Project 1 of CSE 473/573

Sahil Suhas Pathak
Department of Computer Science and Engineering
University at Buffalo
Buffalo, NY, 14214
sahilsuh@buffalo.edu
UBITName: sahilsuh & Person Number: 50289739

Abstract

The report addresses the implementation of Edge Detection, Keypoint Detection and Cursor Detection in detail, illustrating necessary findings and conclusions drawn from their execution, using Python and OpenCV.

1 Edge Detection

We are using Sobel Operator to detect edges in the image. The operator uses two 3×3 kernels which are convolved with the original image.

1.1 Implementation - Source Code

```
#Required imports
import cv2
import numpy as np

#Reading the Image
sample = cv2.imread('task1.png',0)
cv2.imshow('image',sample)
cv2.waitKey(0)
cv2.destroyAllWindows()

#Zero Padding
h, w = sample.shape[:2]
h = h + 2
w = w + 2
pad_img = [[0 for x in range(w)] for y in range(h)]

for i in range(len(sample)):
    for j in range(len(sample[0])):
        pad_img[i+1][j+1] = sample[i][j]
pad_img = np.asarray(pad_img)
```

```
#Methods
#Flips the kernel
def flip_operator(kernel):
    kernel_copy = [[0 for x in range(kernel.shape[1])] for y in range(kernel.shape[0])]
    #kernel_copy = kernel.copy()
    for i in range(kernel.shape[0]):
        for j in range(kernel.shape[1]):
            kernel_copy[i][j] = kernel[kernel.shape[0]-i-1][kernel.shape[1]-j-1]
    kernel_copy = np.asarray(kernel_copy)
    return kernel_copy

#Convolution Logic
def convolution(image, kernel):
    #Flipping the kernel
    kernel = flip_operator(kernel)

    img_height = image.shape[0]
    img_width = image.shape[1]

    kernel_height = kernel.shape[0]
    kernel_width = kernel.shape[1]

    h = kernel_height//2
    w = kernel_width//2

    conv_result = [[0 for x in range(img_width)] for y in range(img_height)]

    for i in range(h, img_height-h):
        for j in range(w, img_width-w):
            sum = 0
            for m in range(kernel_height):
                for n in range(kernel_width):
                    sum = (sum + kernel[m][n]*image[i-h+m][j-w+n])

            conv_result[i][j] = sum
    conv_result = np.asarray(conv_result)
    return conv_result
```

