COMPUTER VISION

ASSIGNMENT-01

Parkhi Mohan - S20160010061
Sree Pragna Vinnakoti - S20160010106
Subash Karthik - S20160010052

1. HYBRID IMAGES

AIM

To create a HYBRID IMAGE from two images by using high frequency of one image and low frequency of other image.

PROCEDURE

Importing necessary Libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

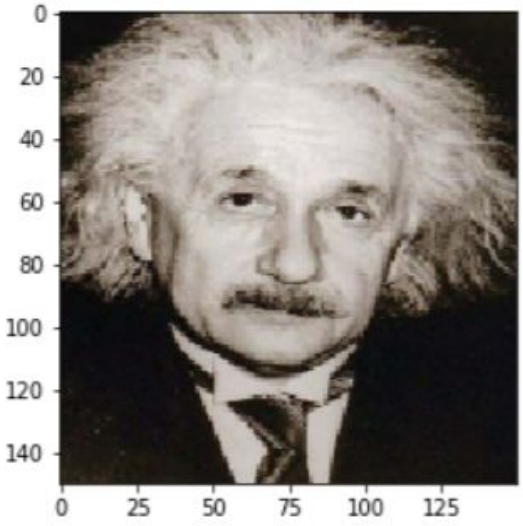
Opening the two images:

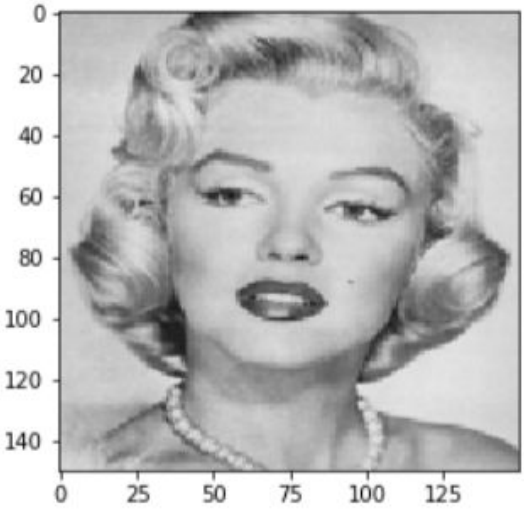
```
img1 = cv2.resize(plt.imread('./HW1_Q1/einstein.bmp'),(150,150))
img2 = cv2.resize(plt.imread('./HW1_Q1/marilyn.bmp'),(150,150))
```

Using the *imread()* functionality in *matplotlib* we can scan/read the image matrices and by using *resize()* function in *cv2* , we can resize the image to fixed dimensions so that we can perform addition on matrices having equal dimensions.

```
plt.imshow(img1)
plt.figure()
plt.imshow(img2)
plt.figure()
```

By using *imshow()* of *matplotlib* we can output the scanned images. The outputs are as follows.





Splitting into R,G,B channels:

The input images can contain R,G,B channels. We need to split the given images into these 3 channels so that we can apply *Fourier Transform* on each of them.

```
img1r = img1[:,:,0]
img1g = img1[:,:,1]
img1b = img1[:,:,2]

img2r = img2[:,:,0]
img2g = img2[:,:,1]
img2b = img2[:,:,2]
```

Converting to Frequency Domain:

Since we've separated the images into 3 channels we can apply *fft()* from *numpy* which is an implementation of *Fast Fourier Transform* on each of them. Then we use *fftshift()* of *numpy* to shift all high values to the centre of the matrix.

```
img1rFT = np.fft.fft2(img1r)
img1rFT = np.fft.fftshift(img1rFT)
img1gFT = np.fft.fft2(img1g)
img1gFT = np.fft.fftshift(img1gFT)
img1bFT = np.fft.fft2(img1b)
img1bFT = np.fft.fftshift(img1bFT)
img2rFT = np.fft.fft2(img2r)
img2rFT = np.fft.fftshift(img2rFT)
img2gFT = np.fft.fft2(img2g)
img2gFT = np.fft.fftshift(img2gFT)
img2bFT = np.fft.fft2(img2b)
img2bFT = np.fft.fftshift(img2bFT)
```

