Computer Vision Assignment – 1

Sudheera Sivani Billakurti - S20160010014

Subhadeep Dash - S20160010021

V. Tejkiran - S20160020155

Question-1: Hybrid images

1. Initialization of libraries:

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import image as mpimg
```

The above snippet of code is for importing the libraries numpy for vectorization and matplotlib for reading and displaying images.

2. Hybrid Image conversion function:

```
def hybridImages(image1, image2, alpha, beta):
    def highpass(values, alpha):
        (M, N) = (values.shape[0], values.shape[1])
        for i in range(M):
            for j in range(N):
                if(np.linalg.norm((i, j)) < alpha):
                    values[i, j] = 0
        return values
    def lowpass(values, beta):
        (M, N) = (values.shape[0], values.shape[1])
        for i in range(M):
            for j in range(N):
                if(np.linalg.norm((i, j)) > beta):
                    values[i, j] = 0
        return values
    image1 = np.array(mpimg.imread(image1))
    image2 = np.array(mpimg.imread(image2))
    if(image1.shape != image2.shape):
        if(image1.shape > image2.shape):
            image1 = image1[:image2.shape[0], :image2.shape[1]]
        else:
            image2 = image2[:image1.shape[0], :image1.shape[1]]
    red1 = image1[:, :, 0]
    green1 = image1[:, :, 1]
    blue1 = image1[:, :, 2]
    red2 = image2[:, :, 0]
    green2 = image2[:, :, 1]
    blue2 = image2[:, :, 2]
    plt.figure();
    plt.imshow(image1)
    plt.show();
    plt.figure();
    plt.imshow(image2);
```

```
34.     plt.show();
35.     freq_image1_red = np.fft.fft2(red1)
36.     freq_image1_green = np.fft.fft2(green1)
37.     freq_image1_blue = np.fft.fft2(blue1)
38.     freq_image2_red = np.fft.fft2(red2)
39.     freq_image2_green = np.fft.fft2(green2)
40.     freq_image2_blue = np.fft.fft2(blue2)
41.     freq_image1_red = highpass(freq_image1_red, alpha)
42.     freq_image1_green = highpass(freq_image1_green, alpha)
43.     freq_image1_blue = highpass(freq_image1_blue, alpha)
44.     freq_image2_red = lowpass(freq_image2_red, beta)
45.     freq_image2_green = lowpass(freq_image2_green, beta)
46.     freq_image2_blue = lowpass(freq_image2_blue, beta)
47.     final_image_red = freq_image1_red + freq_image2_red
48.     final_image_green = freq_image1_green + freq_image2_green
49.     final_image_blue = freq_image1_blue + freq_image2_blue
50.     final_image_red = np.fft.ifft2(final_image_red)
51.     final_image_blue = np.fft.ifft2(final_image_blue)
52.     final_image_green = np.fft.ifft2(final_image_green)
53.     final_image = np.zeros(image1.shape)
54.     final_image[:,:,0] = np.absolute(final_image_red)
55.     final_image[:,:,1] = np.absolute(final_image_green)
56.     final_image[:,:,2] = np.absolute(final_image_blue)
57.     return np.absolute(final_image)/np.max(np.absolute(final_image))
```

The above function is used to apply low pass and high pass filters in the frequency domain of the images and further merging them in frequency domain to give a single hybrid image. The lines 2-8 define the high pass filter which allows the frequencies greater than a certain value α to pass and suppresses other frequencies whereas the lines 9-15 define the low pass filter which only allow frequencies less than β to pass.

In the lines 16-17, the images are read using imread function and from the lines 18 to 22, both the images are made to the same length and width, if they aren't of same shape. In the lines 23-28, the channels are separated in three channels for each image and then in the next 6 lines, we view the images. In the lines 35-46, the data in the channels are converted to frequency domain individually and high pass function is applied on the first image whereas low pass function is applied on the second image. In the next three lines, the result frequency intensities are merged into one and are converted to spatial domain in the further 3 lines. Then, we define the final image and assign the resultant channels to it and return the absolute normalized value of the resultant image as the final output.

INPUTS:

```
plt.imshow(hybridImages("marilyn.bmp", "einstein.bmp", 25, 1000))
plt.show()
```

```
plt.imshow(hybridImages("Afghan_girl_before.jpg","Afghan_girl_after.jpg
", 25, 750))
plt.show()
```

```
plt.imshow(hybridImages("bicycle.bmp", "motorcycle.bmp", 25, 750))
plt.show()
```

