Image and Video Processing
Project 2: Spatial Filtering, Processing

Table of Contents

Image degradation with motion blur and additive noise

For this task, diagonal motion blur and additive noise will be added to the frequency domain of a coloured image.
The end result is a noisy image FT, which will look like the following equation.

$$G(u,v) = F(u,v)H(u,v) + N(u,v)\text{ , where}$$

$F(u, v)$: is the original image Fourier Transform (FT)
$H(u, v)$: is the motion blur filter
$N(u,v)$: is the noise FT

Creating a Horizontal Motion Blur

The diagonal motion blurring function is defined as:

$$H(u, v) = sinc(\alpha \cdot u + \beta \cdot v) \cdot exp(-j\pi(\alpha \cdot u + \beta \cdot v))\text{ , where}$$

$sinc(x) = sin(x)/x$
α: amount of vertical blur
β: amount of horizontal blur

Since the image used is in colour and not grayscale, its size includes the 3 RGB colour channels. Operations have to be performed for each channel.

```
(ch1, ch2, ch3) = cv2.split(img)
ch1_b = blur_image(ch1, 0, 0.2)
ch2_b = blur_image(ch2, 0, 0.2)
ch3_b = blur_image(ch3, 0, 0.2)
blurred = cv2.merge((ch1_b, ch2_b, ch3_b))
```

Thus, each channel of the original image is passed in a method called blur_image() and transformed into the Fourier domain:

```
def blur_image(channel, a, b ):
   w = channel.shape[0]
   h = channel.shape[1]
   [u, v] = np.mgrid[-w / 2:w / 2, -h / 2:h / 2]
   u = 2 * u / w
   v = 2 * v / h
# Fourier Transform
   F = np.fft.fft2(channel)
# Motion blur
   H = np.sinc((u * a + v * b)) * np.exp(-1j * np.pi * (u * a + v * b))
# New image original x motion blur
   G = F*H
# Inverse Fourier transform
   g = np.fft.ifft2(G)
```

```
    channel_blur = np.abs(g) / 255
    return channel_blur
```

As it can be observed from the code above, the $F(u,\ v)$ is achieved by using the function numpy.fft.fft2, which computes the 2-dimensional discrete Fourier transform.
To create a realistic motion blur, only a horizontal blur was applied to the image, thus leaving $\alpha\ =\ 0$.

Original Image

Horizontal Motion Blur

Adding Noise to the blurred image

Gaussian noise, which in the formula is $N(u,\ v)$ is then added to the blurred image $F(u,\ v)H(u,v)$.

```
def gaussian_noise(img):
    noise = random_noise(img, 'gaussian', mean=0, var=0.02)
    return noise
```

This is performed by generating random numbers that follow the Gaussian Distribution, with a mean = 0 and variance = 0.02. The random numbers are stored within an array of the same size as the image.

Noisy and Blurry Image

Removing Motion Blur (Inverse Blur Filtering)

As previously discussed, to add motion blur to an image we multiply. Thus, to remove that same blur, we can divide the FT of the original image by the motion blur filter added.

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

This time, however $F(u, v)$ is not the original image, but a degraded image.

Blurred Image (1)

Inverse Blur Filtering (1)

Noisy and Blurred Image (2)

Inverse Blur Filtering (2)

Minimum Mean Square Error Filtering (MMSE)

MMSE or Weiner filtering is an image restoration technique, that can be applied in this case to restore a noisy image.
The Weiner Filter Transfer function is:

$$H_W(u,v) = \frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + S_n(u,v)/S_f(u,v)}$$

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

- $H(u, v)$ = degradation fuction, $H^*(u, v)$ = complex conjugate
- $|H(u, v)|^2 = H(u, v)H^*(u, v)$
- $S_n(u, v) = |N(u, v)|^2$ = noise power spectrum
- $S_f(u, v) = |F(u, v)|^2$ = original image power spectrum

Also, in this case, each channel of the image has to be transformed.

```
(o_ch1, o_ch2, o_ch3) = cv2.split(img)
(noise_ch1, noise_ch2, noise_ch3) = cv2.split(g_noise)
Fhat_1 = mmse(o_ch1, noise_ch1, 1)
Fhat_2 = mmse(o_ch2, noise_ch2, 1)
Fhat_3 = mmse(o_ch3, noise_ch3, 1)
```