Low pass FIlter on one image:

After converting the image channels to *Frequency Domain* , we need to apply Low pass Filter on one image. This can be done by using a threshold called *beta.* If the absolute value in the frequency domain matrix of the image is greater than this *beta* then attenuate it by making it 0.

```
img2rFT[np.absolute(img2rFT)>beta] = 0
img2gFT[np.absolute(img2gFT)>beta] = 0
img2bFT[np.absolute(img2bFT)>beta] = 0
```

After applying the filter , using *ifftshift()* in *numpy* we again bring the values back to their original position.

```
img2rFT = np.fft.ifftshift(img2rFT)
img2gFT = np.fft.ifftshift(img2gFT)
img2bFT = np.fft.ifftshift(img2bFT)
```

High pass filter on other image:

After converting the image channels to *Frequency Domain* , we need to apply High pass Filter on other image. This can be done by using a threshold called *alpha.* If the absolute value in the frequency domain matrix of the image is less than this *alpha* then attenuate it by making it 0.

```
img1rFT[np.absolute(img1rFT)<alpha] = 0
img1gFT[np.absolute(img1gFT)<alpha] = 0
img1bFT[np.absolute(img1bFT)<alpha] = 0
```

After applying the filter , using *ifftshift()* in *numpy* we again bring the values back to their original position.

```
img2rFT = np.fft.ifftshift(img2rFT)
img2gFT = np.fft.ifftshift(img2gFT)
img2bFT = np.fft.ifftshift(img2bFT)
```

Combine these two results:

As we now have high pass filtered 1st image and low pass filtered second image all we have to do is combine these two to create a new *Hybrid Image.*

```
img3rFT = np.zeros(img1rFT.shape,dtype=complex)
img3gFT = np.zeros(img1gFT.shape,dtype=complex)
img3bFT = np.zeros(img1bFT.shape,dtype=complex)

img3rFT = img1rFT + img2rFT
img3gFT = img1gFT + img2gFT
img3bFT = img1bFT + img2bFT
```

Inverse Fourier Transform:

As we have acquired all the three channels of required result , we have to transform them from frequency domain to spatial domain by using *ifft()* of *numpy*

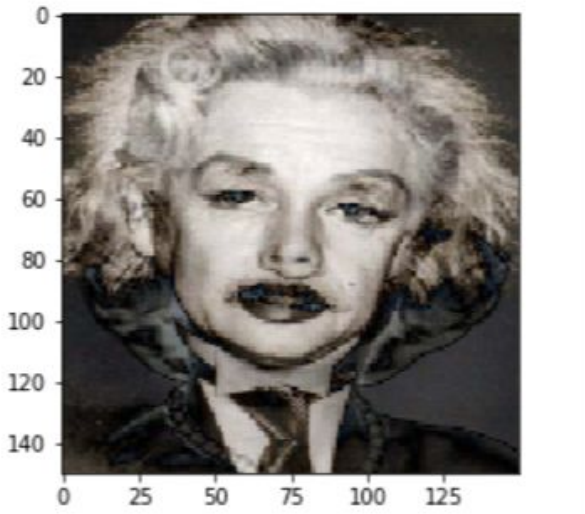
and have to combine all the three channels into one pixel and append it to the final image matrix.

```
img3r = abs(np.fft.ifft2(img3rFT))
img3g = abs(np.fft.ifft2(img3gFT))
img3b = abs(np.fft.ifft2(img3bFT))

img3=[]
for i in range(img3r.shape[0]):
    img3.append([])
    for j in range(img3r.shape[1]):
        img3[i].append([img3r[i][j],img3g[i][j],img3b[i][j]])
img3=np.array(img3)
```

Output:

Using *imshow()* function of *matplotlib* we can output the resultant hybrid image.

```
plt.imshow(np.absolute(img3) / np.max(np.absolute(img3)))
```



Similarly by using different *alpha* and *beta* values , we can find out the resultant hybrid images of their pictures.

