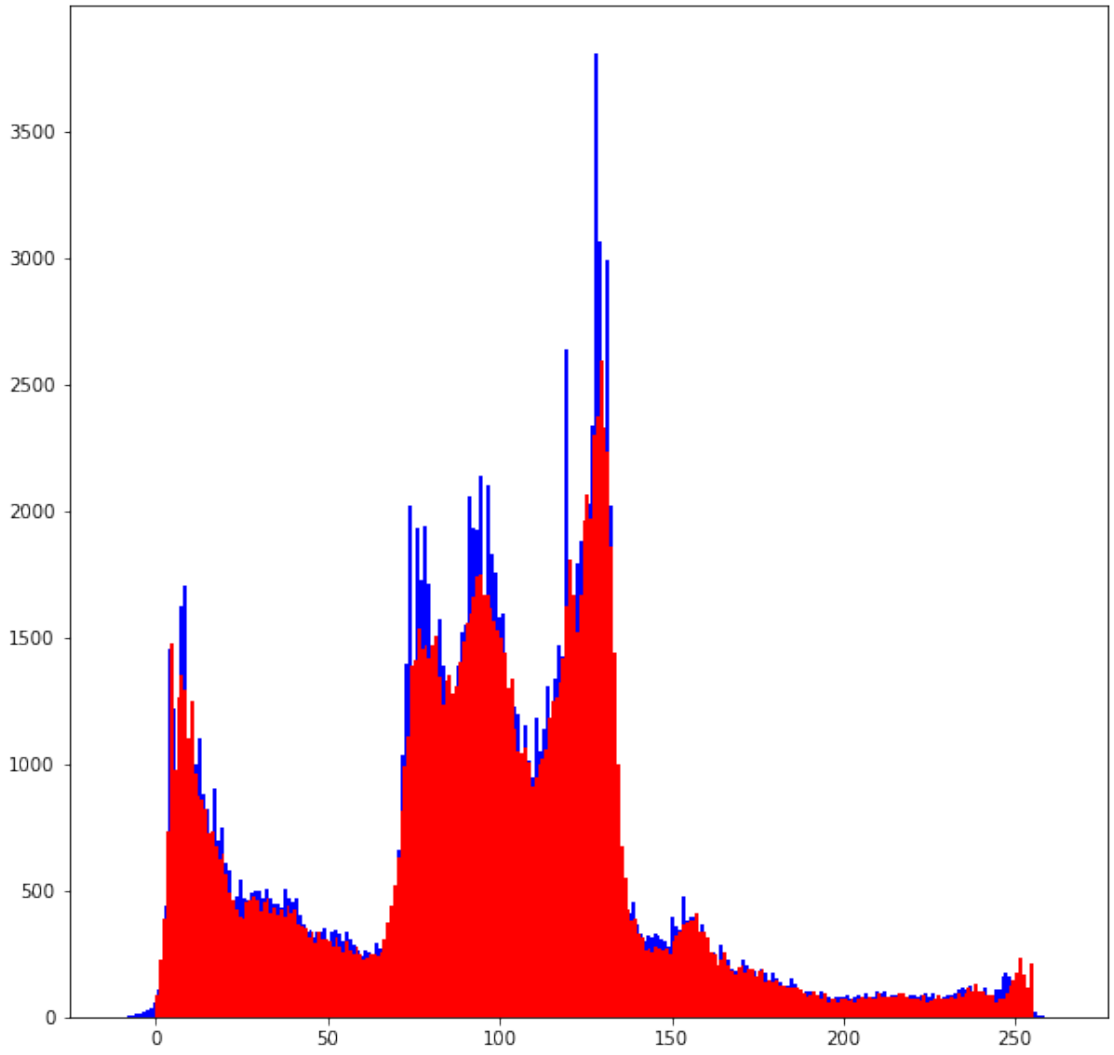
A further comparison can be made between the two images by taking a histogram of all pixel values within the images. The watermarked image has several large spikes that do not show in the orignal, indicating where values have been added to the original when applying the watermark.

[34]:
```
bike_flattened = np.reshape(bike, (bike.shape[0] * bike.shape[1],))
w_flattened = np.reshape(bike_w, (bike_w.shape[0] * bike_w.shape[1],))

fig = plt.figure(figsize=(18,6))
fig.add_subplot(1, 2, 1)
plt.hist(bike_flattened, bins=255, color='red')
plt.title("Original Image")
fig.add_subplot(1, 2, 2)
plt.hist(w_flattened, bins=255, color='blue')
plt.title("Watermarked Image");
```



Superimposing the two histograms on top of eachother gives a clearer indication of the difference between the two images. Note the values for the watermarked image (in blue) stray out of the range $[0-255]$ of the original image. This is likely caused by values close to 0 or 255 being summed with the watermark and going out of range of values for images encoded in 8 bits.

[35]:
```
plt.figure(figsize=(10,10))
plt.hist(w_flattened, bins=255, color='blue')
plt.hist(bike_flattened, bins=255, color='red');
```



2.1.8 DC Coefficient

The Direct Current (DC) coefficient is the [0,0] coefficient, it is the coefficient with no cosine wave signals present; representing the overall colour (or brightness in the case of greyscale images) of the block. The watermark is not added to the DC coefficient as this would have a noticable effect on the overall image quality, especially as it is likely to be one of the K-largest coefficients in almost all blocks. When the DC coefficient is the largest in most cases, the watermarked image will have a reduced (or increased in the case of a negative watermark) overall brightness for all blocks. This would make the watermark far more noticable when compared to the original image.

2.2 Watermark Detection

Watermark detection involves reversing the process outlined above. Once again the K-largest coeficients are indentified and sorted in decending order. This allows for the retrieval of the watermark values in the same order as they were applied to the original. The DC component is excluded from

the watermarked coefficients once again. Two 'mystery' images (taken from the images above) are taken and the DCT of both are computed.