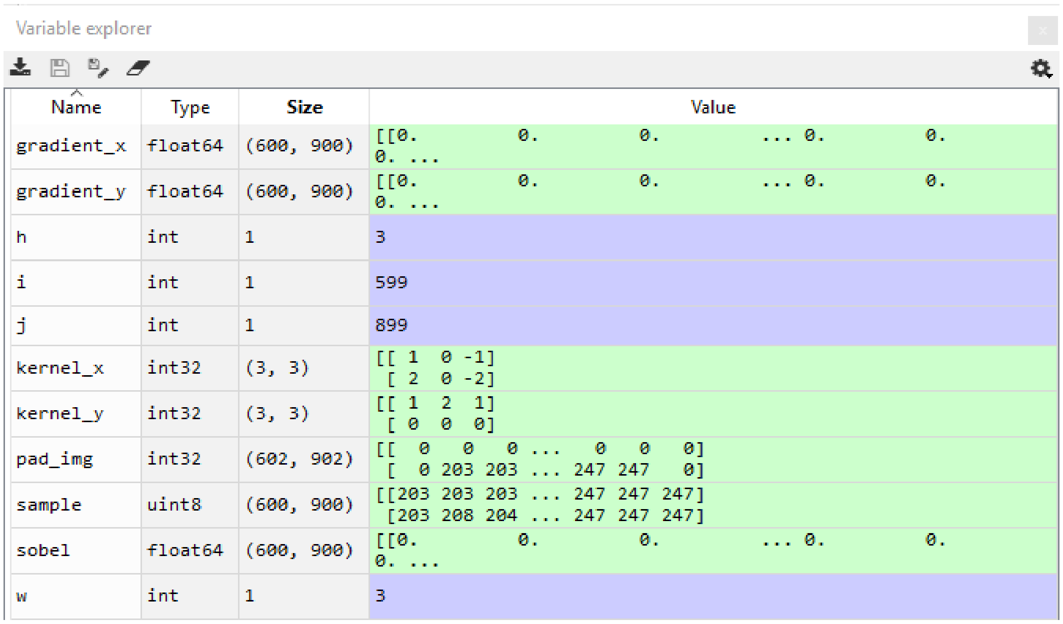
```
#Defines the output image; combination of gradient_x and gradient_y
def output(img1, img2):
    h, w = img1.shape
    result = [[0 for x in range(w)] for y in range(h)]
    for i in range(img1.shape[0]):
        for j in range(img1.shape[1]):
            result[i][j] = (img1[i][j]**2 + img2[i][j]**2)**(1/2)
            if(result[i][j] > 255):
                result[i][j] = 255
            elif(result[i][j] < 0):
                result[i][j] =  0
    result = np.asarray(result)
    return result

#Returns the maximum value from gradient_y/gradient_x
def maximum(gradient):
   max = gradient[0][0]
   for i in range(len(gradient)):
      for j in range(len(gradient[0])):
         if (max < gradient[i][j]):
            max = gradient[i][j]
   return max

#Returns the gradient_y/gradient_x with absolute values of gradient_y/gradient_x
def absolute_value(gradient):
    for i in range(len(gradient)):
        for j in range(len(gradient[0])):
            if(gradient[i][j] < 0):
                gradient[i][j] *= -1
            else:
                continue
    return gradient

#Plotting gradient_y
w, h = 3, 3
kernel_y = [[0 for x in range(w)] for y in range(h)]
kernel_y = np.asarray(kernel_y)
kernel_y[0,0] = 1
```

```
kernel_y[0,1] = 2
kernel_y[0,2] = 1
kernel_y[1,0] = 0
kernel_y[1,1] = 0
kernel_y[1,2] = 0
kernel_y[2,0] = -1
kernel_y[2,1] = -2
kernel_y[2,2] = -1
gradient_y = convolution(sample, kernel_y)
#print(gradient_y)
gradient_y = absolute_value(gradient_y) / maximum(absolute_value(gradient_y))
cv2.imshow("gradient_y",gradient_y)
cv2.waitKey(0)
cv2.destroyAllWindows()

#Plotting gradient_x
w, h = 3, 3
kernel_x = [[0 for x in range(w)] for y in range(h)]
kernel_x = np.asarray(kernel_x)
kernel_x[0,0] = 1
kernel_x[0,1] = 0
kernel_x[0,2] = -1
kernel_x[1,0] = 2
kernel_x[1,1] = 0
kernel_x[1,2] = -2
kernel_x[2,0] = 1
kernel_x[2,1] = 0
kernel_x[2,2] = -1
gradient_x = convolution(sample, kernel_x)
#print(gradient_x)
gradient_x = absolute_value(gradient_x) / maximum(absolute_value(gradient_x))
cv2.imshow("gradient_x",gradient_x)
cv2.waitKey(0)
cv2.destroyAllWindows()

#Plotting final output image
sobel = output(gradient_x, gradient_y)
cv2.imshow("Output_Image", sobel)
```

cv2.waitKey(0)

cv2.destroyAllWindows()

1.2 Variable Explorer



Variable explorer

Name	Type	Size	Value
gradient_x	float64	(600, 900)	[[0. 0. 0. ... 0. 0. 0. ...
gradient_y	float64	(600, 900)	[[0. 0. 0. ... 0. 0. 0. ...
h	int	1	3
i	int	1	599
j	int	1	899
kernel_x	int32	(3, 3)	[[1 0 -1] [2 0 -2]
kernel_y	int32	(3, 3)	[[1 2 1] [0 0 0]
pad_img	int32	(602, 902)	[[0 0 0 ... 0 0 0] [0 203 203 ... 247 247 0]
sample	uint8	(600, 900)	[[203 203 203 ... 247 247 247] [203 208 204 ... 247 247 247]
sobel	float64	(600, 900)	[[0. 0. 0. ... 0. 0. 0. ...
w	int	1	3

1.3 Output

1.3.1 gradient_y

1.3.2 gradient_x

1.3.3 Combined Image of gradient_y and gradient_x

2 Keypoint Detection

Following is the program to detect keypoints in an image according to the following steps, which are also the first three steps of Scale-Invariant Feature Transform (SIFT).