For convenience, a method called **mmse()** was created, which returns $\widehat{F}(u, v)$. This takes a channel of the original image, the corresponding channel in its degraded form, and the degradation function, $H(u, v)$, which in this case will be 1.

```
def mmse(original_c, degraded_c, h):
    o = np.fft.fftshift(np.fft.fft2(original_c))
    s_f = np.abs(o) ** 2
    s_n = np.abs(np.fft.fftshift(np.fft.fft2(degraded_c))) ** 2
    denominator = np.abs(h)**2 + (s_n/s_f)
    Hw = np.conj(h) / denominator
    Fhat = Hw * o
    return Fhat
```

The Inverse Fourier Transfer is then applied to the $\widehat{F}(u, v)$ of each channel, which are then merged to create the restored image.

```
result1 = np.abs(np.fft.ifft2(Fhat_1))
result2 = np.abs(np.fft.ifft2(Fhat_2))
result3 = np.abs(np.fft.ifft2(Fhat_3))
result = cv2.merge((result1,result2,result3))
result = result/np.max(result)
```

Denoised Image using MMSE

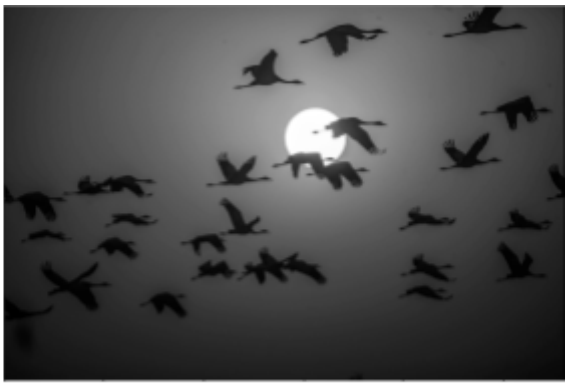
Hide a Secret Message in an Image DCT

Image compression

The aim of image compression is to reduce the amount of data (bits) to represent an image, useful for example when transmitting data. In order to do this, the statistical dependence within the image needs to be reduced. For this task a grayscale image is used, this is also due to the fact that the colour channels of an image are statistically dependent.

The image used for this task is “geese.jpg” and it will be transformed to grayscale

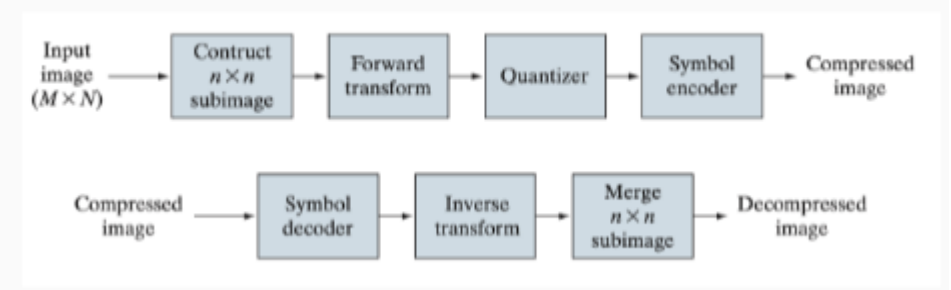
Original Image



Grayscale

Block Transform Coding

Block Transform Coding is a compression technique. The image is divided into pixel blocks (in this case 8x8 or 16x16) that do not overlap. An invertible, linear 2D transform is then applied to each block independently, resulting in a set of transform coefficients (for each block) which will be quantized (lossy step) and coded. More specifically, because the human eye will not notice very few distortions, some of the transform coefficients with small magnitudes can be quantizised or eliminated, this is when the compression takes place. Overall, packing as much information into the smallest number of coefficients.



Discrete Cosine Transform

For this task the Discrete Cosine Transform (DCT) will be the transform applied to each block. Thus, the basis functions of the transform are cosines. This is because fewer cosine

than sine functions are needed to approximate a signal. Because DCT coefficients have an order of importance, the goal is to remove the less important coefficients.