2.2.1 Compute the 2D DCT

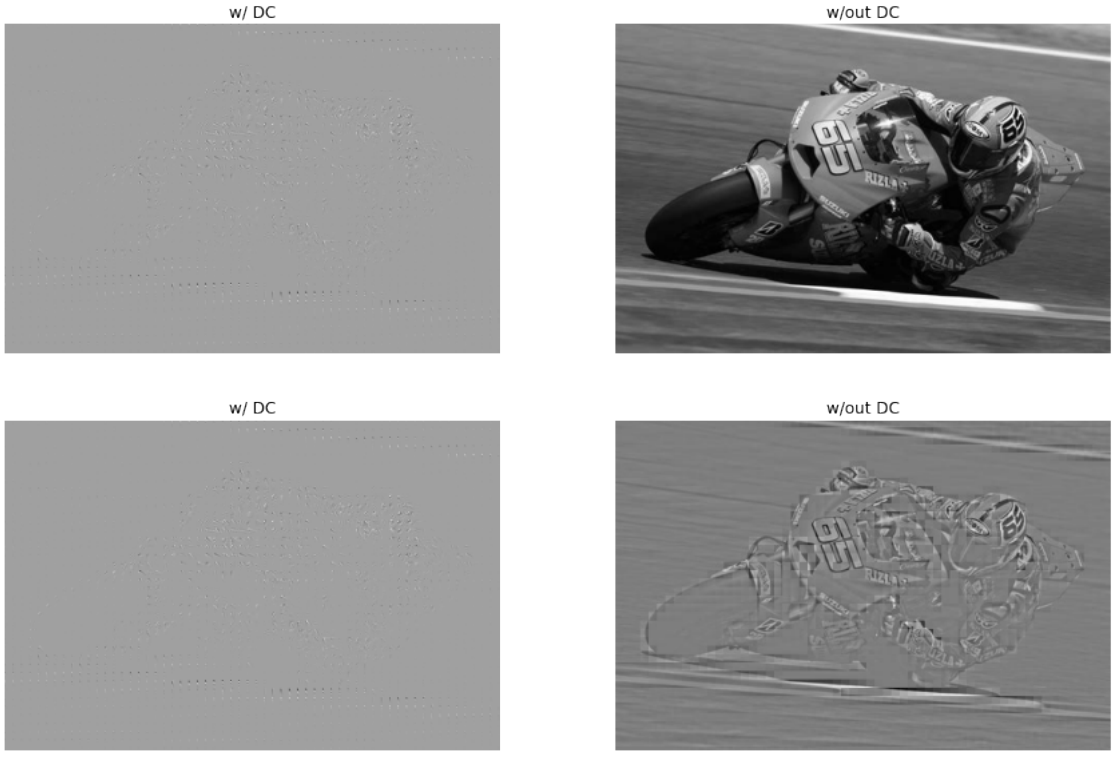
[36]:
```
mystery_1 = bike
mystery_2 = idct(bike_w)
m1_dct = dct(mystery_1)
m2_dct = dct(mystery_2)
```

2.3 Keep the K largest non-DC coefficients

The K largest coefficients are removed with a filter setting them to zero.

[37]:
```
def remove_DC(img_dct):

    oput = img_dct

    for i in r_[:img_dct.shape[0]: block_size]:
        for j in r_[:img_dct.shape[1]: block_size]:

            oput[i, j] = 0

    return oput
```

[38]:
```
dc_less = remove_DC(m1_dct)
display([m1_dct, mystery_1, dc_less, idct(dc_less)],['w/ DC', 'w/out DC', 'w/␣
 ↪DC', 'w/out DC'], cmap='gray')
```



[39]:
```
org_dct_filtered = filter_k_highest(dct(bike_dct), k)
m1_dct_filtered = filter_k_highest(m1_dct, k)
m2_dct_filtered = filter_k_highest(m2_dct, k)
```

Once the blocks have had the K-largest coeficients removed, the difference between the original image blocks, and the watermarked blocks is taken. This difference should reveal the watermark ($\hat{W}$) and can be compared to the inserted watermark (W) formally defined as:

$$\hat{w} = \frac{\hat{c}_i - c_i}{\alpha c_i}, 1 \leq i \leq K$$

γ is defined as the measured similarity of the images, with a threshold T used to decide if an image is the same or not.

$$\gamma = \frac{\sum_{i=1}^{K}(\hat{w}_i - \bar{\hat{w}})(w_i - \bar{w})}{\sum_{i=1}^{K}(\hat{w}_i - \bar{\hat{w}})^2 \sum_{i=1}^{K}(\hat{w}_i - \bar{w})^2}$$

Thus the decision if a watermark is detected can be defined as 1 (image detected) or 0 (image not detected):

$$D = \begin{cases} 1, & \text{if } \gamma \geq T \\ 0, & \text{else} \end{cases}$$

[40]:
```
def compare_block(c_hat, c, k):

    w_hat = []
    c_hat = np.abs(c_hat)
    c_act = np.abs(c)

    c_hat[0, 0] = 0

    for i in range(k):
        r, c = np.unravel_index(np.argmax(c_hat, axis=None), c_hat.shape)

        denom = (alpha * c_act[r, c])
        numer = (c_hat[r, c] - c_act[r, c])

        if denom != 0:
            w_hat.append(numer/denom)
        else:
            w_hat.append(0)

        c_hat[r, c] = 0

    return w_hat
```

[41]:
```
def gamma(img_a, img_b, w, k, block_size=8):

    w_hat = []

    for i in r_[:img_a.shape[0]: block_size]:
        for j in r_[:img_a.shape[1]: block_size]:

            c_hat = img_a[i:(i + block_size),j:(j + block_size)]
            c = img_b[i:(i + block_size),j:(j + block_size)]

            w_i = compare_block(c_hat, c, k)
            w_hat.append(w_i)

    mu_w = np.mean(w)
    mu_w_hat = np.mean(w_hat)

    delta_w = np.subtract(w, mu_w)
    delta_w_hat = np.subtract(w_hat, mu_w_hat)

    numer = np.sum(np.multiply(delta_w_hat, delta_w))
    denom = np.sqrt(np.sum(np.multiply(np.power(delta_w_hat, 2), np.
 ↪power(delta_w, 2))))

    if denom == 0:
```

```
        return 0

    return numer / denom
```

As the γ value for mystery image 1 is below the treshold of 0.5, this can be taken that the image does not contain a watermark due to the similarity. The γ value for the second mystery image is above the threshold thus has a detected threshold.