```
plt.imshow(hybridImages("fish.bmp", "submarine.bmp", 25, 750))
plt.show()
```

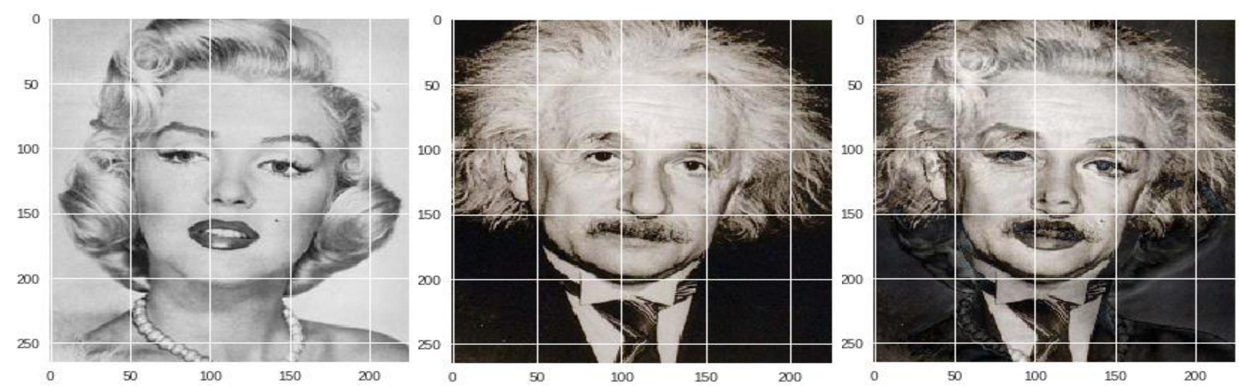
```
plt.imshow(hybridImages("bird.bmp", "plane.bmp", 25, 750))
plt.show()
```

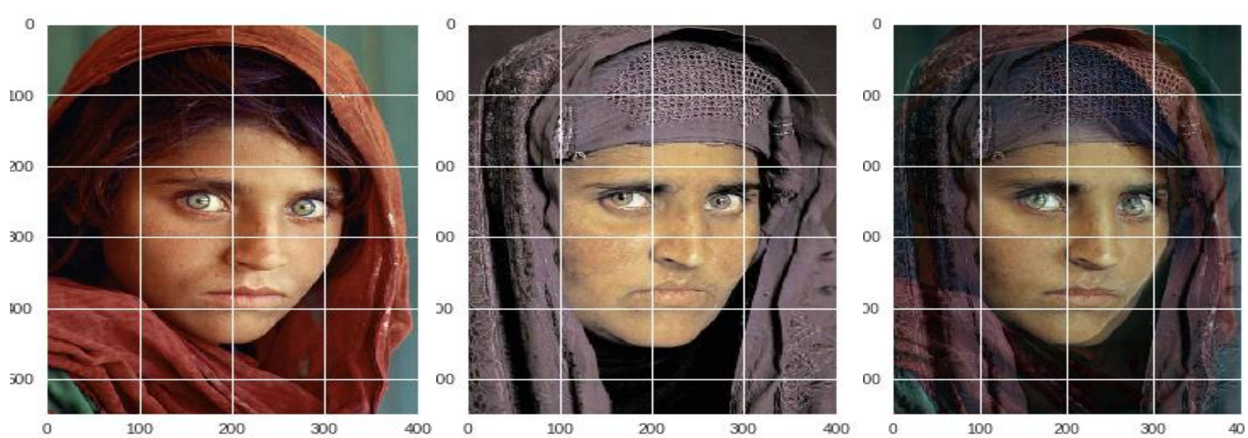
```
plt.imshow(hybridImages("cat.bmp", "dog.bmp", 5, 750))
plt.show()
```

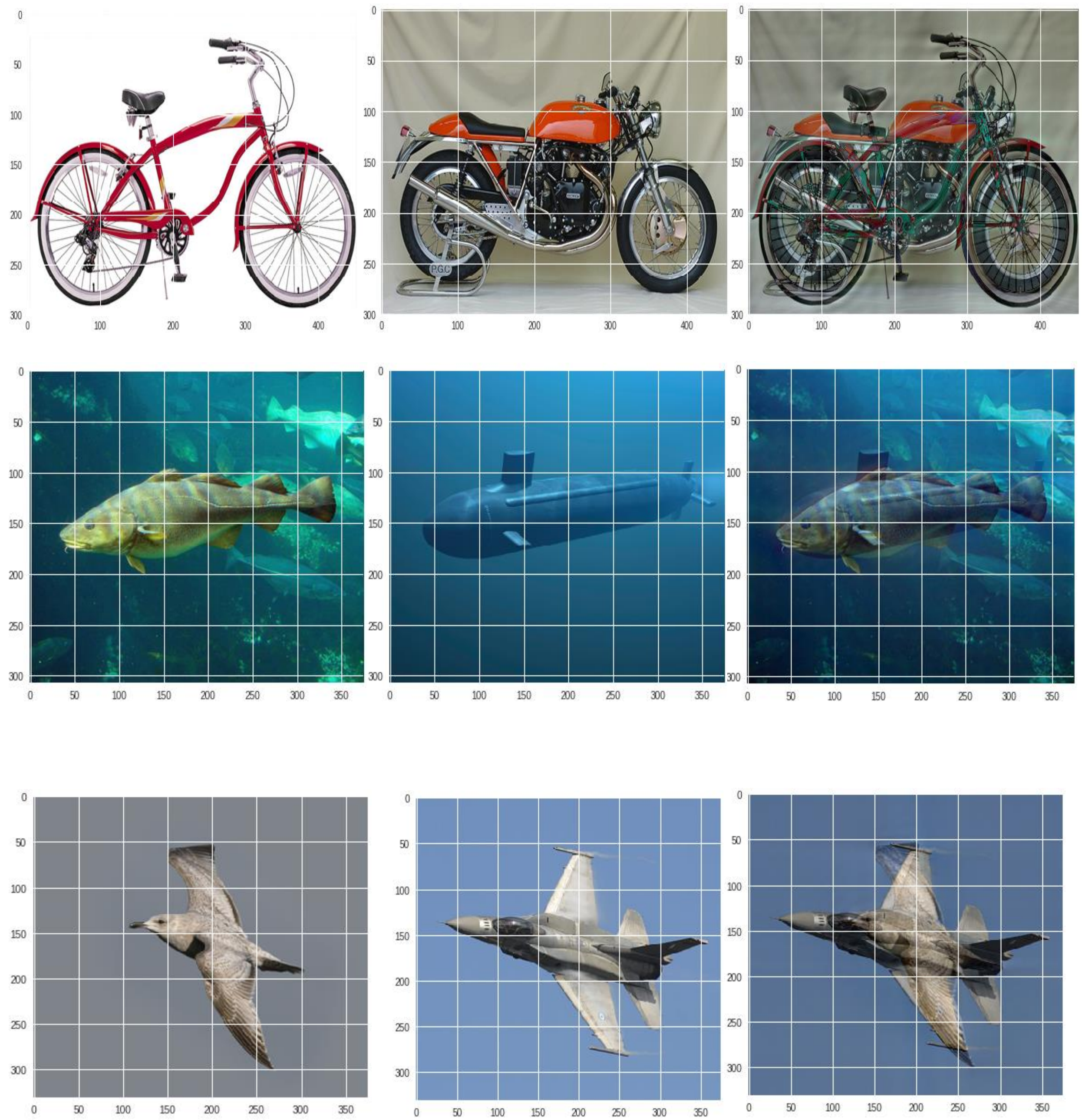
```
plt.imshow(hybridImages("makeup_before.jpg", "makeup_after.jpg", 5,
750))
plt.show()
```

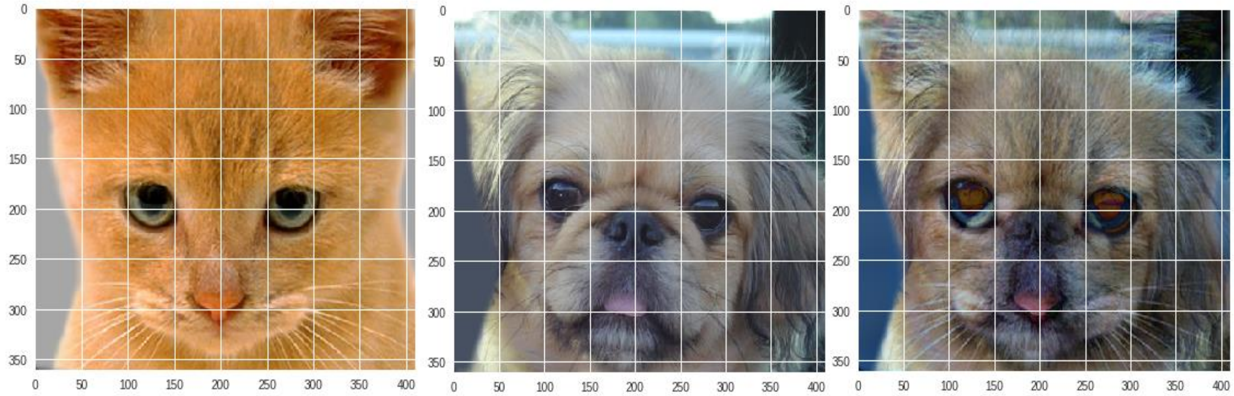
RESULTS:

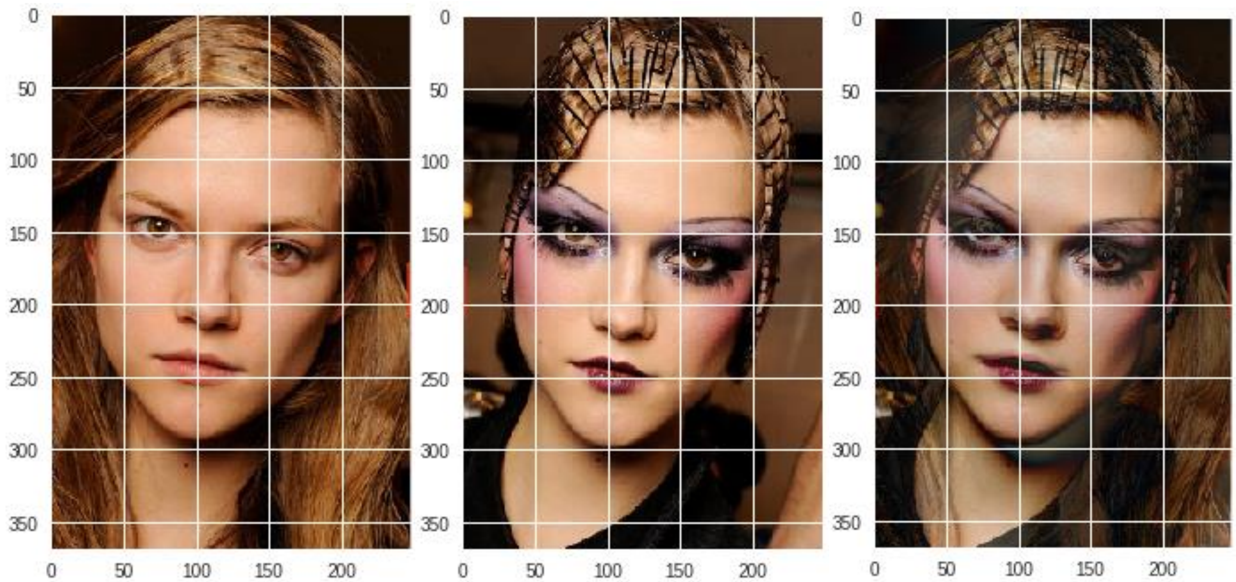
The output of the resultant function on various images can be seen below. In each of the sets, the first two images are the inputs to the function and the final image is the hybrid image obtained.











Question-2: Corner Detection

1. Initialization of libraries:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import image as mpimg
```

The above snippet of code is for importing the libraries numpy for vectorization and matplotlib for reading and displaying images.

2. Shi-Tomasi Corner detection algorithm:

```
def shi_tomasi(org_image, threshold, window_size):
    org_image = np.array(mpimg.imread(org_image))
    plt.figure()
    plt.imshow(org_image)
    plt.show()
    if(len(org_image.shape) == 3):
        image = np.dot(org_image[...,:3], [0.299, 0.587, 0.114])
```

```
8.     else:
9.         image = org_image
10.     left_shifted_image = np.concatenate((image[:, 1:], image[:,-1:]), axis=1)
11.     up_shifted_image = np.concatenate((image[1:, :], image[-1:,:]), axis=0)
12.     x_change = left_shifted_image-image
13.     y_change = up_shifted_image-image
14.     new_image = np.zeros(image.shape)
15.     m = image.shape[0]
16.     n = image.shape[1]
17.     for i in range(window_size//2, m-window_size//2):
18.         for j in range(window_size//2, n-window_size//2):
19.             A = np.square(x_change[i-window_size//2:i+window_size//2, j-window_size//2:j+window_size//2]).sum()
20.             B = np.multiply((x_change[i-window_size//2:i+window_size//2, j-window_size//2:j+window_size//2]),(y_change[i-window_size//2:i+window_size//2, j-window_size//2:j+window_size//2])).sum()
21.             C = np.square(y_change[i-window_size//2:i+window_size//2, j-window_size//2:j+window_size//2]).sum()
22.             H = np.array([[A, B], [B, C]])
23.             min_lambda = np.min(np.linalg.eigvals(H))
24.             if(min_lambda > threshold):
25.                 if(len(org_image.shape) == 3):
26.                     org_image[i][j] = [255, 0, 0]
27.                 else:
28.                     org_image[i][j] = 255
29.                 new_image[i][j] = 255
30.     return org_image, new_image
```

The gradients in each direction i.e. horizontally and vertically is calculated in lines 12-13. Then within a given window the second moment matrix is calculated as follows:

$$Second\ moment\ Matrix = \begin{bmatrix} A & B \\ B & C \end{bmatrix} = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

Then the eigen values for the matrix are calculated and it is checked if the minimum of them is above a threshold value. If $\lambda_{min} > threshold$, then it is marked a corner.

2. Harris Corner detection algorithm:

```
def harris_corners(org_image, threshold, window_size, k):
    org_image = np.array(mpimg.imread(org_image))
    plt.figure()
    plt.imshow(org_image)
    plt.show()
    if(len(org_image.shape) == 3):
        image = np.dot(org_image[...,:3], [0.299, 0.587, 0.114])
    else:
        image = org_image
```

```
    left_shifted_image = np.concatenate((image[:, 1:], image[:,-1:]), axis=1)
    up_shifted_image = np.concatenate((image[1:, :], image[-1:,:]), axis=0)
    x_change = left_shifted_image-image
    y_change = up_shifted_image-image
    new_image = np.zeros(image.shape)
    m = image.shape[0]
    n = image.shape[1]
    for i in range(window_size//2, m-window_size//2):
        for j in range(window_size//2, n-window_size//2):
            A = np.square(x_change[i-window_size//2:i+window_size//2, j-window_size//2:j+window_size//2]).sum()
            B = np.multiply((x_change[i-window_size//2:i+window_size//2, j-window_size//2:j+window_size//2]),(y_change[i-window_size//2:i+window_size//2, j-window_size//2:j+window_size//2])).sum()
            C = np.square(y_change[i-window_size//2:i+window_size//2, j-window_size//2:j+window_size//2]).sum()
            H = np.array([[A, B], [B, C]])
            f = np.linalg.det(H) - k*np.square(np.trace(H))
            if(f > threshold):
                if(len(org_image.shape) == 3):
                    org_image[i][j] = [255, 0, 0]
                else:
                    org_image[i][j] = 255
                new_image[i][j] = 255
    return org_image, new_image
```

In the Harris corner detection function, he second moment matrix is computed similar as in previous case. The only difference is the condition based on which it's decided if it's a corner or not. A point is considered a corner if the following condition satisfies:

$$\boldsymbol{det}\left(\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}\right) - \boldsymbol{k} * \left(\boldsymbol{trace}\left(\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}\right)\right) > \boldsymbol{threshold}$$

Where k is a constant.