Step 1: Generate four octaves. Each octave is composed of five images blurred using Gaussian kernels.

Step 2: Compute Difference of Gaussian (DoG) for all four octaves.

Step 3: Detect keypoints which are located at the maxima or minima of the DoG images.

2.1 Implementation – Source Code

```
import math
import numpy as np
import cv2

img = cv2.imread('task2.jpg',0)

#Zero Padding
h, w = img.shape[:2]
h = h + 6
w = w + 6
pad_img = [[0 for x in range(w)] for y in range(h)]

for i in range(len(img)):
    for j in range(len(img[0])):
        pad_img[i+1][j+1] = img[i][j]

pad_img = np.asarray(pad_img)

#Flips the kernel
def flip_operator(kernel):
    kernel_copy = [[0 for x in range(kernel.shape[1])] for y in range(kernel.shape[0])]
    #kernel_copy = kernel.copy()
    for i in range(kernel.shape[0]):
        for j in range(kernel.shape[1]):
            kernel_copy[i][j] = kernel[kernel.shape[0]-i-1][kernel.shape[1]-j-1]
    kernel_copy = np.asarray(kernel_copy)
    return kernel_copy

#Convolution Logic
def conv(image, kernel):
```

```
    #Flipping the kernel
    kernel = flip_operator(kernel)

    img_height = image.shape[0]
    img_width = image.shape[1]

    kernel_height = kernel.shape[0]
    kernel_width = kernel.shape[1]

    h = kernel_height//2
    w = kernel_width//2

    conv_result = [[0 for x in range(img_width)] for y in range(img_height)]

    for i in range(h, img_height-h):
        for j in range(w, img_width-w):
            sum = 0
            for m in range(kernel_height):
                for n in range(kernel_width):
                    sum = (sum + kernel[m][n]*image[i-h+m][j-w+n])

            conv_result[i][j] = sum
    conv_result = np.asarray(conv_result)
    return conv_result

#Defines the Gaussian Kernel
def gau_kernel(sigma):
    w, h = 7, 7;
    gau_mat = [[0 for x in range(w)] for y in range(h)]
    for i in range(0,7):
        for j in range(0,7):
            gau_mat[i][j]=(1/(2*math.pi*sigma*sigma))*(math.exp(-(((j-3)**2 + (3-
i)**2)/(2*sigma*sigma))))
    gau_mat = np.asarray(gau_mat)
    return gau_mat

#Building octave for every level
def calculate_octave(img, sigma):
    g1 = gau_kernel(sigma[0])
```

```
    ga = conv(img,g1)
    ga = np.asarray(ga)

    g2 = gau_kernel(sigma[1])
    gb = conv(img,g2)
    gb = np.asarray(gb)

    g3 = gau_kernel(sigma[2])
    gc = conv(img,g3)
    gc = np.asarray(gc)

    g4 = gau_kernel(sigma[3])
    gd = conv(img,g4)
    gd = np.asarray(gd)

    g5 = gau_kernel(sigma[4])
    ge = conv(img,g5)
    ge = np.asarray(ge)

    dog1 = gb-ga
    dog2 = gc-gb
    dog3 = gd-gc
    dog4 = ge-gd
    oct = [dog1,dog2,dog3,dog4]
return oct

sigma1=[0.707107,1.000000,1.414214,2.000000,2.828427]
sigma2=[1.414214,2.000000,2.828427,4.000000,5.656854]
sigma3=[2.828427,4.000000,5.656854,8.000000,11.313708]
sigma4=[5.656854,8.000000,11.313708,16.000000,22.627417]


pad_img_copy = [pad_img]

resized_pad_img = pad_img[::2,::2]
pad_img_copy.append(resized_pad_img)

resized_pad_img = resized_pad_img[::2,::2]
```

```
pad_img_copy.append(resized_pad_img)

resized_pad_img = resized_pad_img[::2,::2]
pad_img_copy.append(resized_pad_img)

#Calculating octaves i.e 1,2,3 and 4; Returned value is a list of difference of gaussian's 1,2,3 and 4
oct1 = calculate_octave(pad_img_copy[0],sigma1)
oct2 = calculate_octave(pad_img_copy[1],sigma2)
oct3 = calculate_octave(pad_img_copy[2],sigma3)
oct4 = calculate_octave(pad_img_copy[3],sigma4)



#Detects Keypoints and returns a list of keypoints for a particular octave
def detect(oct,oct_num,scale_factor):
    #Scale factor is to multiply the co-ordinates by a scaling factor inorder to plot those keypoints got from resized image onto the original image
    points = []
    img = oct[oct_num]
    for i in range(3,len(img) - 8, 1):
        for j in range(3,len(img[0]) - 8, 1):
            mid = img[i+1][j+1]
            check = neighbor(oct,i+1,j+1,oct_num)
            min = True
            max = True
            for n in check:
                if n >= mid:
                    max = False
                if n <= mid:
                    min = False
            if (max or min):
                    points.append([(i+1)*scale_factor,(j+1)*scale_factor])
    return points

#Use to check the current, previous and the next pixel values of respective difference of gaussian's
def neighbor(octave,x,y,oct_num):
    #oct_num is to indicate which should be the current difference of gaussian
    img = octave[oct_num]
```

```
    neighbor = [img[x-1,y],
                img[x+1,y],
                img[x,y+1],
                img[x,y-1],
                img[x+1,y+1],
                img[x+1,y-1],
                img[x-1,y+1],
                img[x-1,y-1]]

    prev = octave[oct_num - 1]
    neighbor+=[prev[x,y],
               prev[x+1,y],
               prev[x-1,y],
               prev[x,y+1],
               prev[x,y-1],
               prev[x+1,y+1],
               prev[x+1,y-1],
               prev[x-1,y+1],
               prev[x-1,y-1]]

    next = octave[oct_num + 1]
    neighbor+=[next[x,y],
               next[x+1,y],
               next[x-1,y],
               next[x,y+1],
               next[x,y-1],
               next[x+1,y+1],
               next[x+1,y-1],
               next[x-1,y+1],
               next[x-1,y-1]]
    return neighbor

#Calculating respective keypoints for octave 1, octave 2, octave 3 and octave 4
point11 = detect(oct1,1,1) #Parameters are Octave, Current DOG, Scale Factor
point12 = detect(oct1,2,1)
point21 = detect(oct2,1,2)
point22 = detect(oct2,2,2)
point31 = detect(oct3,1,4)
```