[42]:
```
print(gamma(org_dct_filtered, m1_dct_filtered, w, k))
print(gamma(org_dct_filtered, m2_dct_filtered, w, k))
```

```
0.39324989167218
1.72260486454297
```

3 Exercise 3 - Morphology

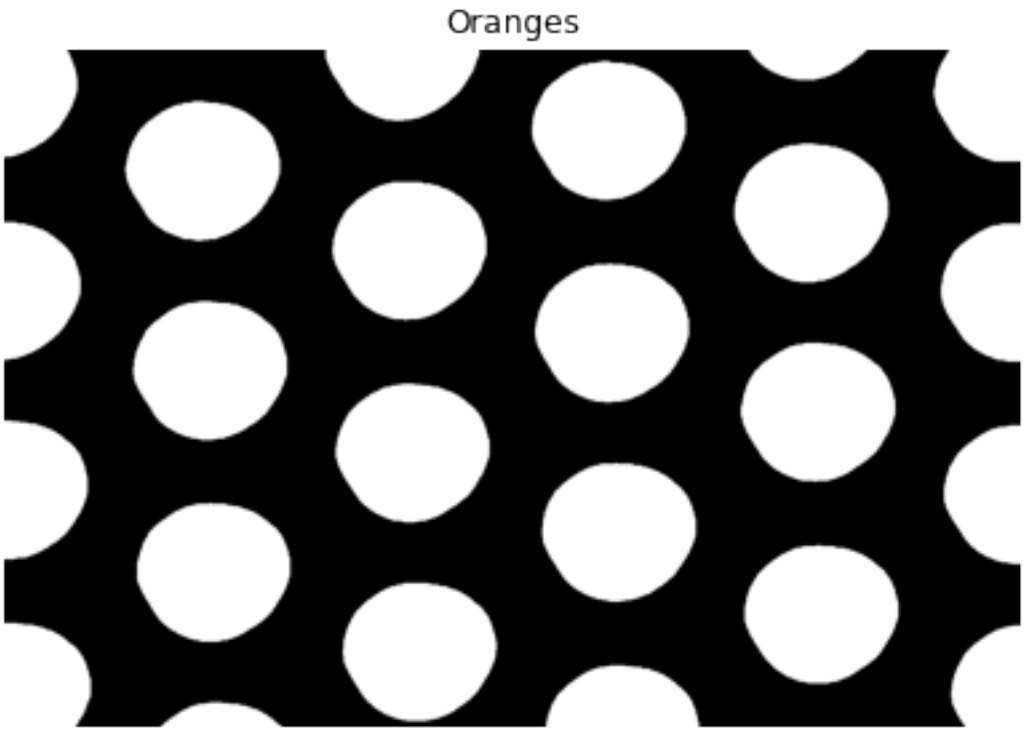
3.1 Count Oranges

[43]:
```
oranges = cv2.cvtColor(cv2.imread('images/oranges.jpg'), cv2.COLOR_BGR2RGB)
tree = cv2.cvtColor(cv2.imread('images/orangetree.jpg'), cv2.COLOR_BGR2RGB)
display([oranges, tree],['Oranges', 'Tree'])
```

[44]:
```
def binary_c(channel, threshold):
    '''Runs a binary threshold on the channel and returns the binary output'''
    thresh, grey = cv2.threshold(channel, threshold, 255, cv2.THRESH_BINARY)
    return grey


def binary(img, t_1, t_2, t_3):
    '''Binarizes each channel and then returns the maximum channel'''
    c_1 = binary_c(img[:, :, 0], t_1)
    c_2 = binary_c(img[:, :, 1], t_2)
    c_3 = binary_c(img[:, :, 2], t_3)
    return np.maximum(c_1, c_2, c_3)
```

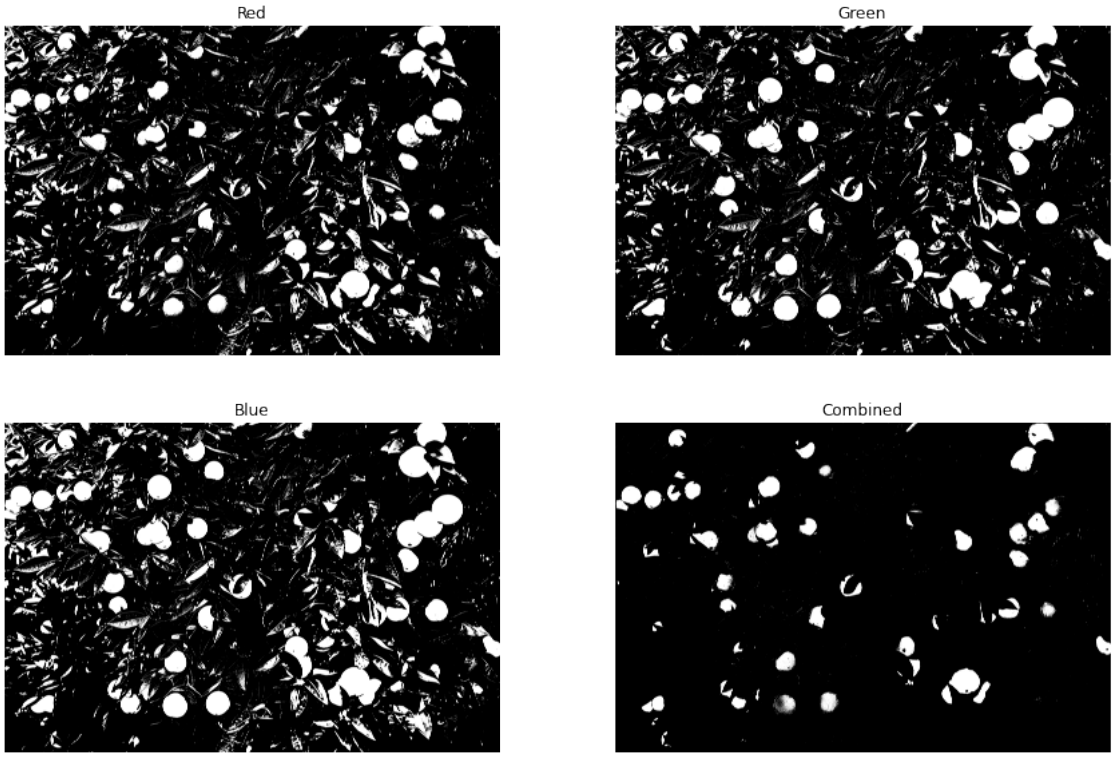
The first step to counting the oranges is to binarize the input with a threshold to identify the key colour channel for the image. In the case of the oranges the same threshold was used and a clear balck and white image shows where the colour differences occured in the image.

[45]:
```
display([binary(oranges, 100, 100, 100)],['Oranges'], cmap='gray')
```



The image of oranges "in their natural habitat" was less simple; through imprical testing a combination of thresholds was found that seperated the oranges reasonably clearly from the tree.