INPUTS:

1) Shi-Tomasi Corner Detection Algorithm:

```
corners_placed, corners = shi_tomasi("chess.jpg", 1000, 7)
plt.figure()
plt.imshow(corners_placed)
plt.show()
plt.figure()
plt.imshow(corners, cmap="gray")
plt.show()
```

We have tested the corner detection image on an additional chess board image apart from the data provided.

```
corners_placed, corners = shi_tomasi("Image1.jpg", 100, 3)
plt.figure()
plt.imshow(corners_placed)
plt.show()
plt.figure()
plt.imshow(corners, cmap="gray")
plt.show()
```

```
corners_placed, corners = shi_tomasi("Image2.jpg", 100, 3)
plt.figure()
plt.imshow(corners_placed)
plt.show()
plt.figure()
plt.imshow(corners, cmap="gray")
plt.show()
```

```
corners_placed, corners = shi_tomasi("Image3.jpg", 100, 3)
plt.figure()
plt.imshow(corners_placed)
plt.show()
plt.figure()
plt.imshow(corners, cmap="gray")
plt.show()
```

2) Harris Corner Detection Algorithm:

```
corners_placed, corners = harris_corners("chess.jpg", 2000, 9, 0.24)
plt.figure()
plt.imshow(corners_placed)
plt.show()
plt.figure()
plt.imshow(corners, cmap="gray")
plt.show()
```

```
corners_placed, corners = harris_corners("Image1.jpg", 20000, 3, 0.10)
plt.figure()
plt.imshow(corners_placed)
plt.show()
plt.figure()
plt.imshow(corners, cmap="gray")
plt.show()
```

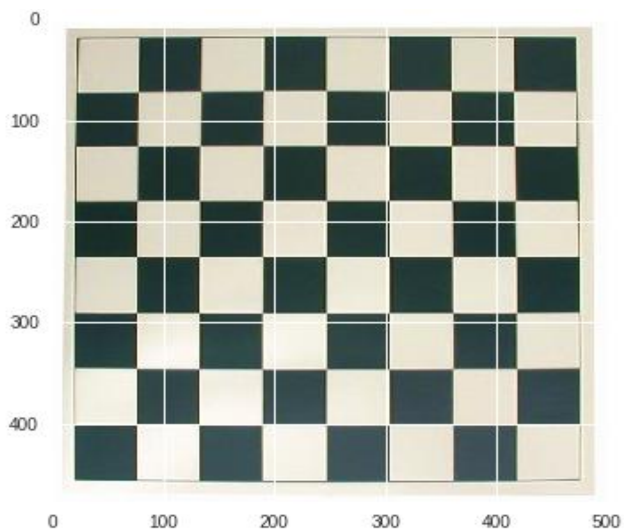
```
corners_placed, corners = harris_corners("Image2.jpg", 10000, 3, 0.1)
plt.figure()
plt.imshow(corners_placed)
plt.show()
plt.figure()
plt.imshow(corners, cmap="gray")
plt.show()
```

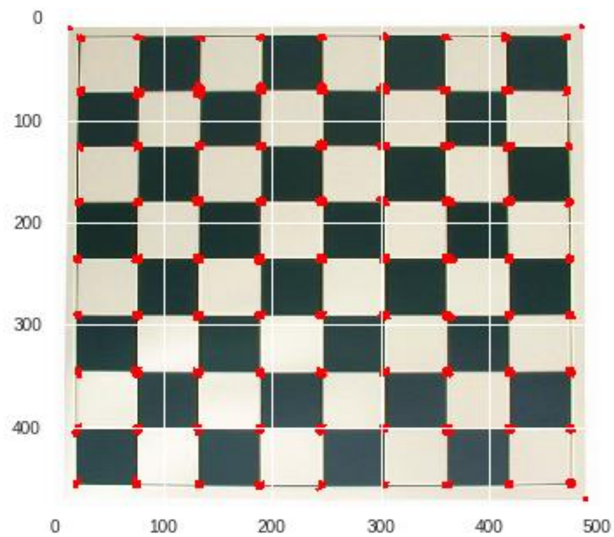
```
corners_placed, corners = harris_corners("Image3.jpg", 20000, 3, 0.10)
plt.figure()
plt.imshow(corners_placed)
plt.show()
plt.figure()
plt.imshow(corners, cmap="gray")
plt.show()
```

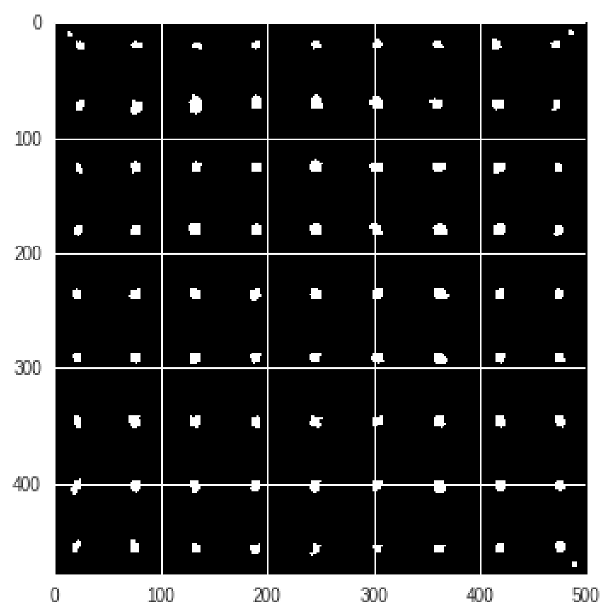
RESULTS:

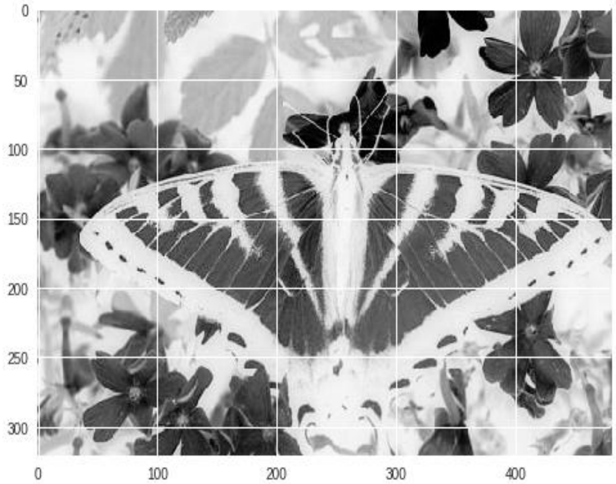
1) Shi-Tomasi Corner Detection Algorithm:

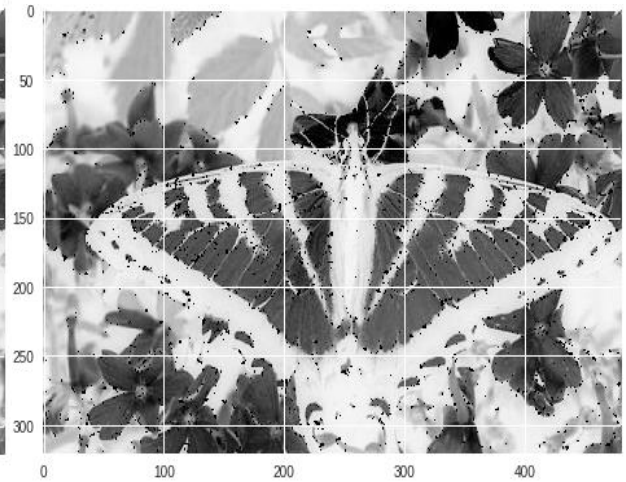
In each row, the first image is the original image. The second image contains the picture as well as the points. The third image contains the corners.