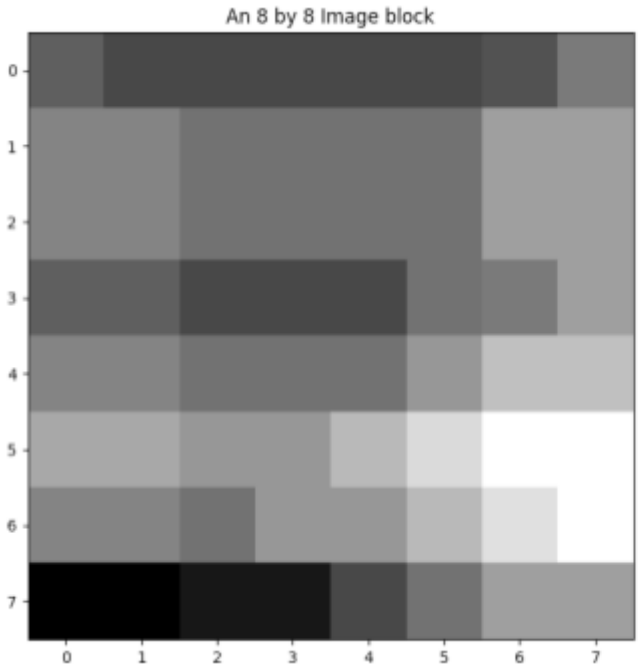
To compute the 2D DCT in a blockwise manner, we use an inbuilt function:

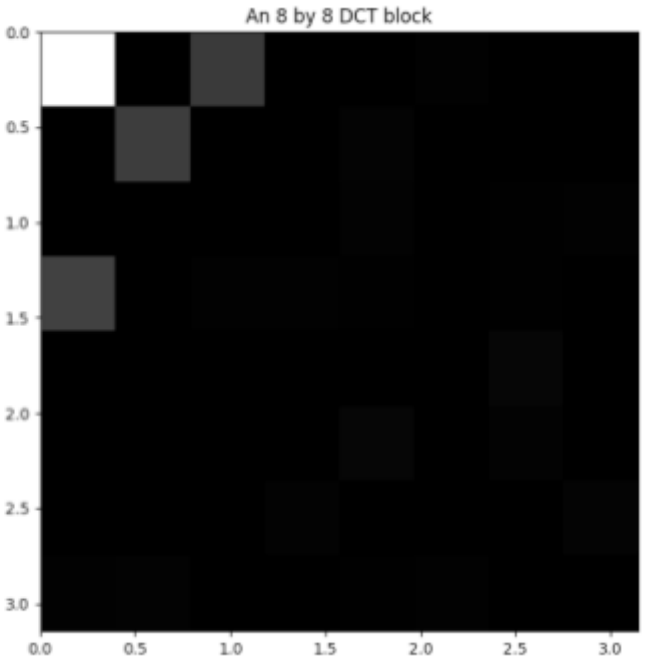
```
def dct2(a):
   return scipy.fftpack.dct( scipy.fftpack.dct(a, axis=0, norm='ortho'), axis=1,
norm='ortho')
```