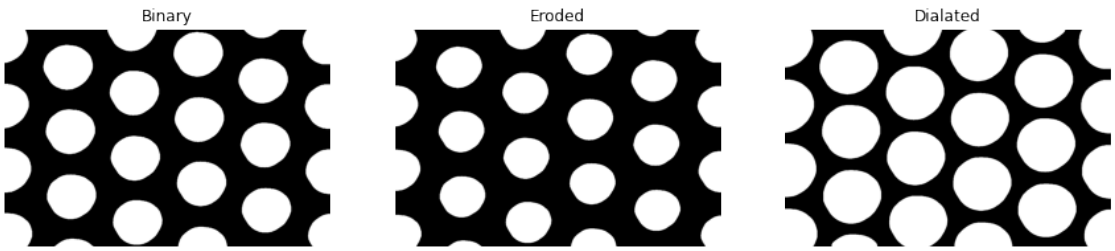
[46]:
```
display([binary(tree, 255, 100, 100),
         binary(tree, 100, 255, 100),
         binary(tree, 100, 100, 255),
         binary(tree, 205, 255, 255)],
        ['Red', 'Green', 'Blue', 'Combined'], cmap='gray')
```



[47]:
```
erode = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (15, 15))
dialate = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (50, 50))
```

Through the use of dialation followed by erosion kernels the images underwent a closing operation, this removed the small sub-sections of the image where oranges had been covered by leaves and left clearer, less noisy contours.

[48]:
```
oranges_bin = binary(oranges, 100, 100, 100)
oranges_eroded = cv2.morphologyEx(oranges_bin, cv2.MORPH_ERODE, erode)
oranges_dialated = cv2.morphologyEx(oranges_eroded, cv2.MORPH_DILATE, dialate)

display([oranges_bin, oranges_eroded, oranges_dialated],
        ['Binary', 'Eroded', 'Dialated'], cmap='gray', cols=3)
```



To count the elements in the image, the image was treated as a set of connnected graphs, where each pixel has at most 8 neighbours (less for those on the edges) which could either be zero or non-zero. Using a depth first search algorithm, every vertex in the connected graph was found, its indices added to the frontier of explored vertices and set to "visited" by setting the value to zero. Each of its non-zero neighbours were subsequently appended to the frontier until the frontier was exhausted.

[49]:
```
def remove_blob(img, coord):
    '''Sets pixel at [x,y] and all connected neigbours to zero'''
    w, h = img.shape
    connected = [coord]

    while len(connected) > 0:

        x, y = connected.pop()

        if img[x, y] != 0:

            img[x, y] = 0

            if 0 <= x+1 < w and img[x+1, y] != 0:
                connected.append((x+1, y))

            if 0 <= x-1 < w and img[x-1, y] != 0:
                connected.append((x-1, y))

            if 0 <= y+1 < h and img[x, y+1] != 0:
                connected.append((x, y+1))

            if 0 <= y-1 < h and img[x, y-1] != 0:
                connected.append((x, y-1))

            if 0 <= x+1 < w and 0 <= y+1 < h and img[x+1, y+1] != 0:
                connected.append((x+1, y+1))

            if 0 <= x+1 < w and 0 <= y-1 < h and img[x+1, y-1] != 0:
                connected.append((x+1, y-1))

            if 0 <= x-1 < w and 0 <= y+1 < h and img[x-1, y+1] != 0:
                connected.append((x-1, y+1))

            if 0 <= x-1 < w and 0 <= y-1 < h and img[x-1, y-1] != 0:
                connected.append((x-1, y-1))

    return img
```

To count the number of white "blobs" in the image, the non-zero indicies were found as a pair of (x, y) tuples. The first pair was selected and the blob removed before the next set of indicies which

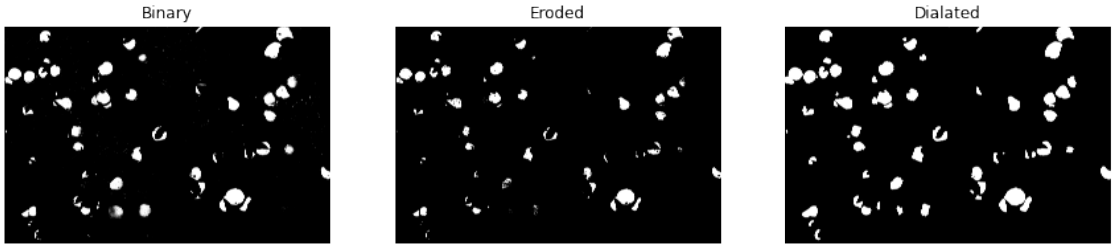
were still not zero were used

[50]:
```
def count_blobs(img):
    '''Counts the white blobs in the image'''
    count = 0
    X, Y = np.nonzero(img)

    while len(X) > 0:
        x = X[0]
        y = Y[0]

        img = remove_blob(img, (x, y))
        count += 1
        X, Y = np.nonzero(img)

    return count
```

[51]:
```
print(f"There are {count_blobs(oranges_dialated)} objects in the image")
```

```
There are 24 objects in the image
```

[52]:
```
tree_bin = binary(tree, 205, 255, 255)
tree_eroded = cv2.morphologyEx(tree_bin, cv2.MORPH_ERODE, erode)
tree_dialated = cv2.morphologyEx(tree_eroded, cv2.MORPH_DILATE, dialate)

display([tree_bin, tree_eroded, tree_dialated],
        ['Binary', 'Eroded', 'Dialated'], cmap='gray', cols=3)
```

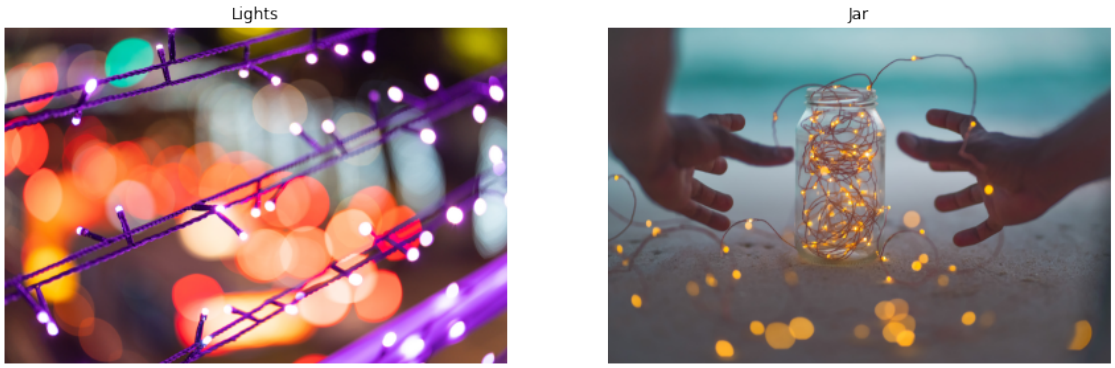


[53]:
```
print(f"There are {count_blobs(tree_dialated)} objects in the image")
```

```
There are 60 objects in the image
```