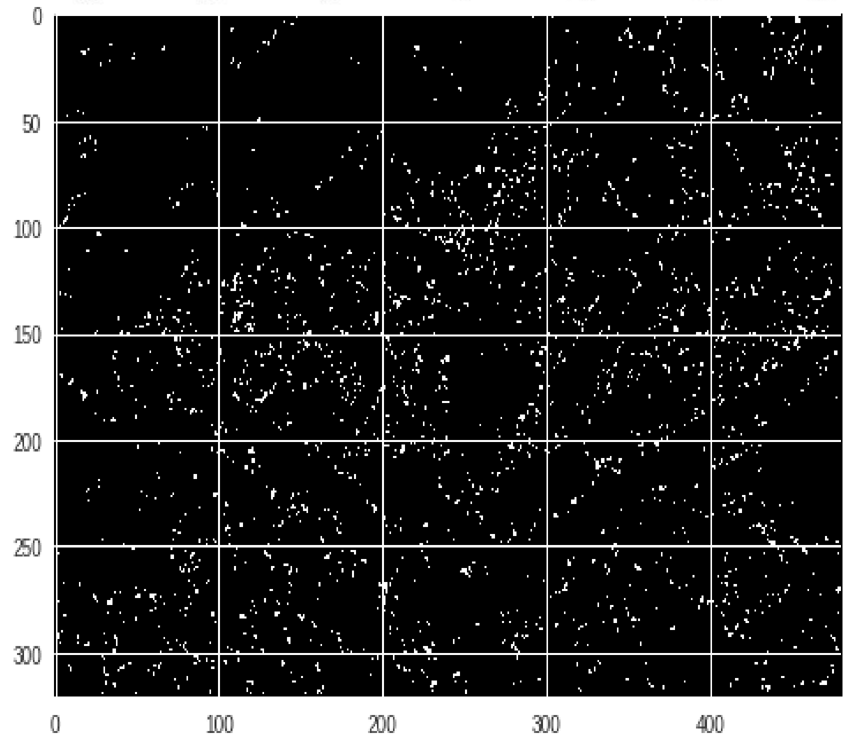


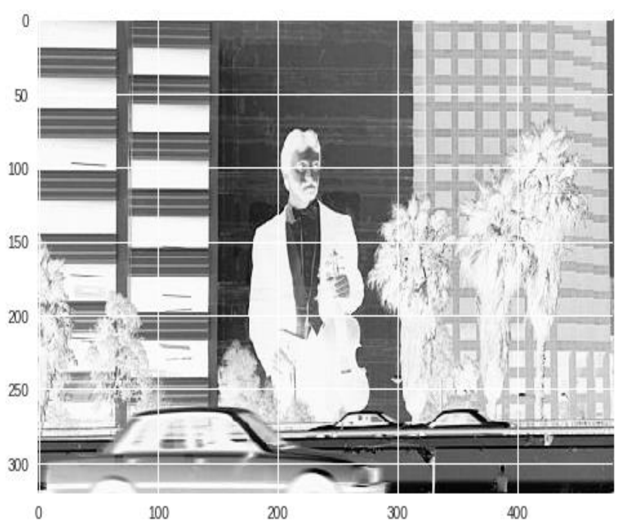




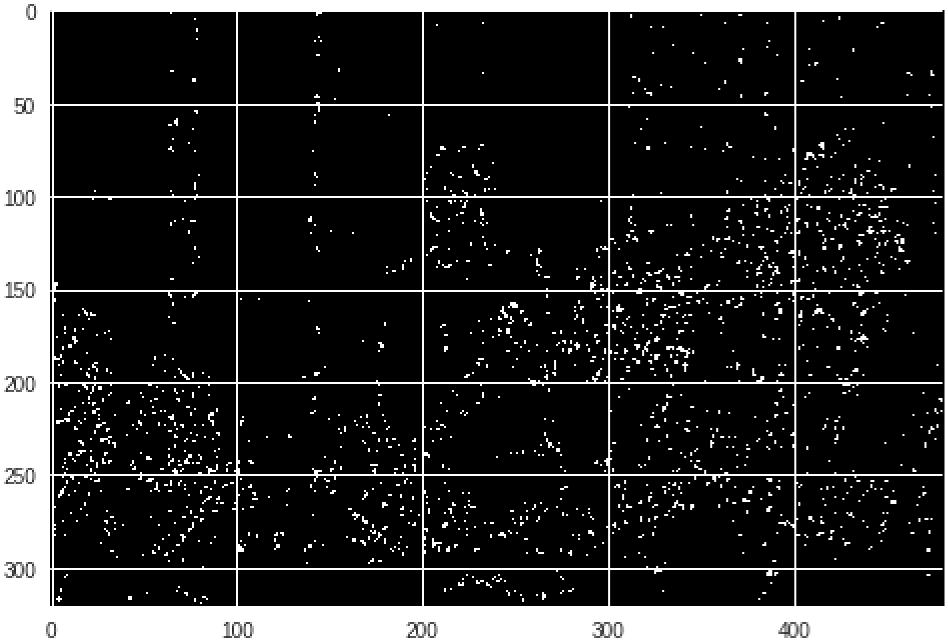

0
50
100
150
200
250
300
0
100
200
300
400

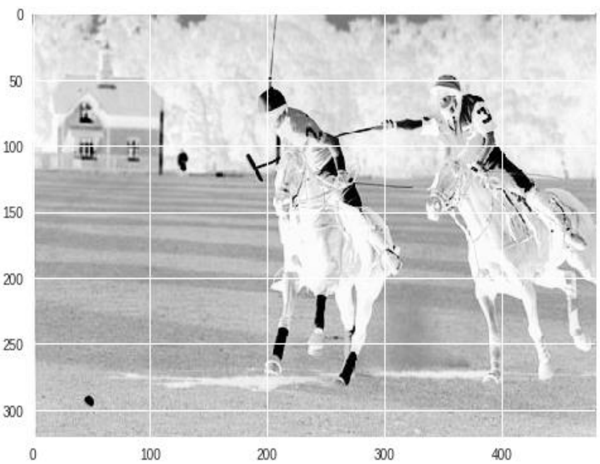

0
50
100
150
200
250
300
0
100
200
300
400

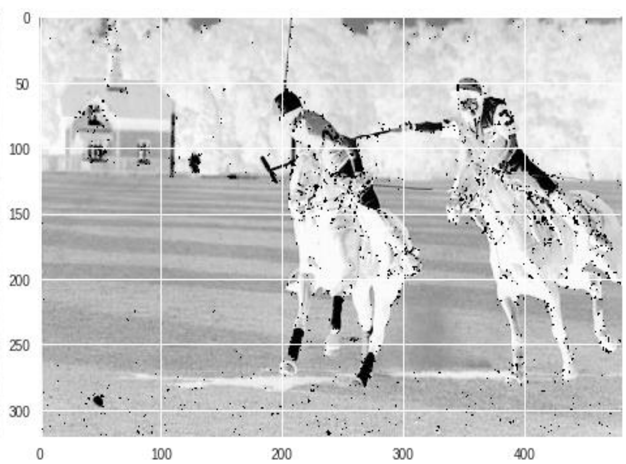

0
50
100
150
200
250
300
0
100
200
300
400

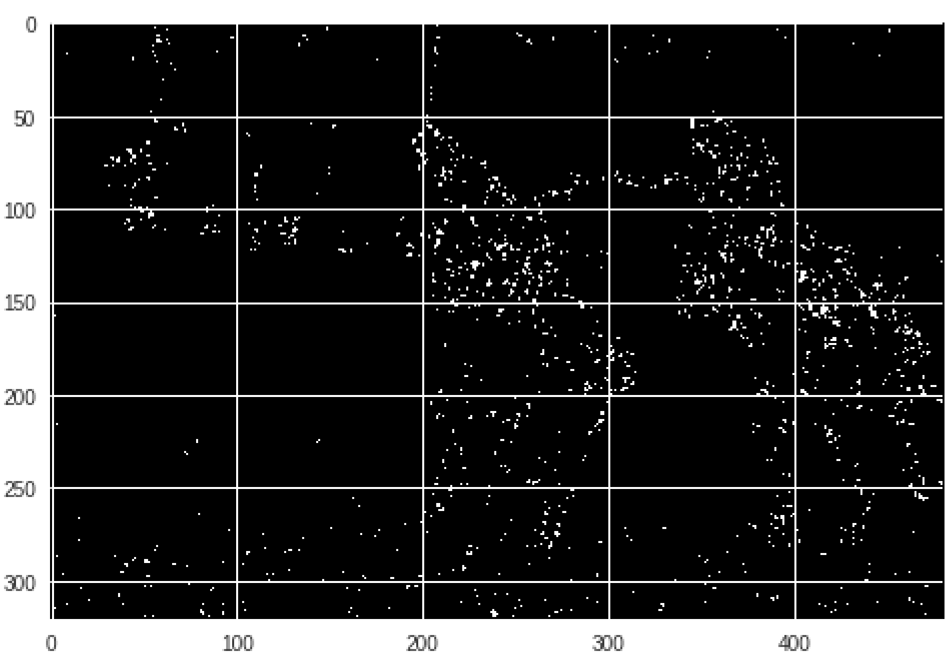

0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400

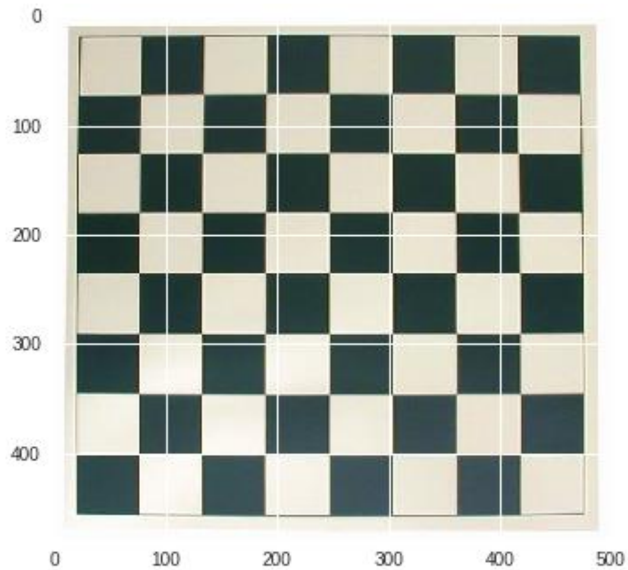