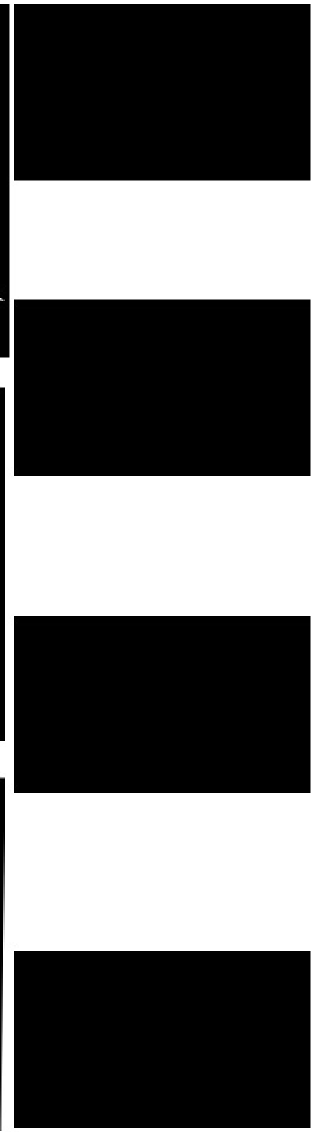
```
point32 = detect(oct3,2,4)
point41 = detect(oct4,1,8)
point42 = detect(oct4,2,8)

#Storing all keypoints from Octave 1,2,3,4 in array named 'points'
points=[]
points+=point11
points+=point12
points+=point21
points+=point22
points+=point31
points+=point32
points+=point41
points+=point42

#Finding coordinates of the five left-most detected keypoints when the origin is set to be the top-left corner
sorted_points = []
for i in range(len(points)):
    x = points[i][0]
    y = points[i][1]
    euc_dis = (x**2 + y**2)**1/2 #Calculating Euclidean Distance with respect to origin
    sorted_points.append([x,y,euc_dis])
    img[x,y] = 255

#Plotting keypoints on the GrayScale image
cv2.imshow("Keypoints.jpg",img)
cv2.waitKey(0)

sorted_points = sorted(sorted_points,key=lambda l:l[2], reverse=False)
print("five left-most detected keypoints: ")
for i in range(5):
    x = sorted_points[i][0]
    y = sorted_points[i][1]
    print(x,y)
```