OBSERVATIONS

The edges increase/decrease based on the alpha beta values taken.

Low pass filter removes edges.

Higher the alpha, more smooth the image becomes.

High pass filter sharpens the image.

Higher the beta value, more evident the edges become.

2. CORNER DETECTION

AIM

Implementation of different versions of corner detection algorithms on the three provided images. Namely, Shi-Tomasi and Harris corner detection techniques have to be implemented.

PROCEDURE

Importing necessary Libraries:

```
from skimage.io import imread
from skimage.color import rgb2gray
import matplotlib
import matplotlib.pyplot as plt
import cv2
```

Opening and displaying the image:

```
img = imread('Image3.jpg')
#imggray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
imggray = rgb2gray(img)
plt.imshow(imggray, cmap="gray")
#plt.imshow(img)
plt.axis("off")

plt.show()
```

Using the *imread()* functionality in *matplotlib* we can scan/read the image matrices in *cv2*.

The image is converted to gray scale for further computation using *rgb2gray(img)*.

Using *imshow()* the image (now converted to gray) is displayed as follows:

Calculating gradient in x and y direction respectively:

```
from scipy import signal as sig
import numpy as np

def gradient_x(imggray):
    ##Sobel operator kernels.
    sobel_x = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
    return sig.convolve2d(imggray, sobel_x, mode='same')
def gradient_y(imggray):
    sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    return sig.convolve2d(imggray, sobel_y, mode='same')

I_x = gradient_x(imggray)
I_y = gradient_y(imggray)
```

The gradient of the gray colored image is now calculated in both the x and y direction respectively by running functions *gradient_x() and gradient_y(). Sobel* operator ([-1, 0, 1], [-2, 0, 2], [-1, 0, 1]) is used and convolution is performed to calculate *I_x* and *I_y*.

```
Ixx = I_x**2
Ixy = I_y*I_x
Iyy = I_y**2
```

To calculate the *Hessian Matrix* we need [*Ixx, Ixy, Ixy, Iyy]* that are calculated above.

Corner detection using Shi-Tomasi:

```
import numpy as np
from numpy import linalg as LA

img_copy2=cv2.cvtColor(imggray, cv2.COLOR_GRAY2RGB)
h,w = imggray.shape
shi_tomasi_response = []
D = np.zeros((2,2))
#print(h,w)
img_resp = np.zeros((h,w))
window_size = 5
offset = window_size//2
for y in range(offset, h-offset):
    for x in range(offset, w-offset):
        Sxx = np.sum(Ixx[y-offset:y+1+offset, x-offset:x+1+offset])
        Syy = np.sum(Iyy[y-offset:y+1+offset, x-offset:x+1+offset])
        Sxy = np.sum(Ixy[y-offset:y+1+offset, x-offset:x+1+offset])
        D[0, 0] = Sxx
        D[0, 1] = Sxy
        D[1, 0] = Sxy
        D[1, 1] = Syy
        val, vec = LA.eig(D)
        e = min(val[0], val[1])
        if(e>100000):
            img_resp[y][x] = e
            cv2.circle(img_copy2,(x,y),2,(0, 0, 255),-1)
        else:
            img_resp[y][x] = 0
        #print(e)
        shi_tomasi_response.append((x,y,e))
#    print(Ixx)
cv2.imwrite("HW_1_Q_2_st_Solns/image_3_before.jpg",img_copy2)
plt.imshow(img_copy2)
plt.show()
```

The method used by Shi-Tomasi involves summation of terms *Ixx, Ixy and Iyy* stored respectively in *Sxx, Sxy and Syy* used to form *matrix D.* After that respective

eigenvalues of *matrix D* are calculated and the minimum one is stored in *variable e.* If the value of the minimum eigenvalue e is greater than the *threshold* value (100000 in this case) it is stored else rejected. If stored - a circle is made on the original image depicting a corner.