0
50
100
150
200
250
300
0
100
200
300
400

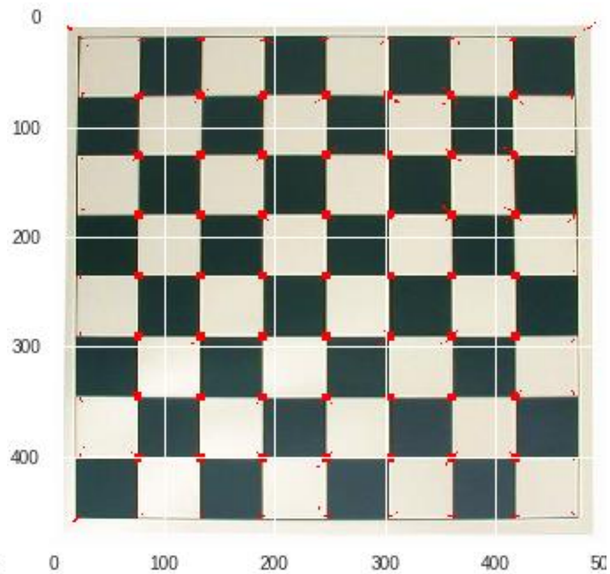

0
50
100
150
200
250
300
0
100
200
300
400

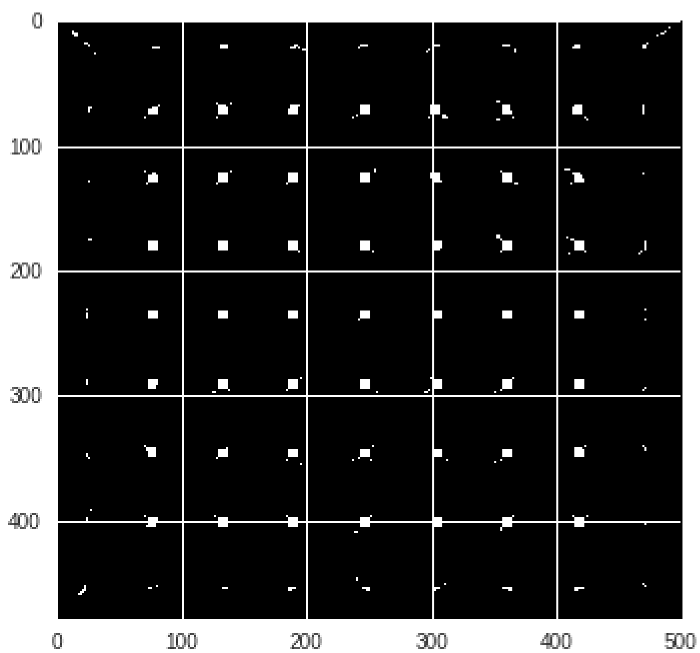

0
50
100
150
200
250
300
0
100
200
300
400

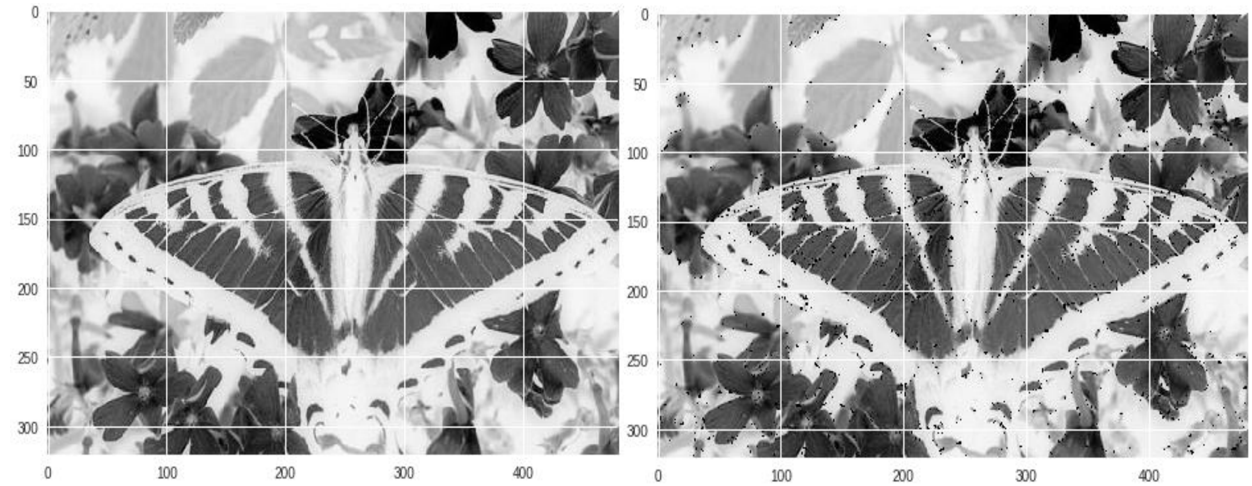
2) Harris Corner Detection Algorithm:

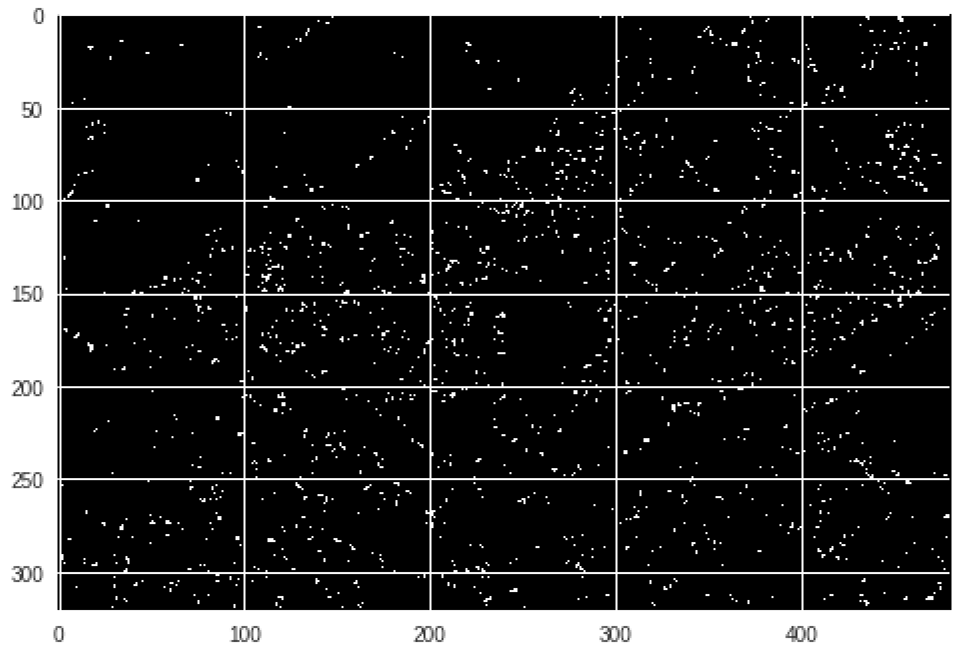
In each row, the first image is the original image. The second image contains the picture as well as the points. The third image contains the corners.