2.2 Images of the second and third octave specifying their resolution (width and Height)

Following are the Images of second and third octave with their resolution

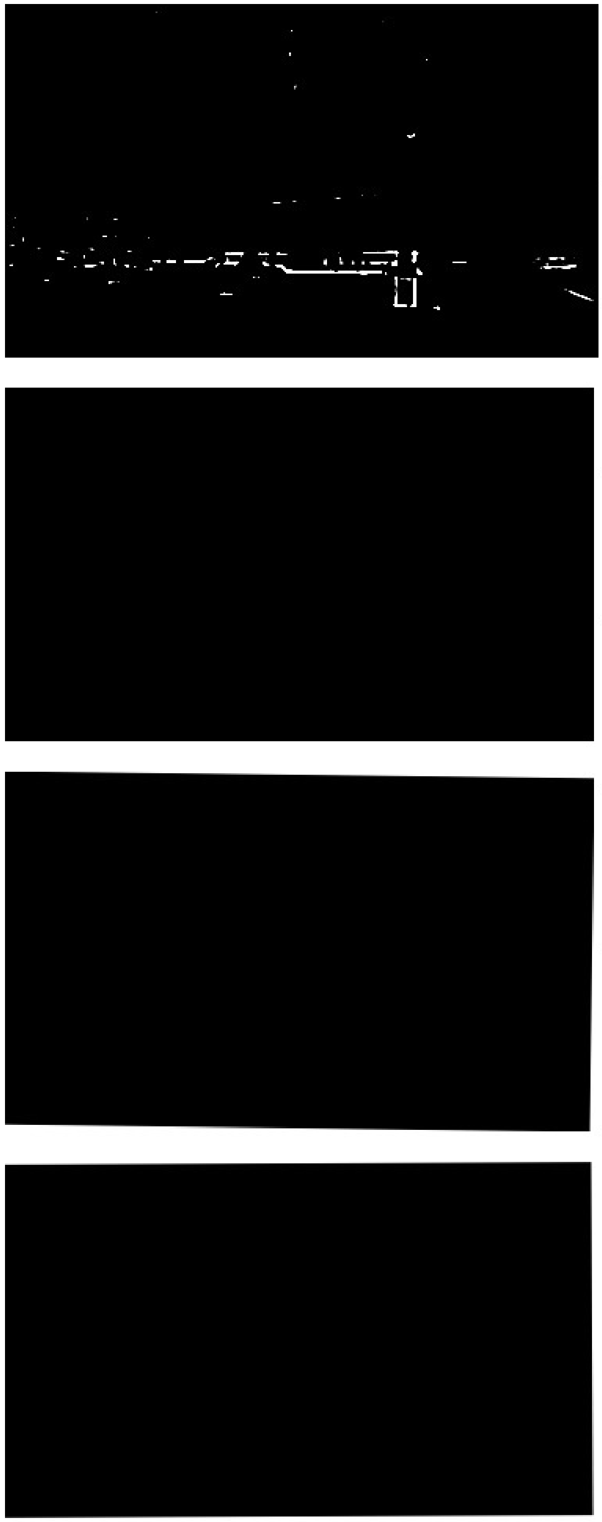
Images of the second and third octave specifying their resolution (width and Height)