Having performed the procedure on the whole image, it is displayed using *imshow()*:

Shi-Tomasi corner detection without non maximum suppression

Non maximum suppression:

```
img_copy=cv2.cvtColor(imggray, cv2.COLOR_GRAY2RGB)
#print(img_copy.shape)
#img_copy = img
plt.imshow(img_copy)
ws = 5
#for response in shi_tomasi_response:
#   print(response)
"""for response in shi_tomasi_response:
    x, y, e = response
    #1000000000000
    if e > 100000:
        #img_copy[y,x] = 250
        img_copy[y,x] = (250, 0 ,0)
        #print(img_copy[y,x])
plt.imshow(img_copy, cmap="gray")
plt.show()
"""
I = np.zeros((ws,ws))
for j in range(ws//2,h-(ws//2)):
    for i in range(ws//2,w-(ws//2)):
        check_arr = np.array(img_resp[j:j+ws, i:i+ws])
        maxpix = np.amax(check_arr)
        #print(maxpix)
        if(maxpix!=0):
            maxloc = np.where(check_arr == np.amax(check_arr))
            if(check_arr[ws//2][ws//2]==maxpix):
                #img_copy[j+maxloc[0], i+maxloc[1]] = (255, 0, 0)
                cv2.circle(img_copy,(i+maxloc[1],j+maxloc[0]),2,(0, 0, 255),-1)
cv2.imwrite("HW_1_Q_2_st_Solns/image_3_after.jpg",img_copy)
plt.imshow(img_copy)
plt.show()
```

A window size *ws = 5* is defined and for that window the max value of the previously stored eigenvalues is saved in *maxpix.*

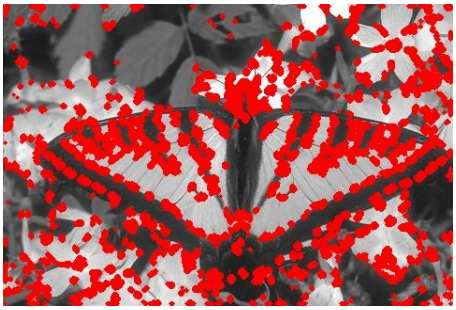
If that pixel happens to be the centre pixel of that window it is considered as a corner and a circle is made on the original image to depict the same. This method is called non-max suppression.

Finally the image is displayed using *imshow()*:

Shi-Tomasi corner detection after non max suppression

Output:

Having performed Shi-Tomasi technique on other two images using the method described above, the results are as follows:



Before non max suppression

After non max suppression

Before non max suppression

After non max suppression

Corner detection using Harris technique:

The initial procedure for both Shi-Tomasi and Harris is the same:

```
from skimage.io import imread
from skimage.color import rgb2gray
import matplotlib
import matplotlib.pyplot as plt
import cv2
img = imread('Image3.jpg')
#imggray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
imggray = rgb2gray(img)
plt.imshow(imggray, cmap="gray")
plt.axis("off")

plt.show()
```

```
from scipy import signal as sig
import numpy as np

def gradient_x(imggray):
    ##Sobel operator kernels.
    sobel_x = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
    return sig.convolve2d(imggray, sobel_x, mode='same')
def gradient_y(imggray):
    sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    return sig.convolve2d(imggray, sobel_y, mode='same')

I_x = gradient_x(imggray)
I_y = gradient_y(imggray)
```

```
Ixx = I_x**2
Ixy = I_y*I_x
Iyy = I_y**2
```