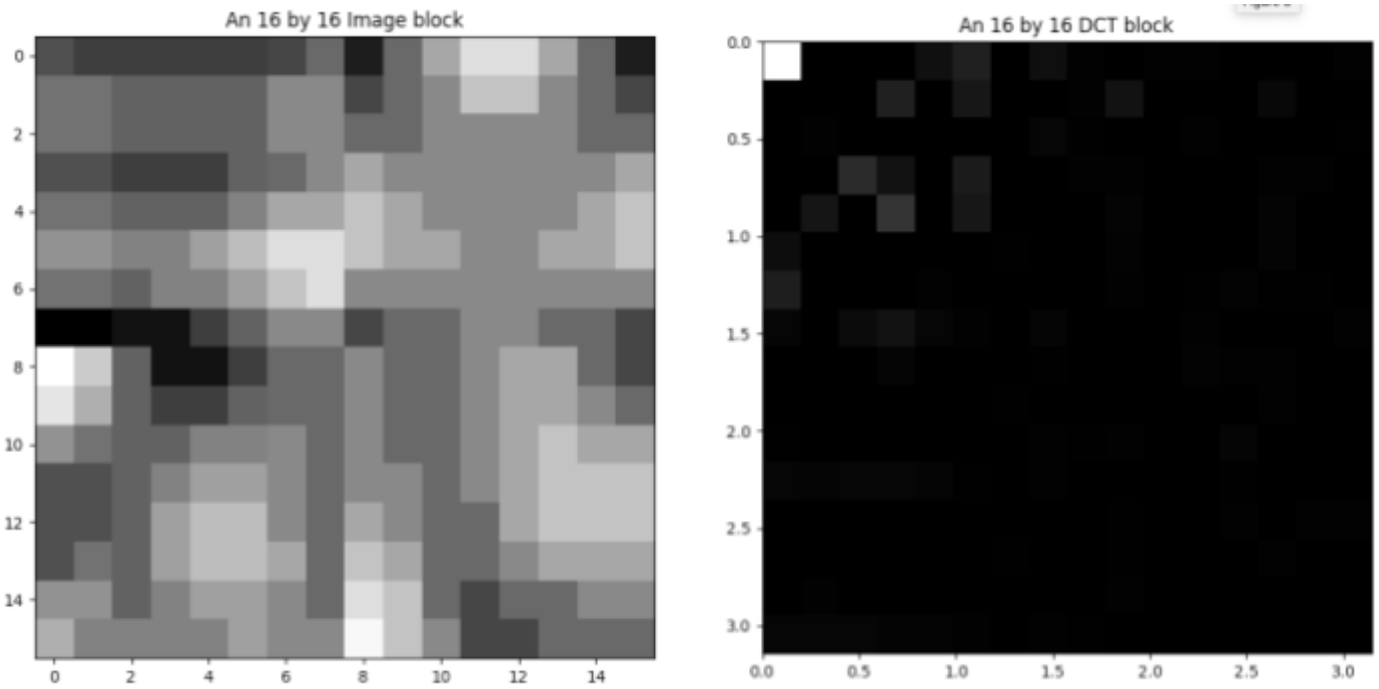
Which is then used within the created method computeDCT(), that takes in the size of the block.

```
def computeDCT(blockSize):
   # Do 8x8 DCT on image (in-place)
   for i in r_[:imsize[0]:blockSize]:
       for j in r_[:imsize[1]:blockSize]:
           dct[i:(i+blockSize),j:(j+blockSize)] = dct2(
im[i:(i+blockSize),j:(j+blockSize)] )
   print(dct.shape)
   pos = 128
   # Extract a block from image
   plt.figure()
   plt.imshow(im[pos:pos+blockSize,pos:pos+blockSize],cmap='gray')
   plt.title("An {} by {} Image block".format(blockSize, blockSize))

   # Display the dct of that block
   plt.figure()
   plt.imshow(dct[pos:pos+blockSize,pos:pos+blockSize],cmap='gray',vmax=
np.max(dct)*0.01,vmin = 0, extent=[0,pi,pi,0])
   plt.title("An {} by {} DCT block".format(blockSize, blockSize))
```