2.3 DoG images obtained using the second and third octave



Octave 3

DoG 1 , DoG2 , DoG 3 and DoG 4 Images



Octave 2

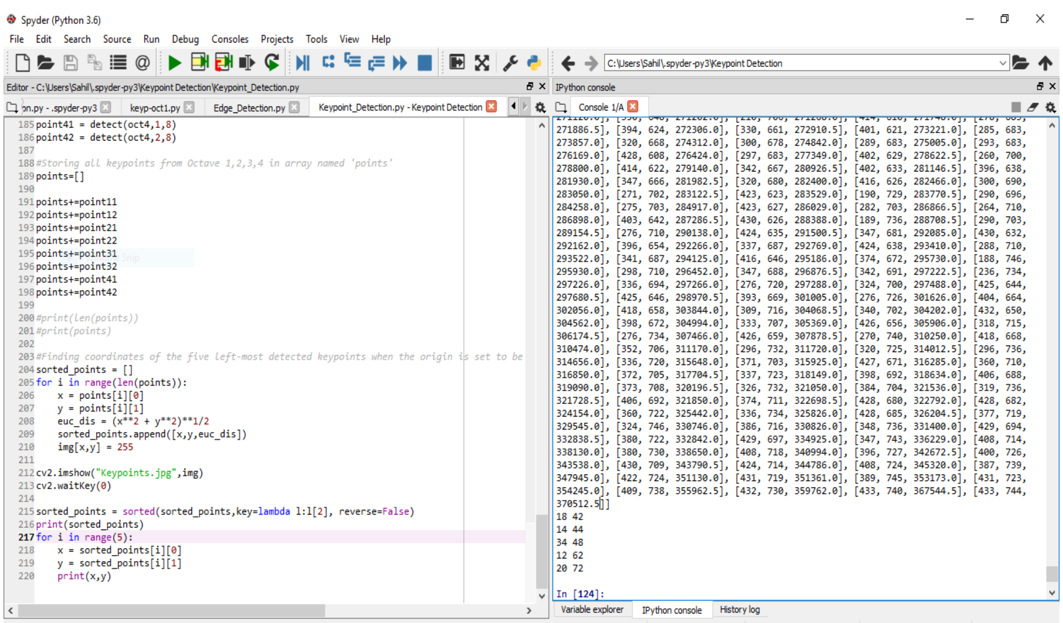
DoG 1, DoG 2, DoG 3 and DoG 4 Images

2.4 Detected keypoints depicted using white dots on the original image

2.5 Coordinates of the five left-most detected keypoints when the origin is set to be at the top-left corner

With respect to the origin which is at the top leftmost corner and considering it as (0,0) we calculate Euclidean Distance with respect to the origin and the Keypoint co-ordinates. The first 5 Keypoints having the minimum Euclidean Distance are printed as output on the console.

Five left-most detected Keypoints: (18, 42), (14, 44), (34, 48), (12, 62), (20, 72)



3 Cursor Detection

3.1 Proposed Methodology

Step 1: Storing the set of images with similar cursor into a list

Step 2: Traversing the list and reading each and every image at a time

Step 3: Reading the template in every iteration, templates named as ‘template_1_hand.png’ for hand cursor, ‘template_1_blk.png’ for black point cursor and ‘template_1_white.png’ for normal cursor

Step 4: Resizing the template as per required

Step 5: Applying Gaussian Filter over the image as blur = gaussian_filter(img,0) where 0 is the sigma value. Library used is from ‘scipy.ndimage.filters import gaussian_filter’

Step 6: Calculating Laplacian of Gaussian filtered image and the template as -

lap_i = cv2.Laplacian(blur, cv2.CV_32F)

lap_t = cv2.Laplacian(temp, cv2.CV_32F)

Step 7: Matching template by passing lap_i and lap_t and also the method -

cv2.matchTemplate(lap_i, lap_t, cv2.TM_CCORR_NORMED)

Step 8: Setting up a relevant threshold, threshold varies according to the cursor that is to be detected. For eg. To detect black point cursor, threshold set is 0.44 and to detect hand cursor, threshold set is 0.45