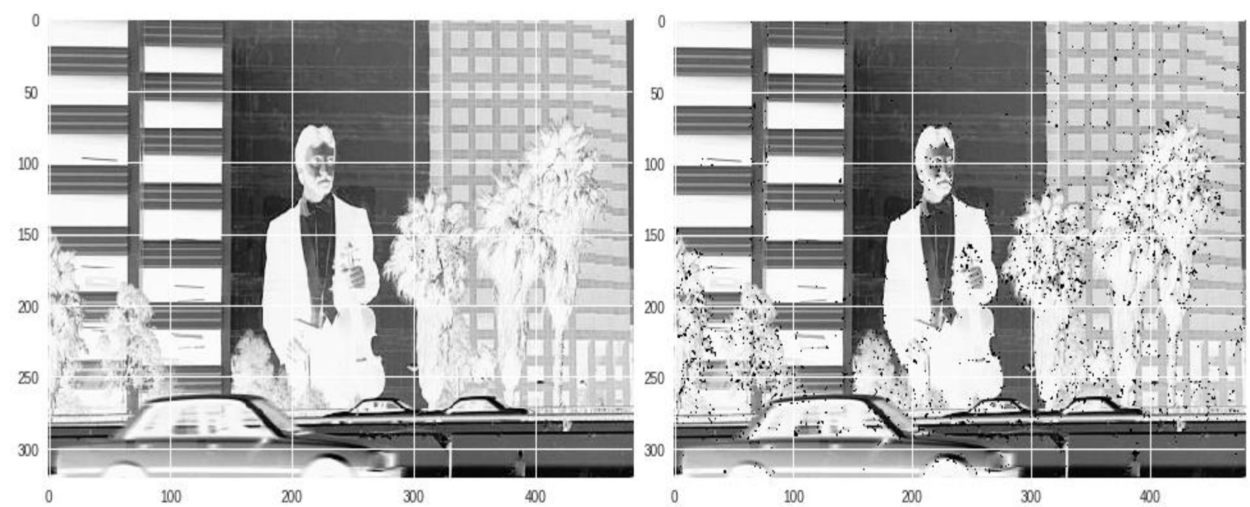


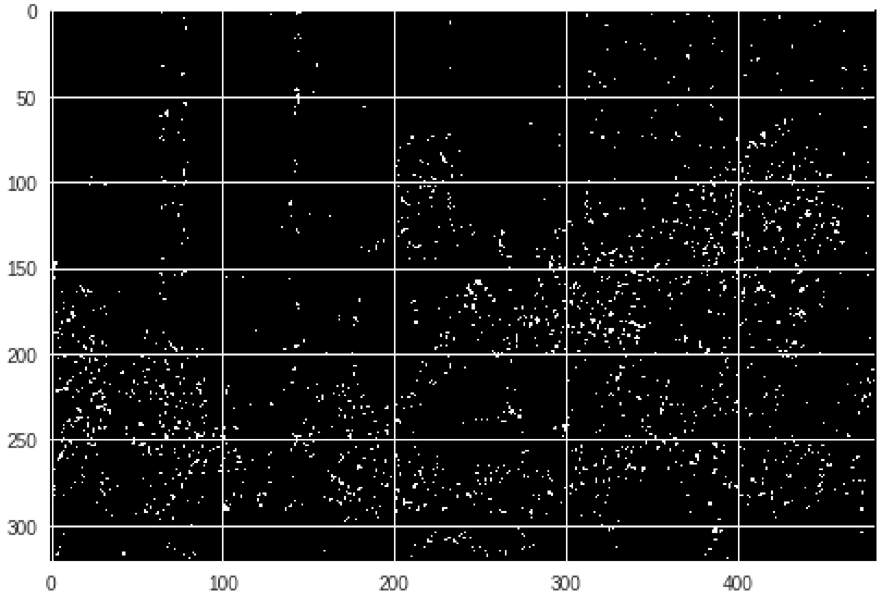


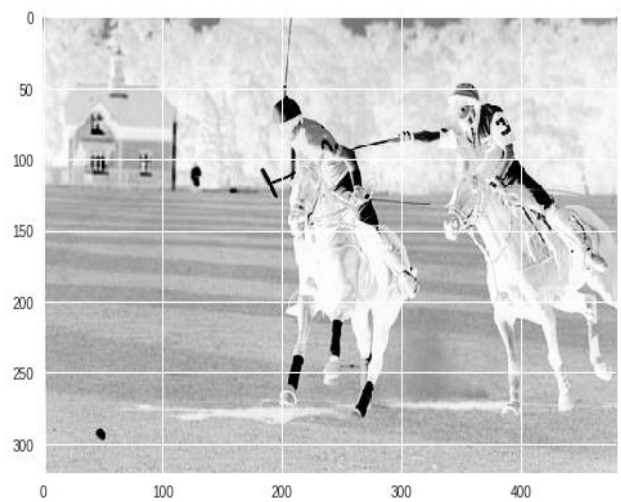


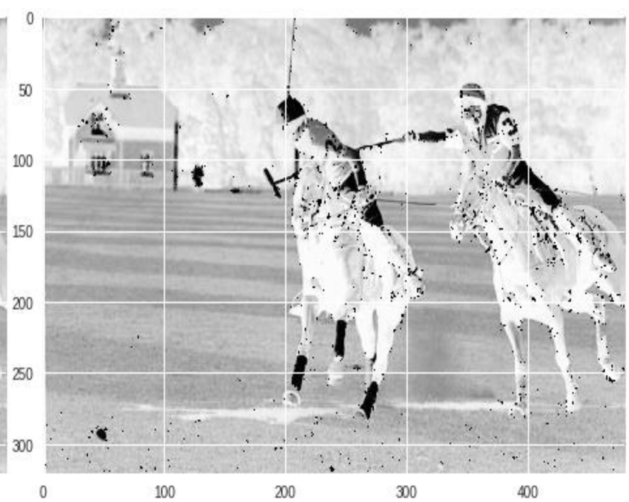


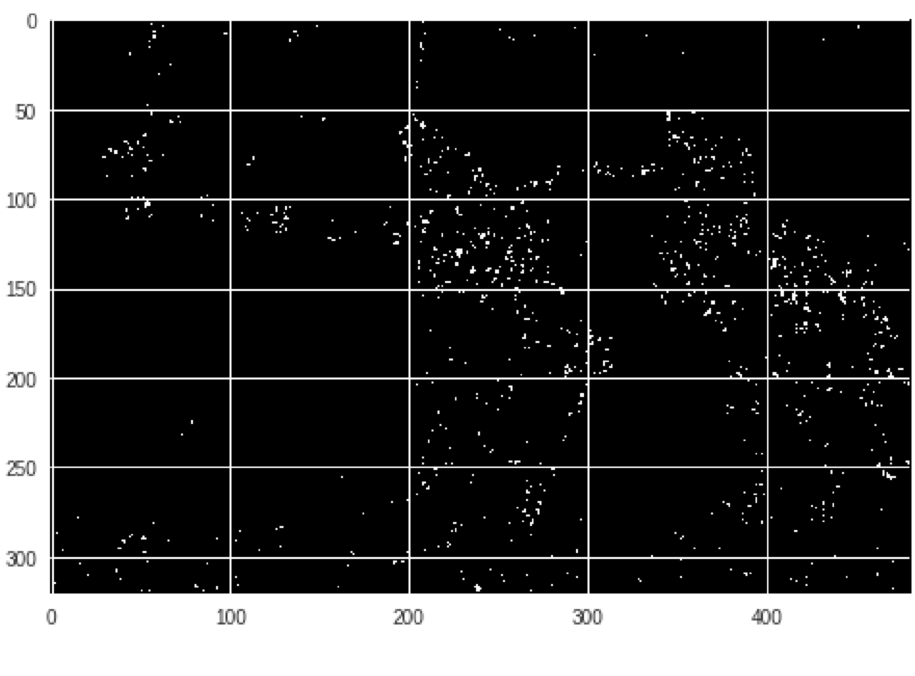





0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400

Question-3: Scale space blob detection

1. Initialization of libraries:

```
import numpy as np
import time
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

2. Various functions:

```
def Laplacian_of_Gaussian(n, sigma):
    laplacian = np.array([range(-n//2+1, n//2+1, 1) for _ in range(n)])
    common_factor = (np.square(laplacian)+np.square(laplacian.T))
    laplacian = (common_factor-2*np.square(sigma))*np.exp(-common_factor/(2*np.square(sigma)))
    return Laplacian

def Gaussian_filter(n, sigma):
    gaus_filter = np.array([range(-n//2+1, n//2+1, 1) for _ in range(n)])
    gaus_filter = np.square(gaus_filter) + np.square(gaus_filter.T)
     gaus_filter = (1/(2*np.pi*np.square(sigma)))*np.exp(-gaus_filter/(2*np.square(sigma)))
     #gaus_filter = np.repeat(gaus_filter, 3).reshape(n, n, 3)
     return(gaus_filter)

 def dog(k,n,sigma):
     a=Gaussian_filter(n,sigma)
     b=Gaussian_filter(n,k*sigma)
     return b-a

 def padding(image, n):
     padded_image = np.concatenate((np.array([image[0] for _ in range(n//2)]), image, np.array([image[-1] for _ in range(n//2)])), axis=0)
     padded_image = np.concatenate((np.repeat(padded_image[:, 0:1],n//2, axis=1), padded_image, np.repeat(padded_image[:, -1:],n//2, axis=1)), axis=1)
     return padded_image

 def apply_filter(img, lapl):
     M, N = img.shape
     K = lapl.shape[0]
     final_image = np.zeros(img.shape)
     for i in range(M-K):
         for j in range(N-K):
             final_image[i+K//2, j+K//2] = np.multiply(img[i:i+K, j:j+K], lapl).sum()
     return np.square(final_image)

```

```
33. def max_filter(img, K, sigma):
34.     M, N = img.shape
35.     final_image = np.zeros(img.shape)
36.     plt.figure()
37.     fig, ax = plt.subplots()
38.     for i in range(M-K):
39.         for j in range(N-K):
40.             if(img[i+K//2, j+K//2] == np.max(img[i:i+K, j:j+K])):
41.                 final_image[i+K//2, j+K//2] = np.sqrt(2)*sigma
42.                 ax.add_artist(plt.Circle((j+K//2, i+K//2), np.sqrt(2)*sigma, color='r', fill = False, linewidth=3))
43.     plt.imshow(final_image, cmap="gray")
44.     plt.show()
45.     return final_image
```

Implementation of various functions such as Laplacian of Gaussian (in lines 1-5), Gaussian (in lines 7-12) and Difference of Gaussian (in lines 14-17), Padding (in lines 19-22), Convolution (in lines 24-31) and Non-max Suppression (in lines 33-45).

```
start = time.time()
n = 10
path = "butterfly.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 1
org_image = mpimg.imread(path)
if(len(org_image.shape) == 3):
    image = np.dot(org_image[...,:3], [0.299, 0.587, 0.114])
else:
    image = org_image
layers = np.zeros((n, image.shape[0], image.shape[1]))
for i in range(n):
    sigma = org_sigma*np.power(factor, i)
    window_size = int(np.ceil(6*sigma))
    if(window_size%2 == 0):
        window_size += 1
    if(filter_type == 1):
      LoG = Laplacian_of_Gaussian(window_size, sigma)
    else:
      LoG=dog(np.power(factor, i+1),window_size,org_sigma)
    plt.figure()
    layers[i] = apply_filter(padding(image, window_size), LoG)[window_size//2:-window_size//2+1, window_size//2:-window_size//2+1]
    plt.imshow(layers[i], cmap = "gray")
centres = np.zeros(layers.shape)
for i in range(n):
    sigma = org_sigma*np.power(factor, i)
    window_size = int(np.ceil(6*sigma))
    if(window_size%2 == 0):
        window_size += 1
    centres[i] = max_filter(layers[i], window_size, sigma)
plt.figure()
fig, ax = plt.subplots()
for i in range(centres.shape[0]):
    sigma = org_sigma*np.power(factor, i)
    for j in range(centres.shape[1]):
        for k in range(centres.shape[2]):
```

```
38.                 if(centres[i][j][k]!=0 and centres[i, j, k] ==
   np.max(centres[:, int(j-np.sqrt(2)*sigma):int(j+np.sqrt(2)*sigma), k-
   int(np.sqrt(2)*sigma):k+int(np.sqrt(2)*sigma)])):
39.                     ax.add_artist(plt.Circle((k, j), np.sqrt(2)*sigma,
   color='r', fill = False, linewidth=1))
40.  plt.imshow(org_image)
41.  plt.show()
42.  end = time.time()
43.  elapsed = end - start
44.  print(elapsed)
```

In this snippet, the variables n, org_sigma, factor, path are used to represent the number of iterations, the initial standard deviation, the factor by which standard deviation is scaled every iteration, the path of the image file respectively and the filter_type 1 corresponds to usage of Laplacian of Gaussian filter whereas filter_type 2 refers to the usage of Difference of Gaussian filter.

The line 7 reads the image and immediate after we convert the 3D RGB image to grayscale image. From the lines 12-24, the filter is applied with the scaled variance and then after applying non-max suppression the resultant information is stored in the layers list. Then the lines (25-32) perform the non-max suppression on the image giving centers to the circles of radius $\sqrt{2}\sigma$. These centers are stored in center list on which again Non-max Suppression is applied over the stack of layers and only then the circles are plotted with their respective radii.

INPUTS:

The following parameters are send for detecting blobs:

```
n = 10
path = "butterfly.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 1
```

```
n = 10
path = "butterfly.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 2
```

```
n = 10
path = "einstein.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 1
```

```
 n = 10
path = "einstein.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 2
```

```
n = 10
path = "fishes.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 1
```

```
n = 10
path = "fishes.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 2
```

```
n = 10
path = "sunflowers.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 1
```

```
n = 10
path = "sunflowers.jpg"
org_sigma=3.5
factor = 1.21
filter_type = 2
```