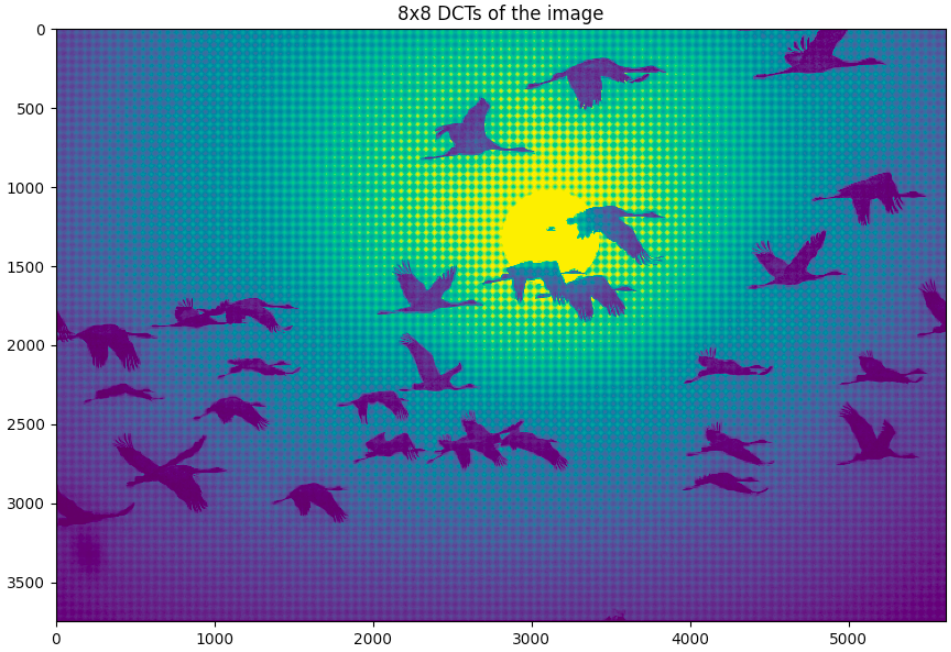






Looking at the DCT block, in both cases the leftmost (on top) coefficient is the DC Coefficient is the constant-valued basis function.
Lets take more specifically, the 8 by 8 block. After applying the DCT to the block, we have 64 DCT coefficients. These now have to be quantized.

The entire image in DCT (8 by 8 blocks) looks like this:



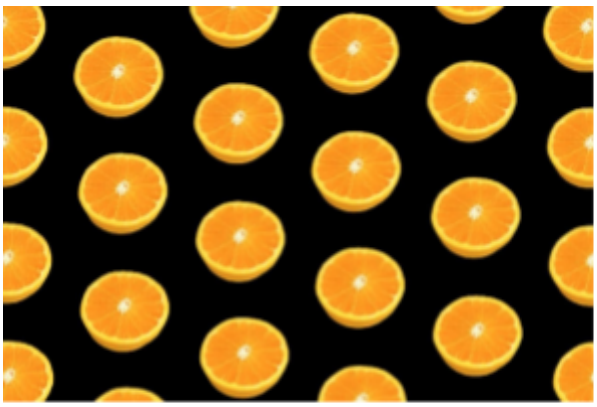
Decodes the quantized coefficients, computes the inverse two-dimensional DCT of each block, and puts the blocks back together into a single image

Morphology

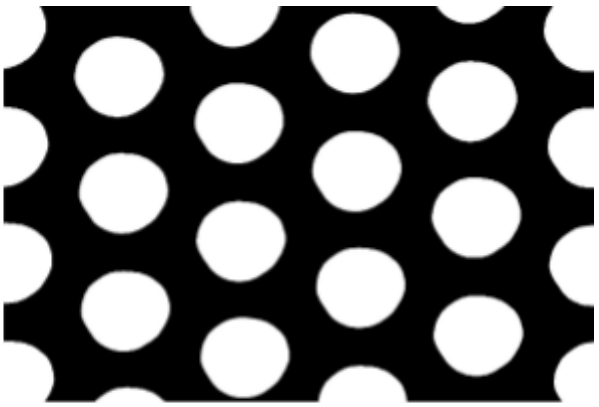
Pre-processing

For this task, the original RGB images are converted first to grayscale and then a threshold technique is applied, such that all round object are white and the rest is black. This is done so that the mophological operations to dilate and erode can be applied later.

```
oranges = cv2.imread("images project 2/oranges.jpg", 0)
orangeTree = cv2.imread("images project 2/orangetree.jpg", 0)
ret, thresh1Oranges = cv2.threshold(oranges, 120, 255, cv2.THRESH_BINARY) # 120
is the threshold
ret, thresh1OrangeTree = cv2.threshold(orangeTree, 120, 255, cv2.THRESH_BINARY)
```