3.2 3.2 - Granulometry

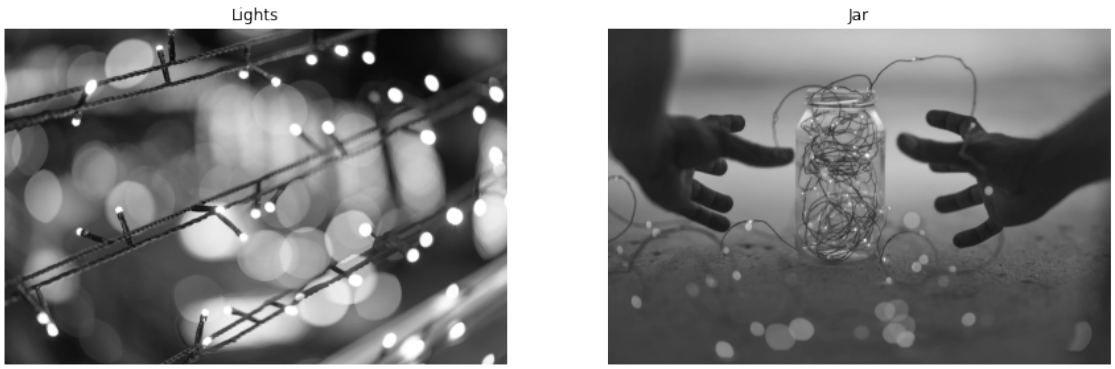
[54]:
```
lights = cv2.cvtColor(cv2.imread('images/lights.jpg'), cv2.COLOR_BGR2RGB)
jar = cv2.cvtColor(cv2.imread('images/jar.jpg'), cv2.COLOR_BGR2RGB)
display([lights, jar],['Lights', 'Jar'])
```



3.2.1 3.2.1 - Pre-processing

The images were converted to greyscale in the pre-processing step.

[55]:
```
lights_g = cv2.cvtColor(lights, cv2.COLOR_BGR2GRAY)
jar_g = cv2.cvtColor(jar, cv2.COLOR_BGR2GRAY)
display([lights_g, jar_g],['Lights', 'Jar'], cmap='gray')
```



Granulometry works through successive erotion and dialation steps at incrementally increasing diameters such that, in the same way that using successively smaller sieves can be used to seperate sand and rocks into different sizes. A starting diameter and increment factor were chosen, with each iteration the diameter was increased by the factor.

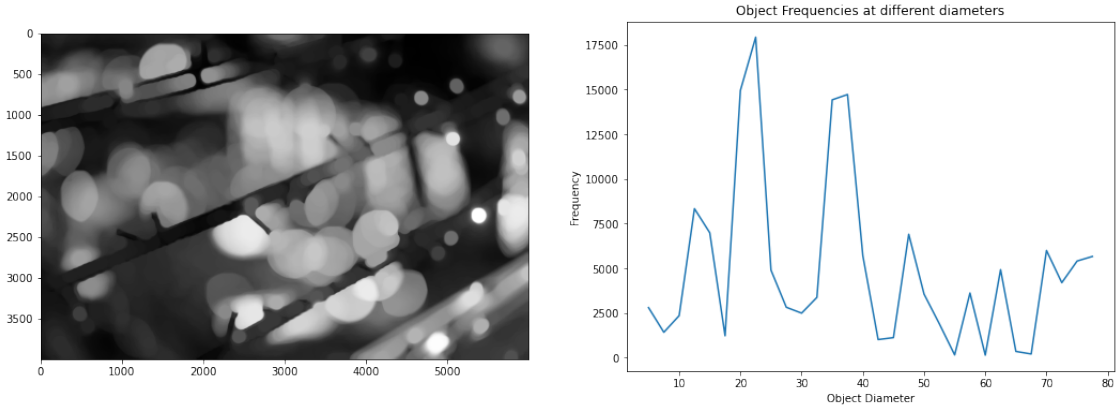
At each stage the surface area (calculated by taking the sum of all of the diameters within the filter) is taken and compared to the previous surface area. This allows for the calculation of the difference and recording of the frequencies encountered in the image.

[56]:
```
def granulometry(img, d_start, factor, iterations):

    retval = []
    sa = sum(sum(img))
    diameters = [d_start + (d * factor) for d in range(iterations)]

    for d in diameters:
        # Create erode and dialate elements at diameter d
        erode = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (d, d))
        dialate = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (d, d))

        # Open the original image using erosion and dialation
        opening = cv2.morphologyEx(img, cv2.MORPH_ERODE, erode)
        opening = cv2.morphologyEx(opening, cv2.MORPH_DILATE, dialate)
        img = opening

        # Calculate the surface area of the image
        new_sa = sum(sum(img))

        # Compare the surface area to the previous surface area
        retval.append([d / 2, abs(sa - new_sa)])
        sa = new_sa

    return opening, np.asarray(retval)
```

[57]:
```
lights_hc, lights_fq = granulometry(lights_g, 10, 5, 30)
```

When the granulometry is run on the image of lights, spikes can be seen in the ouput of frequencies around 20 and 35px diameter. From this it is hypothesised that the lights themselves are around 20 - 35px in diameter