The Harris technique is now used:

```
alpha = 0.03

height, width = imggray.shape
img_copy2=cv2.cvtColor(imggray, cv2.COLOR_GRAY2RGB)

harris_response = []
window_size = 3
offset = window_size//2
img_resp = np.zeros((height,width))
for y in range(offset, height-offset):
    for x in range(offset, width-offset):
        Sxx = np.sum(Ixx[y-offset:y+1+offset, x-offset:x+1+offset])
        Syy = np.sum(Iyy[y-offset:y+1+offset, x-offset:x+1+offset])
        Sxy = np.sum(Ixy[y-offset:y+1+offset, x-offset:x+1+offset])

        #Find determinant and trace, use to get corner response
        det = (Sxx * Syy) - (Sxy**2)
        trace = Sxx + Syy
        r = det - alpha*(trace**2)
        if(r>10000000000):
            img_resp[y][x] = r
            cv2.circle(img_copy2,(x,y),2,(0, 0, 255),-1)
        else :
            img_resp[y][x] = 0
        harris_response.append((x,y,r))
cv2.imwrite("HW_1_Q_2_h_Solns/image_3_h_before.jpg",img_copy2)
plt.imshow(img_copy2)
plt.show()
```

Instead of finding the eigenvalues which is both time consuming and computationally more expensive, the *determinant and trace* of the matrix are calculated and put in formula:

r = det - alpha(trace^2)*

Only if the value of r is greater than the threshold (in this case 10000000000) then point is marked as a corner. The image is then displayed:

Harris corner detection without non maximum suppression

Non maximum suppression:

It is the same in both Shi-Tomasi and Harris techniques:

```
img_copy=cv2.cvtColor(imggray, cv2.COLOR_GRAY2RGB)
#print(img_copy.shape)
#img_copy = img
#plt.imshow(img_copy)
"""for response in harris_response:
    x, y, r = response
for y in range(0,height):
    for x in range(0,width):
        #1000000000000
        if img_resp[y][x] > 50000000000:
            #img_copy[y,x] = 250
            img_copy[y,x] = (250, 0 ,0)
            #print(img_copy[y,x]
"""
ws = 5
I = np.zeros((ws,ws))
for j in range(ws//2,height-(ws//2)):
    for i in range(ws//2,width-(ws//2)):
        check_arr = np.array(img_resp[j:j+ws, i:i+ws])
        maxpix = np.amax(check_arr)
        #print(maxpix)
        if(maxpix!=0):
            maxloc = np.where(check_arr == np.amax(check_arr))
            if(check_arr[ws//2][ws//2]==maxpix):
                #img_copy[j,i] = (0, 255, 0)
                cv2.circle(img_copy,(i+maxloc[1],j+maxloc[0]),2,(0, 0, 255),-1)
cv2.imwrite("HW_1_Q_2_h_Solns/image_3_h_after.jpg",img_copy)
plt.imshow(img_copy)
plt.show()
```

Harris corner detection after non max suppression

Output:

Having performed Harris technique on other two images using the method described above, the results are as follows:

Before non max suppression

After non max suppression

Before non max suppression

After non max suppression

OBSERVATIONS

The accuracy obtained from shi tomasi corner detection algorithm is higher than that obtained from harris corner detection algorithm. But shi tomasi consumes a lot of time and is computationally complex as the eigen value needs to be calculated every single time where as harris computes f value taking into account determinant, trace of hessian matrix leading to comparatively lesser computation.

3. SCALE-SPACE BLOB DETECTION

AIM

Aim is the implementation of a Laplacian Blob Detector.

PROCEDURE

Importing necessary Libraries:

```
from skimage.io import imread
from skimage.color import rgb2gray
import matplotlib
import matplotlib.pyplot as plt
import cv2
from skimage.feature import peak_local_max
```

Opening the image:

```
img = cv2.imread('butterfly.jpg')
imggray = cv2.imread('butterfly.jpg',0)
#imggray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
#imggray = rgb2gray(img)

plt.imshow(imggray, cmap="gray")
print(imggray.shape)
plt.axis("off")

plt.show()
```

Gray input image

Laplacian filter:

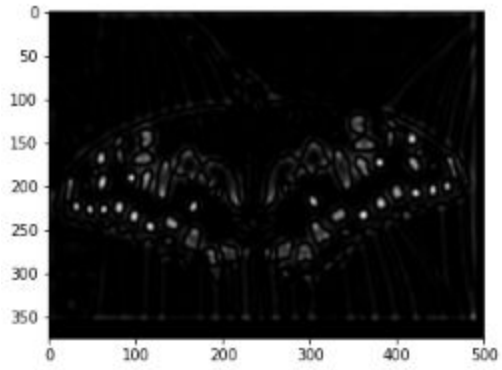
```
import math
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from scipy import signal as sig
import cv2

print(imggray.shape)
h,w = imggray.shape
def filter(sigma):
    s = sigma
    f = int(6*s)
    if(f%2!=0):
        F = np.zeros((f,f))
        for i in range(0,f):
            for j in range(0,f):
                x = i-(f//2)
                y = j-(f//2)
                F[i][j] = ((x**2)+(y**2)-2*(s**2))*math.exp(-((x**2)+(y**2))/(2*(s**2)))*(s**2)
    else:
        F = np.zeros((f+1,f+1))
        for i in range(0,f+1):
            for j in range(0,f+1):
                x = i-((f+1)//2)
                y = j-((f+1)//2)
                F[i][j] = ((x**2)+(y**2)-2*(s**2))*math.exp(-((x**2)+(y**2))/(2*(s**2)))*(s**2)
    return F
```

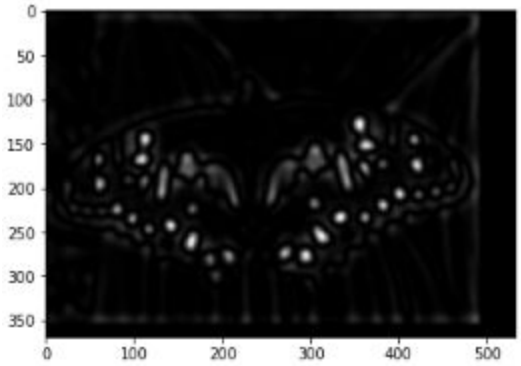
Here a function call is defined where the laplacian is calculated:

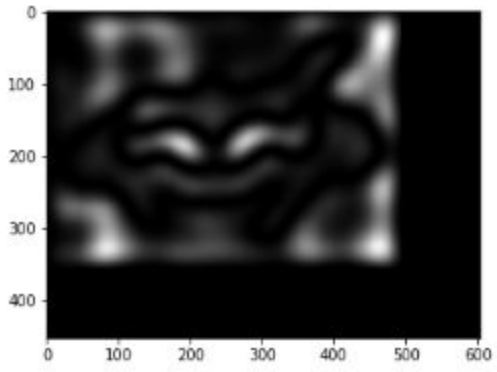
L = (x^2)+(y^2)-2(sigma^2)*e^ -((x^2)+(y^2)/(2*(sigma^2)))*(sigma^2)*

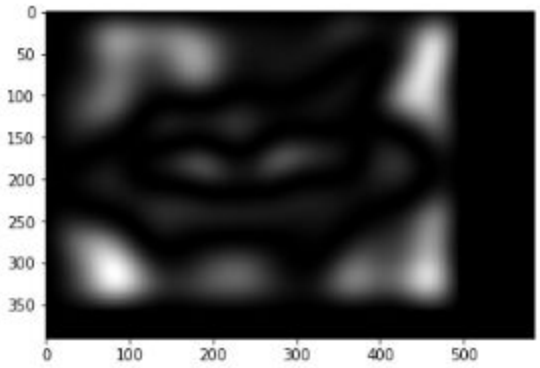
```
sigma = 4
response = []
for i in range(0,10):
    sigma = 4*(1.25**i)
    #print(sigma)
    f = int(6*sigma)
    ws = f
    if(f%2!=0):
        L = np.zeros((f,f))
        L = filter(sigma)
    else:
        L = np.zeros((f+1,f+1))
        L = filter(sigma)
    #plt.imshow(L,cmap="gray")
    #plt.figure()
    if(ws%2==0):
        ws+=1
    NI = sig.convolve2d(imggray, L, mode='same')
    NI2 = np.square(NI)
    #print(NI2.shape)
    color = [0, 0, 0]
    h,w = NI2.shape
    if(h%ws!=0):
        top = 0
        bottom = ws-(h%ws)
    else:
        top = 0
        bottom = 0
    if(w%ws!=0):
        right = ws-(w%ws)
        left = 0
    else:
        right = 0
        left = 0
    #print(ws,h%ws,w%ws)
    #print(top,bottom,right,left)
    NI3 = cv2.copyMakeBorder(NI2, top, bottom, left, right, cv2.BORDER_CONSTANT,value=color)
    plt.imshow(NI3,cmap="gray")
    #print(NI3.shape)
    response.append(NI3)
    plt.figure()
```