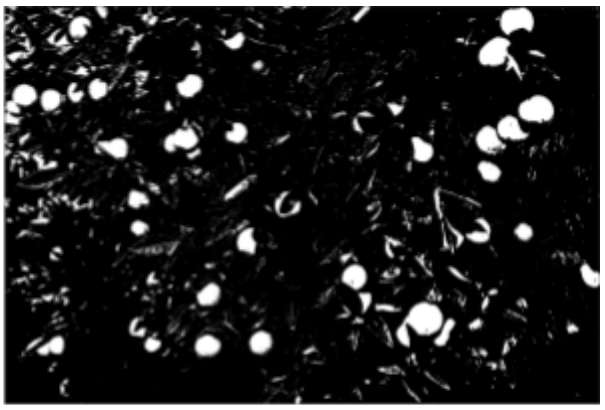


Oranges Original



Oranges with threshold

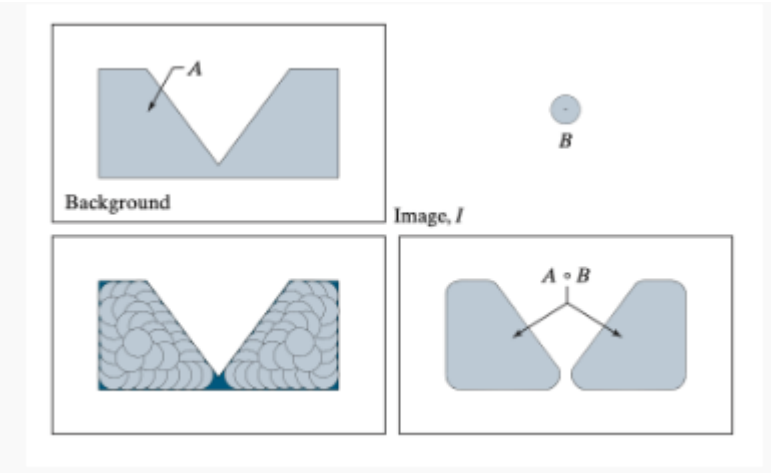
Orange Tree original



Orange tree threshold

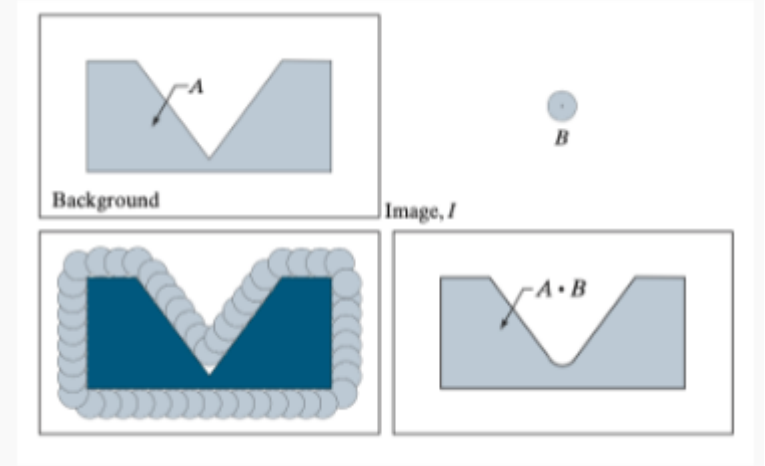
The cv2.threshold() method works in such way that if the pixel value of an image is below the threshold it is set to 0, thus black, else to the maximum value which is 255, white.
While this method works perfectly for the image of the oranges, it is not enough for the image of the orange tree. This is because some of the pixels on the leaves are converted to

255, most likely because of the light in the picture. To denoise this image, erosion and dilation have to be applied.
Dilation is used to smooth outlines, generally making the elements in the image "bigger". Erosion, on the other hand makes the elements in the image "smaller". Combined, they create two morphological operations called *Opening* and *Closing*.
Opening is erosion followed by dilation. To do this we need a structuring element (kernel) which in this case takes the form of an ellipse of a chosen size. During *Opening* the structuring element is pushed under the surface of the image, and all the *peaks*, in which it does not fit, are removed.




```
def opening(img, k):
    erode = cv2.erode(img, k)
    dilate = cv2.dilate(erode, k)
    return dilate
```

Closing is dilation followed by erosion. Thus, the structuring element is pushed over the surface of the image and all the *valleys* where it does not fit are removed.



```
def closing(img, k):
    dilate = cv2.dilate(img, k)
    erode = cv2.erode(dilate, k)
    return erode
```