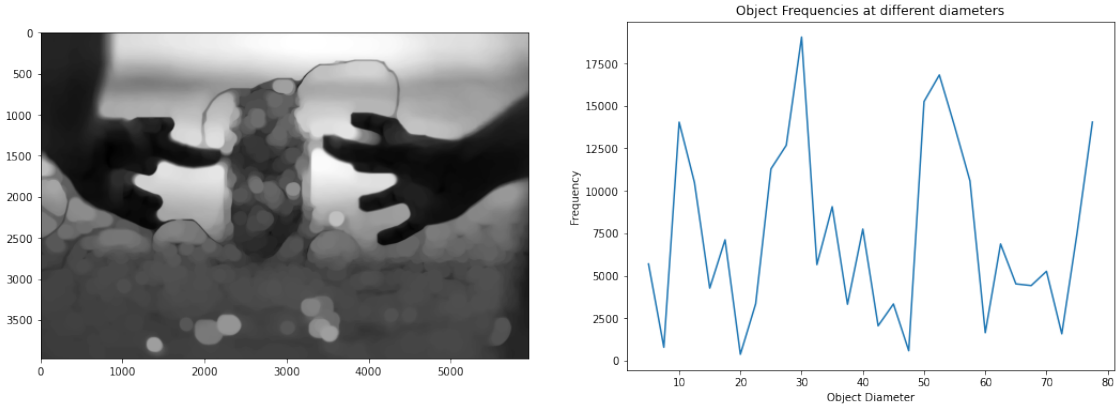
[58]:
```
fig = plt.figure(figsize=(18,6))
fig.add_subplot(1, 2, 1)
plt.imshow(lights_hc, cmap='gray');

fig.add_subplot(1, 2, 2)
plt.plot(lights_fq[:,0], lights_fq[:,1])
plt.title('Object Frequencies at different diameters')
plt.xlabel('Object Diameter')
plt.ylabel('Frequency');
```



[59]:
```
jar_hc, jar_fq = granulometry(jar_g, 10, 5, 30)
```

In contrast, the image of the jar has spikes at frequencies 10, 30 and 55, these are expected to correspond to the lights in the jar, those in the mid ground and those in the foreground respectively. It should be noted that the finders in the picture may also have been counted, due to the degredation of the final output image.

[60]:
```
fig = plt.figure(figsize=(18,6))
fig.add_subplot(1, 2, 1)
plt.imshow(jar_hc, cmap='gray');

fig.add_subplot(1, 2, 2)
plt.plot(jar_fq[:,0], jar_fq[:,1])
plt.title('Object Frequencies at different diameters')
plt.xlabel('Object Diameter')
plt.ylabel('Frequency');
```



4 4 - Principal Component Analysis

Prior to the execution of this exercise, front on images of actresses heads were selected, resized to the same size and each image's eyes were aligned to the same point. Four images of the same actress were selected to allow for comparison of the effects of different hair styles and makeup.

4.1 4.1 - Eigenfaces

The following steps should be taken to conduct principle component analysis:

1. Format the data into a matrix of points where each is a vector [x, y, z] (M_data)
2. Calculate the Mean vector (x_mean)
3. Subtract Mean from data matrix (M = M_data - x_mean)
4. Calculate the Covariance matrix (C = M * M ^T)
5. Calculate the Eigen vectors and Eigen values of the covariance matrix

The steps have been outlined below in code, however they have been repeated using the cv2 inbuilt function due to the size of the image.

[61]:
```
import glob

faces = []

for filename in glob.glob('faces_aligned/face2*.jpeg'):
    faces.append(cv2.cvtColor(cv2.imread(filename), cv2.COLOR_BGR2RGB))

height, width, channels = faces[0].shape

print(f"Images have dimensions:  height = {height}px , width = {width}px,␣
 ↪channels = {channels}")
display(faces, [i for i in range(len(faces))], cols=4)
```

```
Images have dimensions:  height = 900px , width = 600px, channels = 3
```

The steps outlined above were translated into code, however the final outputs used OpenCV's PCACompute function to calculate the eigen vectors due to the increased speed and reduced space complexity of the inbuilt function.

[62]:
```
# 1. Format the data into a matrix of points where each image is a vector of m
 ↪* n (900 * 600)
lst = []

for face in faces:
    m, n, c = face.shape
    lst.append(np.reshape(face, m * n * c))

M_data = np.stack(lst, axis=0)
M_data = M_data / 255


# 2. Calculate the Mean vectors
mean = np.mean(M_data, axis=0)


# 3. Subtract Mean from data matrix  (M = M_data - x_mean)
M_data = np.subtract(M_data, mean)
M_data.shape


# 4. Calculate the Covariance matrix  (C = M * M ^T)
C = np.dot(M_data, M_data.T)

# 5. Calculate the Eigen vectors and Eigen values of the covariance matrix
m, eigen_vectors = np.linalg.eig(C)
```

[63]:
```
m, eigen_vectors = cv2.PCACompute(M_data, mean=None, maxComponents=10)

print(f"Mean vector shape: {mean.shape}")
print(f"Mean vector max: {np.max(mean)}")
print(f"Mean vector min: {np.min(mean)}\n")

print(f"Eigen Vectors shape: {eigen_vectors.shape}")
print(f"Eigen Vectors max: {np.max(eigen_vectors)}")
print(f"Eigen Vectors min: {np.min(eigen_vectors)}\n")
print(eigen_vectors)
```

```
Mean vector shape: (1620000,)
Mean vector max: 0.9980392156862745
Mean vector min: 0.05784313725490196

Eigen Vectors shape: (4, 1620000)
Eigen Vectors max: 0.0033377415196004074
```

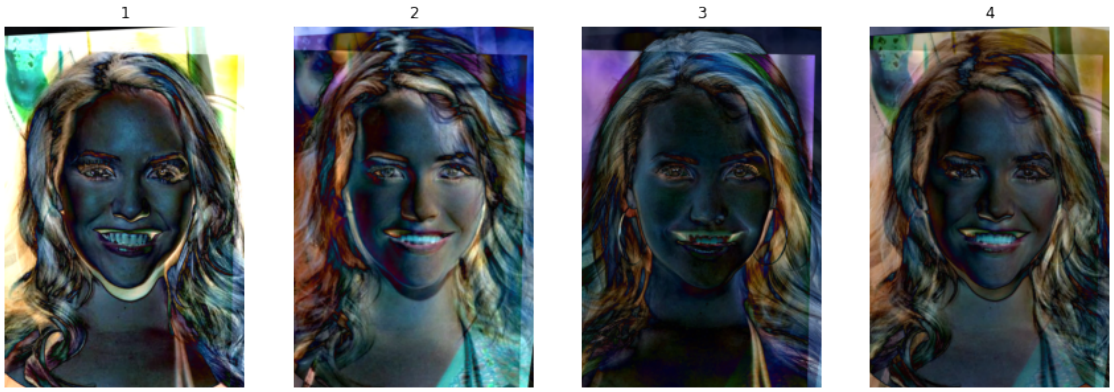
```
Eigen Vectors min: -0.003257387817442098

[[ 4.18326433e-05  4.32575716e-05  4.75323566e-05 … -3.57166732e-04
  -4.02204408e-04 -5.43492156e-04]
 [-9.40057026e-04 -9.91174261e-04 -1.14452597e-03 … -4.96340339e-05
  -7.02549561e-05 -1.89008580e-05]
 [ 3.03780814e-04  3.23689958e-04  3.83417392e-04 … -1.19043452e-03
  -1.83596152e-03 -2.33403563e-03]
 [-5.06741962e-04 -5.30968670e-04 -6.05667684e-04 … -1.31732721e-04
  -3.09899965e-04 -3.30088888e-04]]
```

4.1.1 4.1 - Show the resulting eigenfaces and explain their apperance

The eigen faces, giving the difference between the mean face and the face they represent show very small values. When normalised to the maximum value it is clear that each eigenface shows a combination of differences between the mean and the image they represent.