```
n = 10
path = "flowers.jpeg"
org_sigma=3.5
factor = 1.21
filter_type = 1
```

```
n = 10
path = "flowers.jpeg"
org_sigma=3.5
factor = 1.21
filter_type = 2
```

```
n = 10
path = "colors.jpeg"
org_sigma=3.5
factor = 1.21
filter_type = 1
```

```
n = 10
path = "colors.jpeg"
org_sigma=3.5
factor = 1.21
filter_type = 2
```

```
n = 10
path = "rods.jpeg"
org_sigma=3.5
factor = 1.21
filter_type = 1
```

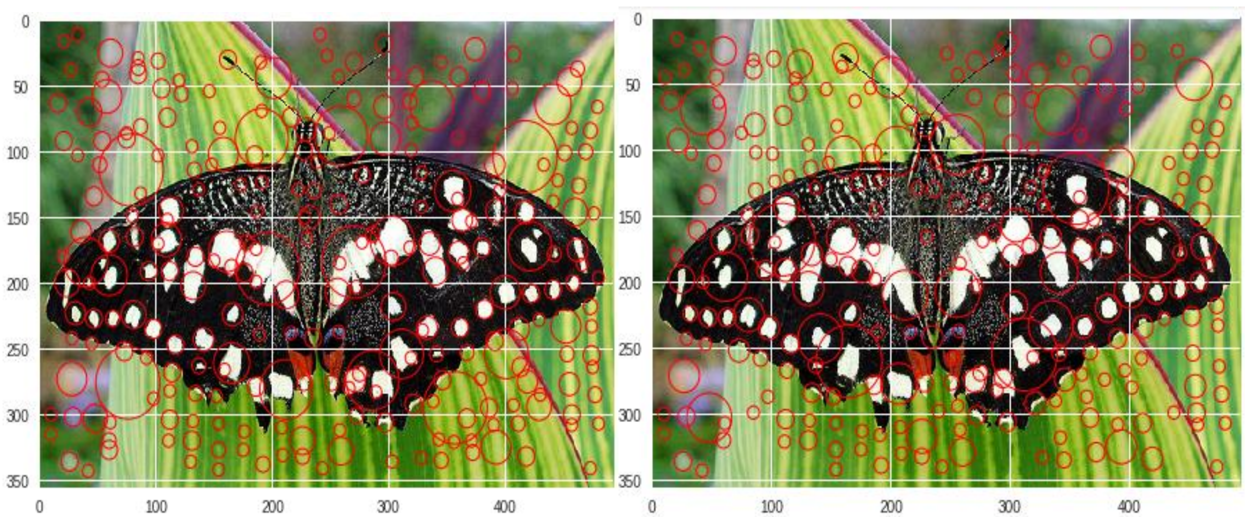
```
1. n = 10
2. path = "rods.jpeg"
3. org_sigma=3.5
4. factor = 1.21
5. filter_type = 2
```

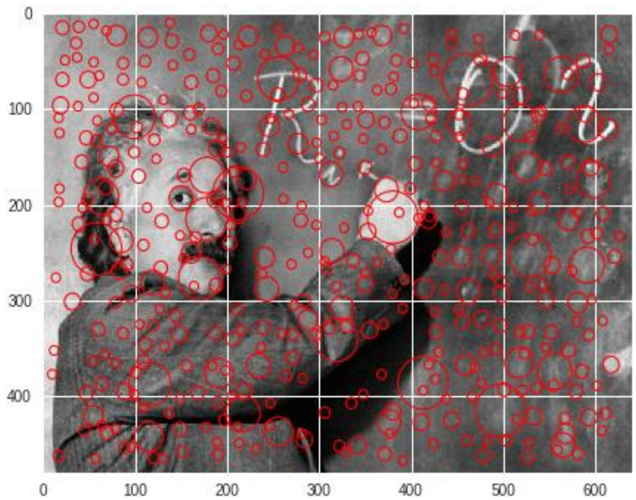
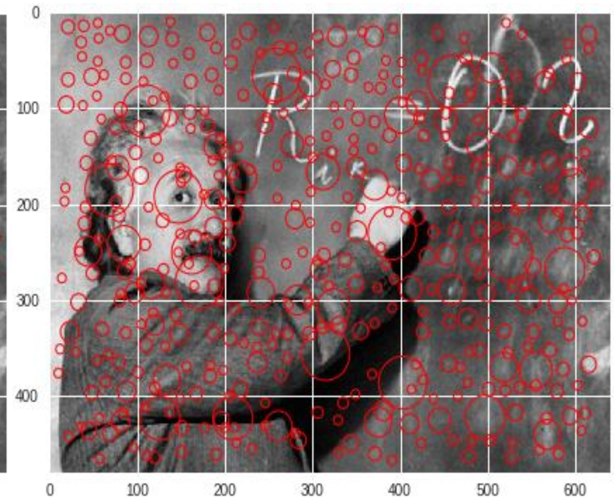
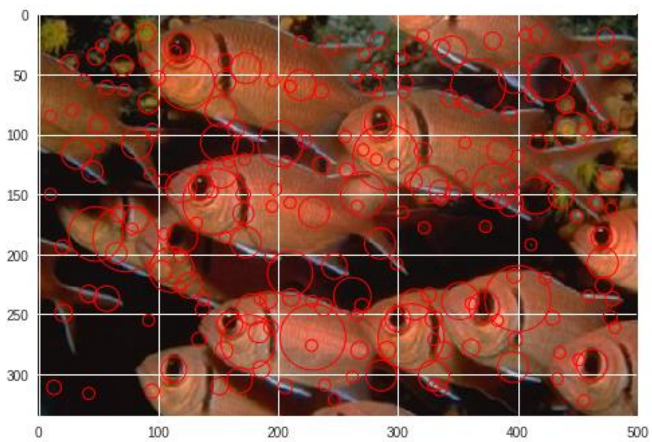
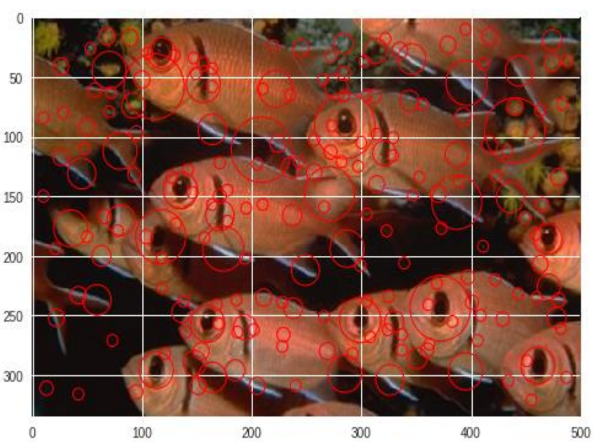
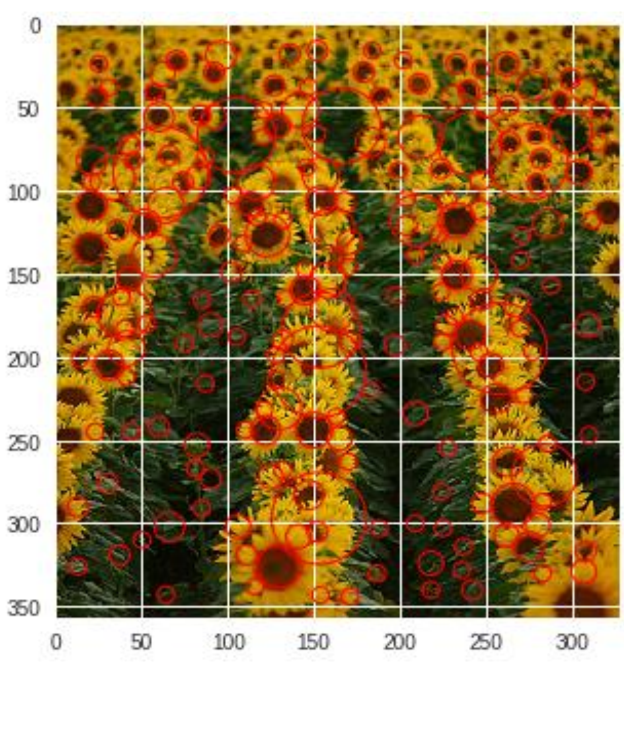
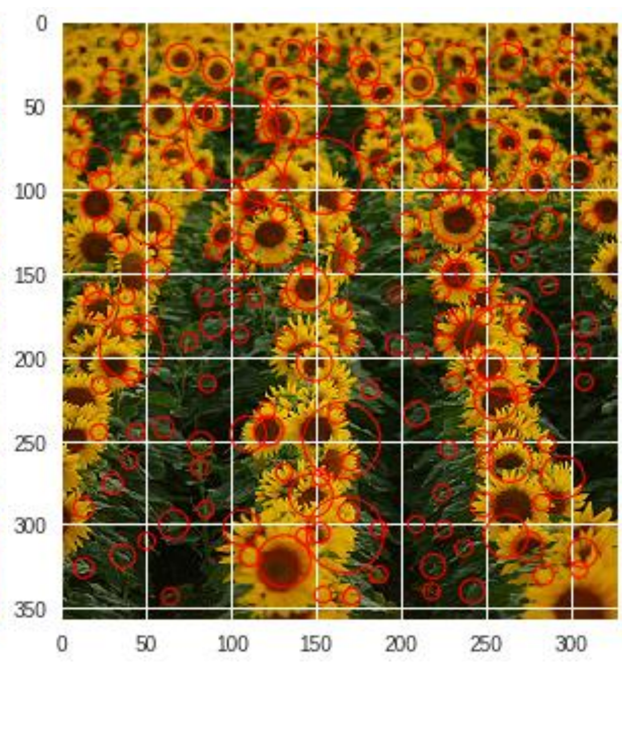
```
1. n = 10
2. path = "bubbles.jpeg"
3. org_sigma=3.5
4. factor = 1.21
5. filter_type = 1
```

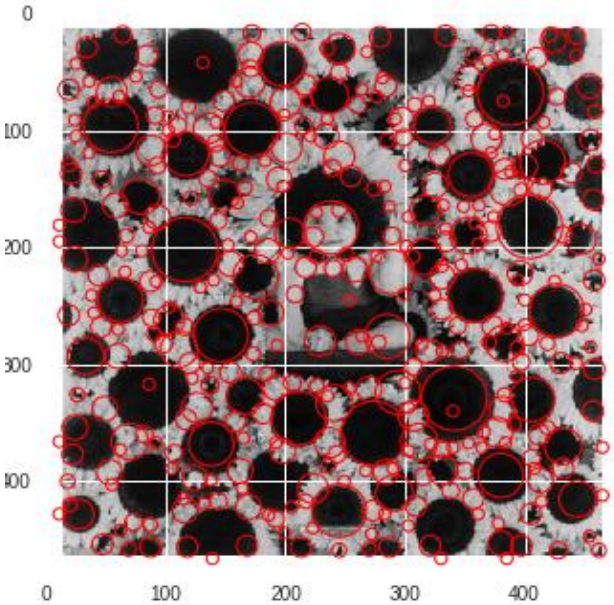
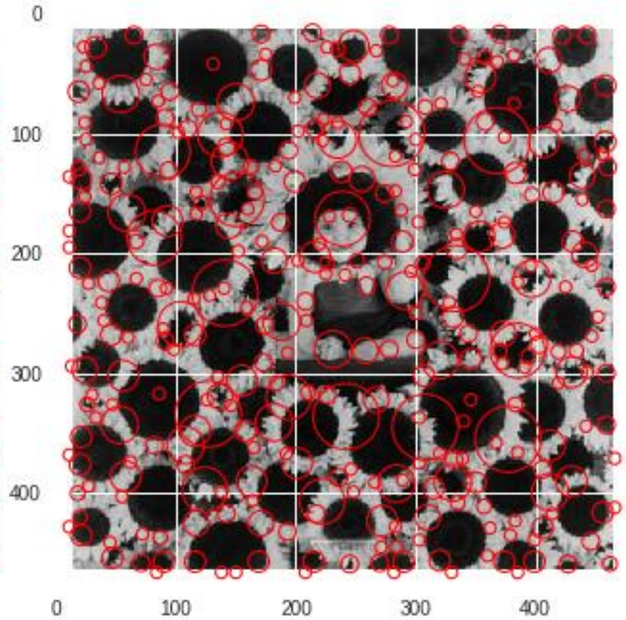
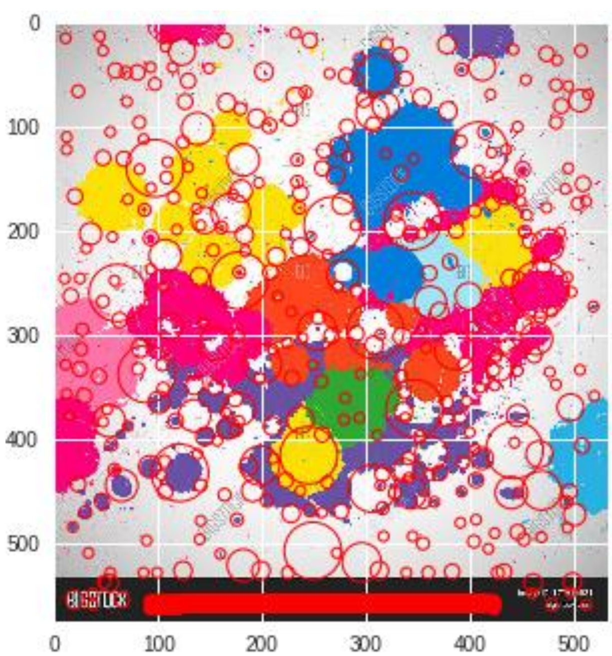
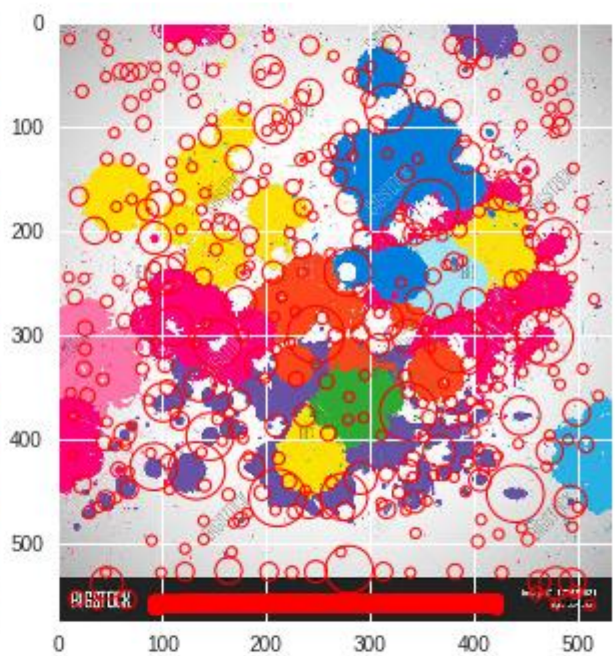
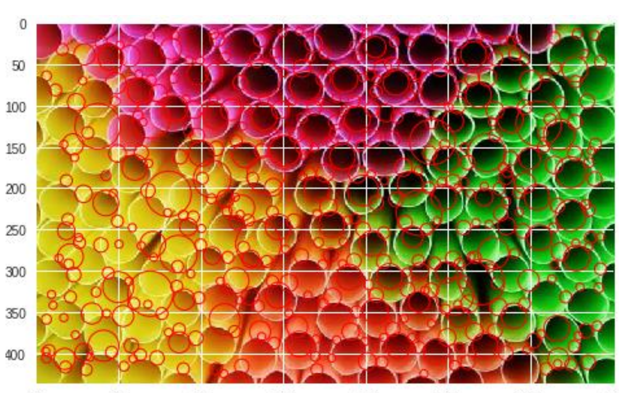
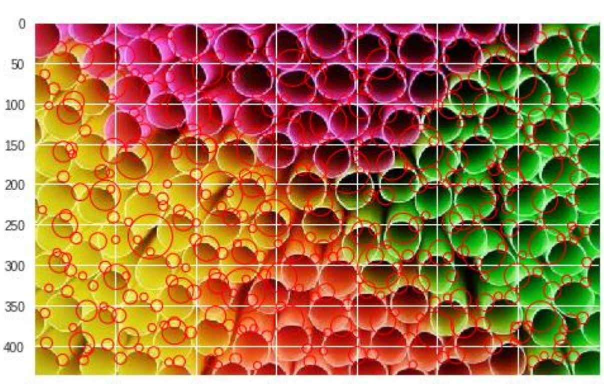
```
1. n = 10
2. path = "bubbles.jpeg"
3. org_sigma=3.5
4. factor = 1.21
5. filter_type = 2
```

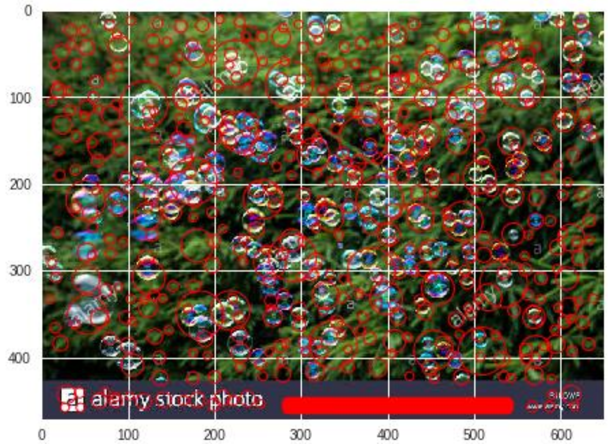
RESULTS:

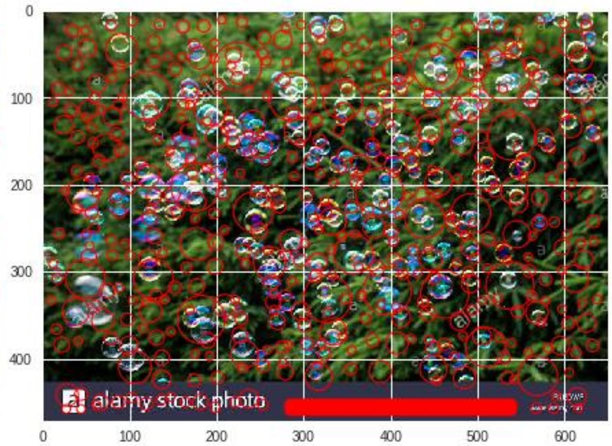
The outputs of Laplacian of Gaussian are on left where as the ones of Difference of Gaussian are on right.