For the first part of this task, we are asked to find a way to calculate the number of oranges in the orange tree. This was done by using the *Connected Components Method.* This method is a version of this original method extensively explained in: https://medium.com/swlh/image-processing-with-python-connected-components-and-region-labeling-3eef1864b951

```
def countOranges(img):
    k = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7,7))
    open = opening(img, k)
    close = closing(open, k)
    label_img = label(close)
    regions = regionprops(label_img)
    count = 0
    for num, x in enumerate(regions):
        area = x.area
        convex_area = x.convex_area
        if (num!=0 and (area>10) and (convex_area/area < 1.05) and
(convex_area/area > 0.95)):
            count = count + 1

    return count
```

At the beginning of this method a structuring element in the form of an ellipse is created. This structuring element is the kernel which will be used to first open and then close the image. Opening is also useful as it removes noise in the image (which was needed in the orange tree image). The if statement is used to check, firstly whether the region is 0, so it disregards regions that are black, and checking whether the convex area to area ratio is that of an ellipse. If so the counter increases, assuming that the region is within an orange. This method finds 19 oranges in the orange tree.

Grayscale morphology and granulometry

For the second task, an image displaying lights is chosen and transformed into the grayscale domain. In this domain, erosion makes increases the sizes of dark features, while decreasing the ones of light features in the image. Dilation has the opposite effect.

Jar Original Image | Grayscale

```
# Load picture
jar = cv2.cvtColor(cv2.imread("images project 2/jar.jpg"), cv2.COLOR_BGR2RGB)
plt.imshow(jar)
plt.show()

# Convert to grayscale - preprocessing
gray_jar = cv2.cvtColor(jar, cv2.COLOR_BGR2GRAY)
plt.imshow(gray_jar, cmap='gray')
plt.show()
```

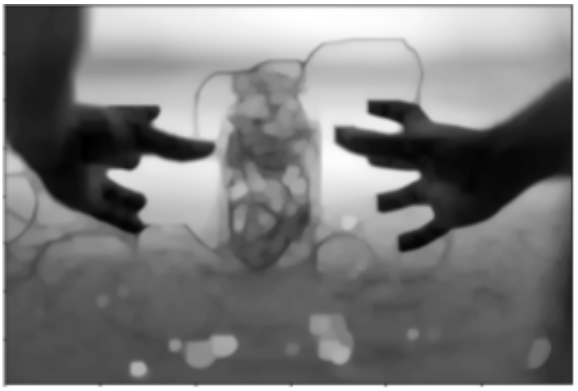
To find the frequencies of the different sized lights Granulometry was used. This technique is used to find the distribution of particles of certain light intensity and size.

```
def granulometry_two(img, size):
    s = 1
    s_open = img
    iter = size
    while iter > 0:
        se = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (s,s))
        open = opening(s_open, se)
        s = s + 1
        s_open = open
        iter = iter - 1
    return open
```

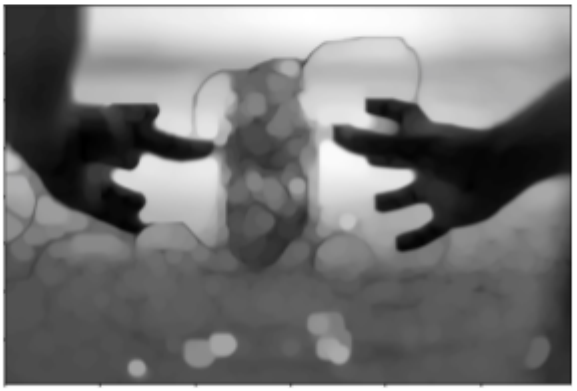
Opening, or more specifically, successive grayscale opening operations have to be performed. Each opening operation will use a structuring element of increased size (compared to the one used in the previous operation).

The following images show how the intensities in the original jar image change based on how many openings or iterations were made:

10 Iterations



20 Iterations



30 Iterations

To see how the frequency changes, the surface area (or the sum of pixel values) can be computed after each opening.