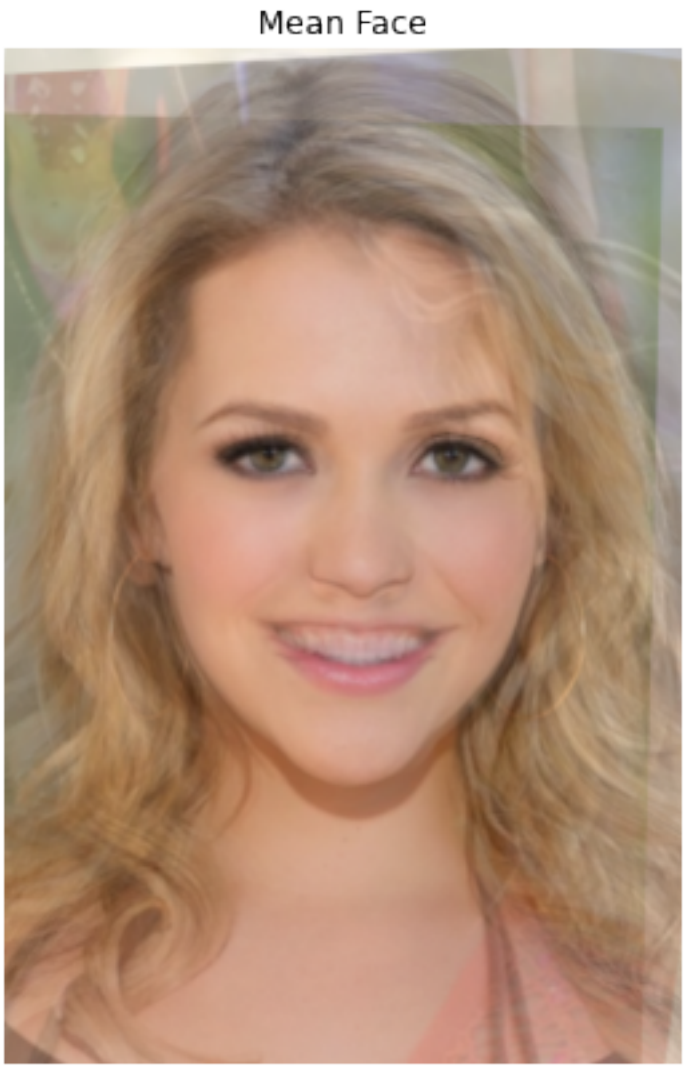
[64]:
```
# Reshape EigenVectors to obtain EigenFaces
eig_faces = [np.reshape(e, (height, width, channels)) for e in eigen_vectors]

print(f"There are {len(eig_faces)} eigen faces")

display([ np.abs(f) / np.max(f) for f in eig_faces], [i+1 for i in␣
 ↪range(len(faces))], cols=4)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).

There are 4 eigen faces
```



[65]:
```
mean_face = np.reshape(mean, (height, width, channels))
```

```
display([mean_face],['Mean Face'], figsize=(10,15))
```