0 100 200 300 400
0 100 200 300 400 500 600

0 100 200 300 400
0 100 200 300 400 500 600

0 50 100 150 200 250 300
0 100 200 300 400 500

0 50 100 150 200 250 300
0 100 200 300 400 500

0 50 100 150 200 250 300 350
0 50 100 150 200 250 300

0 50 100 150 200 250 300 350
0 50 100 150 200 250 300


0
100
200
300
400
0 100 200 300 400

0
100
200
300
400
0 100 200 300 400

0
100
200
300
400
500
0 100 200 300 400 500

0
100
200
300
400
500
0 100 200 300 400 500

0
50
100
150
200
250
300
350
400
0 100 200 300 400 500 600 700

0
50
100
150
200
250
300
350
400
0 100 200 300 400 500 600 700





Comparison of both algorithms based on their runtimes:

Images	**Running time of Laplacian of Gaussian (in secs)**	**Running time of Difference of Gaussian (in secs)**
butterfly.jpg	34.674816608428955	34.38932180404663
einstein.jpg	57.405301094055176	57.869709968566895
fishes.jpg	33.20057916641235	33.29722452163696
sunflowers.jpg	24.47645902633667	24.39238691329956
flowers.jpeg	43.549209117889404	43.62589621543884
colors.jpeg	63.052472829818726	63.8885293006897
rods.jpeg	58.875545263290405	58.92578172683716
bubbles.jpeg	63.670729637145996	64.42570161819458

We observe that the runtimes of both the algorithms are quite similar but performance wise, we can see that the Laplacian of Gaussian is better than Difference of Gaussian in detecting blobs.

<u>Thanks for Reading</u>