Variable NI stores the converted gray image. *Variable NI2* stores the squares (since square of laplacian is needed). *Variable NI3* stores the *laplacian* final answer.









Blobs:

```
img_copy = img.copy()
sigma = 4
w_size = np.zeros(imggray.shape)
radius = []
for a in range(0,10):
    peakvalues = peak_local_max(response[a],min_distance = (a+1)*5)
    radius = int(sigma*(1.25**a)*math.sqrt(2))
    win_size = int(6*sigma*(1.25**a))
    if(win_size%2==0):
        win_size+=1
    for b in peakvalues:
        cv2.circle(img_copy,(b[1],b[0]),radius,(0, 0, 255),2)
        w_size[b[0],b[1]] = win_size
cv2.imwrite("butterfly_b4_blob.jpg",img_copy)
plt.imshow(img_copy)
plt.show()
```

Blobs without non max suppression

Applying non max suppression:

```
h,w = imggray.shape
for a in range(0,h):
    for b in range(0,w):
        if(w_size[a, b]!=0):
            win_size_2 = int(w_size[a, b]/2)-int(w_size[a, b]/3)
            for c in range(a-win_size_2,a+win_size_2):
                for d in range(b-win_size_2,b+win_size_2):
                    if((c>=0 and c<h) and (d>=0 and d<w)):
                        if(w_size[c, d] < w_size[a, b]):
                            w_size[c, d] = 0
```

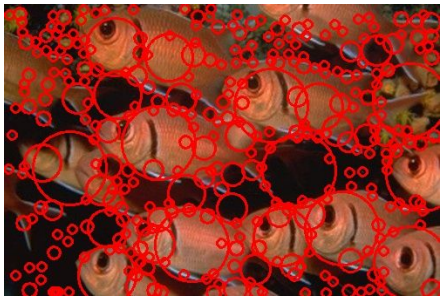
Size a little smaller than the window size is taken and maximum is calculated. This helps even include some overlapping blobs and makes output more accurate.

Displaying final blob image:

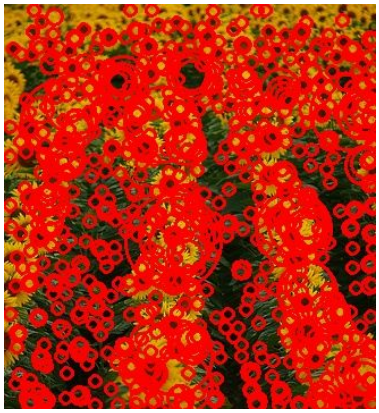
Blobs after non max suppression

Output:

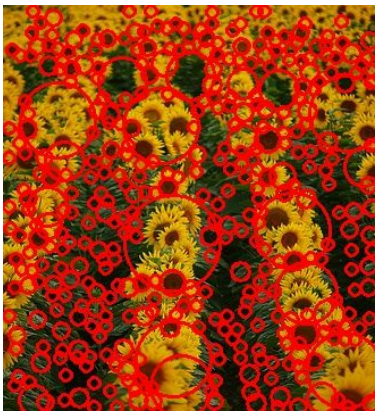
Before non max suppression



After non max suppression



Before non max suppression



After non max suppression

OBSERVATIONS

For different values of sigma, we generally get various sizes of blobs.Like for smaller values of sigma, small and more number of blobs are obtained whereas for larger values, big blobs are obtained.