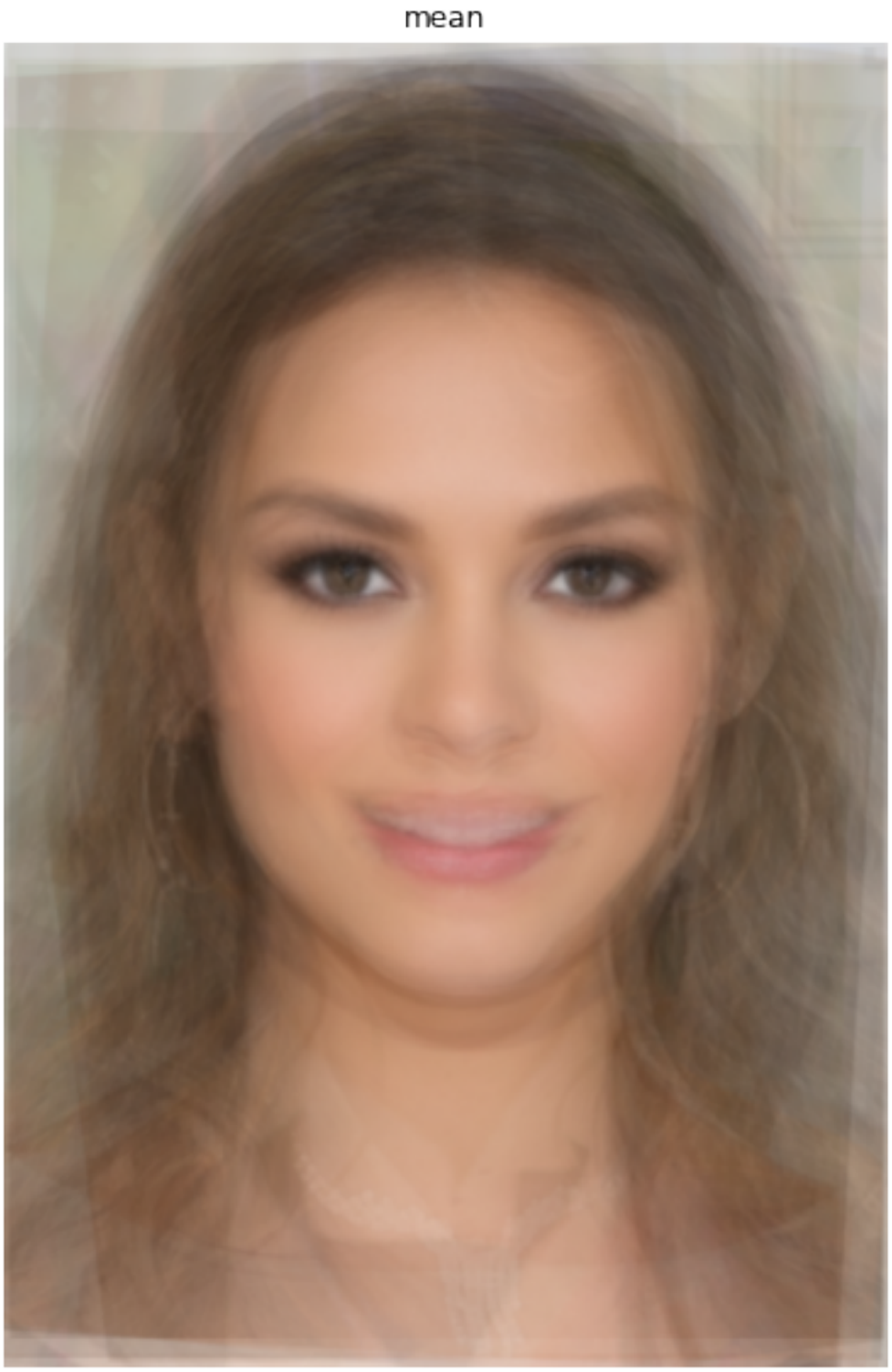


[66]:
```
def pca(images):
    # 1. Stack images into a matrix2
    lst = []

    for img in images:
        m, n, c = img.shape
        lst.append(np.reshape(img, m * n * c))

    data = np.stack(lst, axis=0)
    data = data / 255

    # 2. Calculate the Mean vectors
```

```
    mean = np.mean(data, axis=0)

    # 3. Subtract Mean from data matrix  (M = M_data - x_mean)
    data = np.subtract(data, mean)

    # 4. Calculate the Eigen vectors and Eigen values
    m, eigen_vectors = cv2.PCACompute(data, mean=None, maxComponents=10)

    # 5. Reshape mean face back to image
    mean_face = np.reshape(mean, (height, width, channels))

    eig_faces = [np.reshape(e, (height, width, channels)) for e in␣
↪eigen_vectors]

    return mean_face, eig_faces
```

[67]:
```
images = []
for filename in glob.glob('faces_aligned/*.jpeg'):
    images.append(cv2.cvtColor(cv2.imread(filename), cv2.COLOR_BGR2RGB))

display(images, [i for i in range(len(images))], cols=5)
```

[68]:
```
mean_face, eig_faces = pca(images)

display([mean_face], ['mean'])
```



Due to the prevalance of caucasian faces in the dataset, the mean face is biased to a light skin tone. If an eigenface of a principal component with a darker skin tone, the mean face takes on the atributes of the faces with darker features.

[69]:
```
output = mean_face
output = np.add(output, np.multiply(eig_faces[3], 150))
display([output],['Reconstructed Face'], figsize=(10,15))
```

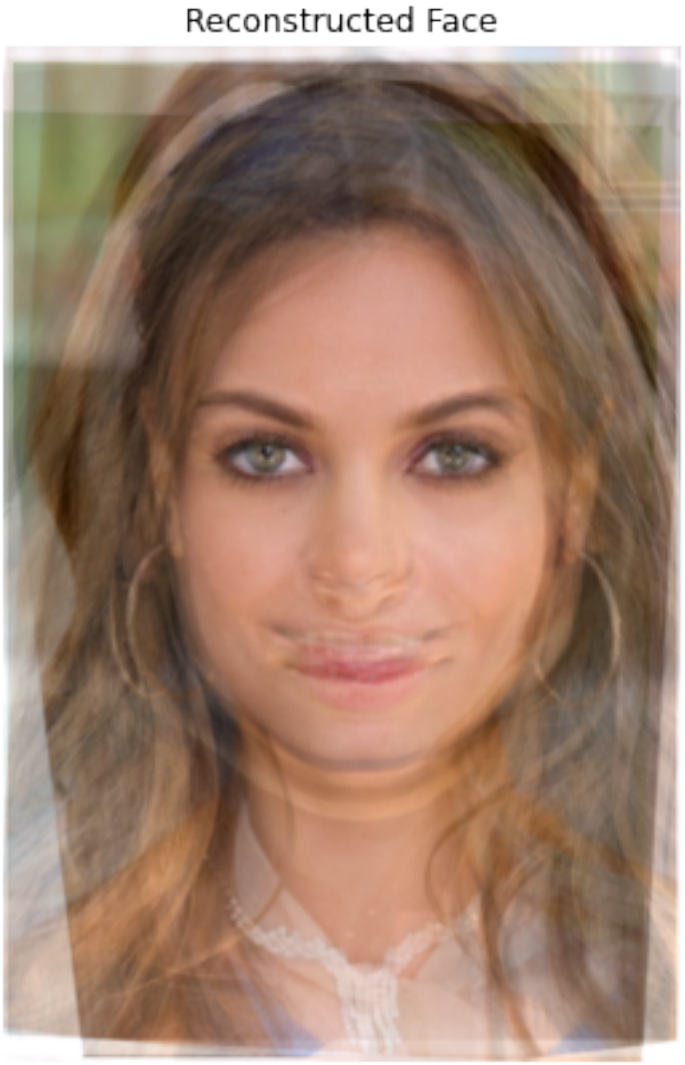
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

4.1.2 4.2.1 - Reconstruct Faces with all Eigenfaces

Reconstructing the mean face with all eigenfacers superimpsoes all of the differences over one another. This creates a terrifying distorted face.

[70]:
```
output = mean_face

weights = [50 for i in range(len(eig_faces))]

for i, weight in enumerate(weights):
    output = np.add(output, np.multiply(eig_faces[i], weight))

display([output],['Reconstructed Face'], figsize=(10,15))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



4.1.3 4.2.2 - Reconstruct Faces with few Eigenfaces

The eigenfaces can be used to select certain features to create new faces. For example, by selecting the first and second eigenfaces, an image can be created with the mean face with blonde hair from one actress and green earrings from image two, where the actress has brown hair.

[71]:
```
output = mean_face

weights = [0 for i in range(len(eig_faces))]
weights[0] = 100
weights[1] = 200

for i, weight in enumerate(weights):
    output = np.add(output, np.multiply(eig_faces[i], weight))

display([output],['Reconstructed Face'], figsize=(10,15))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

Reconstructed Face