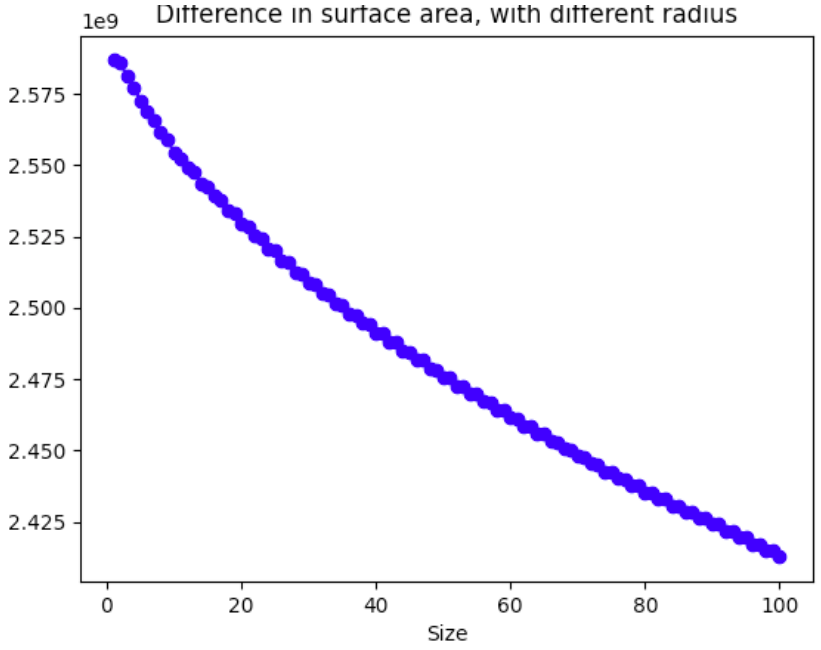
```
def granulometry(img, size):
    areas = np.zeros(len(size))

    for i in range(len(size)):
        se = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (i+1, i+1))
        open = opening(img, se)
        areas[i] = np.sum(open)
    return open, areas
```

```
size = np.linspace(1, 100, 100)
gran, frequencies = granulometry(gray_jar, size)
plt.plot(size, frequencies, '-bo')
plt.title('Difference in surface area, with different radius')
plt.xlabel('Size')
```

```
plt.ylabel('Differences in surface area')
plt.show()
```

Pixels with a smaller size than the structuring element will have been suppressed. Hence, the surface area decreases as the size of the structuring element increases. This can be seen by the graph below, with the structuring element going from 1 to 100 mm in radius:



PCA - Recognition

For this task, the following images were used:

For each image 4 other variations were created:

The first step is to turn each of the original images append each of the image variations into one big array of images.

```
womanPictures = []
for img in glob.glob("images project 2/PCAImages/woman*.JPG"):
   womanPictures.append(cv2.imread(img))
```

Since these images are coloured, they are represented as a 3D array n x m x 3. Thus the next step is to turn this array into a longer 1D array (or a vector) of mxnx3 elements.

```
def createDataMatrix(pictures):
   mat = np.stack(pictures, axis=0)
   mat = mat / 255
   return mat
```

```
h, w, c = womanPictures[0].shape
data_matrix = createDataMatrix(womanPictures)
mean = np.mean(data_matrix, axis=0)
data_matrix = np.subtract(data_matrix, mean)
m, eigen_vectors = pca(data_matrix)
mean_face = np.reshape(m, (h, w, c))
```

To get the mean we can compute the PCA with an Inbuilt function.
The Eigenvectors so obtained will have a length of m x n x 3. We can reshape these Eigenvectors so that the EigenFaces are again represented as a 3DArray m x n x 3.
The mean eigenface of the two images is the following

Eigenfaces are images that can be added to a mean (average) face to create new facial images

References

https://www.javatpoint.com/opencv-erosion-and-dilation#:~:text=Erosion%20and%20Dilation%20are%20morphological,or%20structure%20of%20an%20object.

https://www.geeksforgeeks.org/python-thresholding-techniques-using-opencv-set-1-simple-thresholding/#:~:text=Thresholding%20is%20a%20technique%20in,maximum%20value%20(generally%20255)

https://nl.mathworks.com/help/images/discrete-cosine-transform.html

https://medium.com/swlh/image-processing-with-python-connected-components-and-region-labeling-3eef1864b951

https://learnopencv.com/eigenface-using-opencv-c-python/