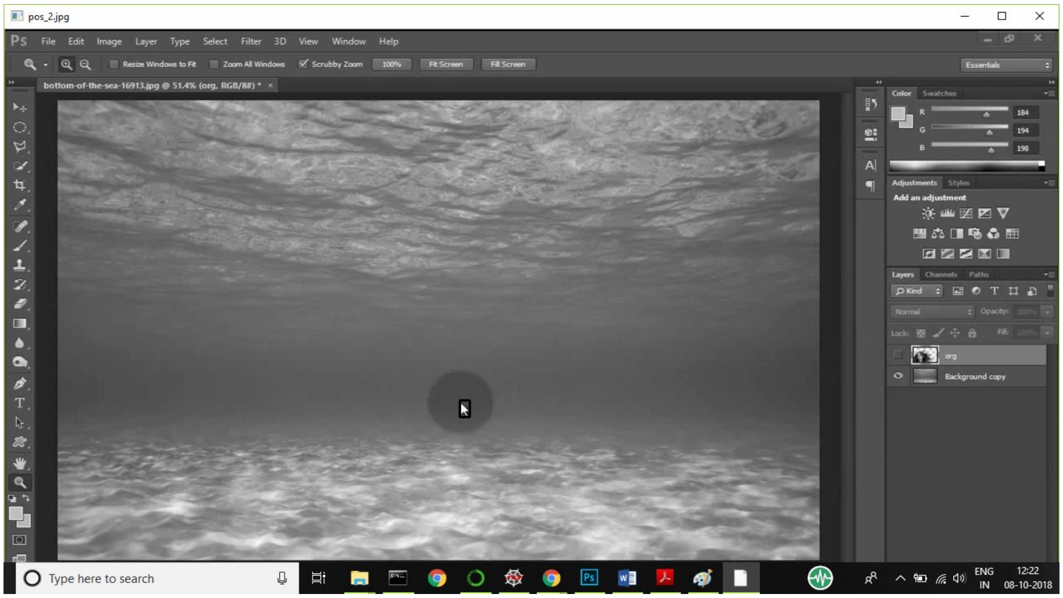
Step 9: Plotting a rectangle where the cursor is detected

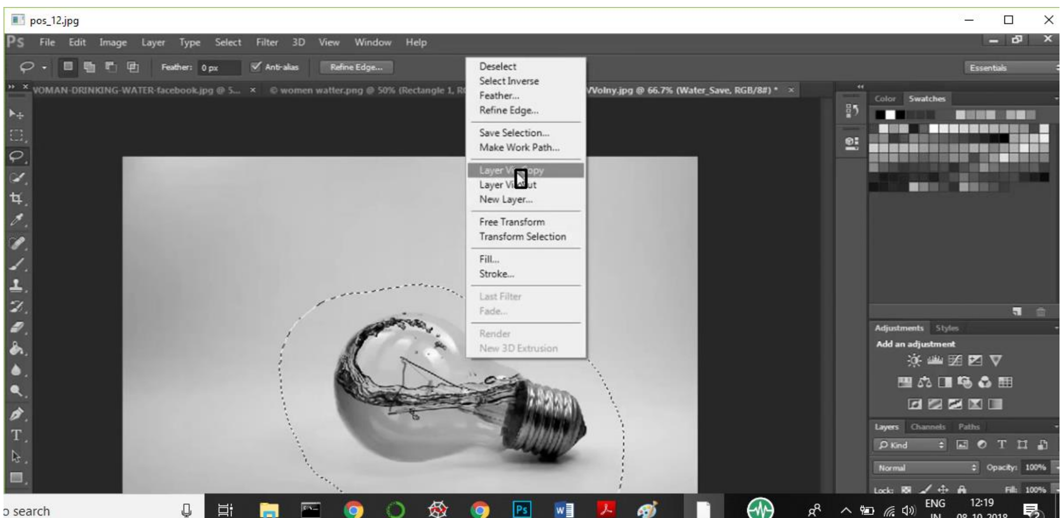
3.2 Detecting Normal White Cursor

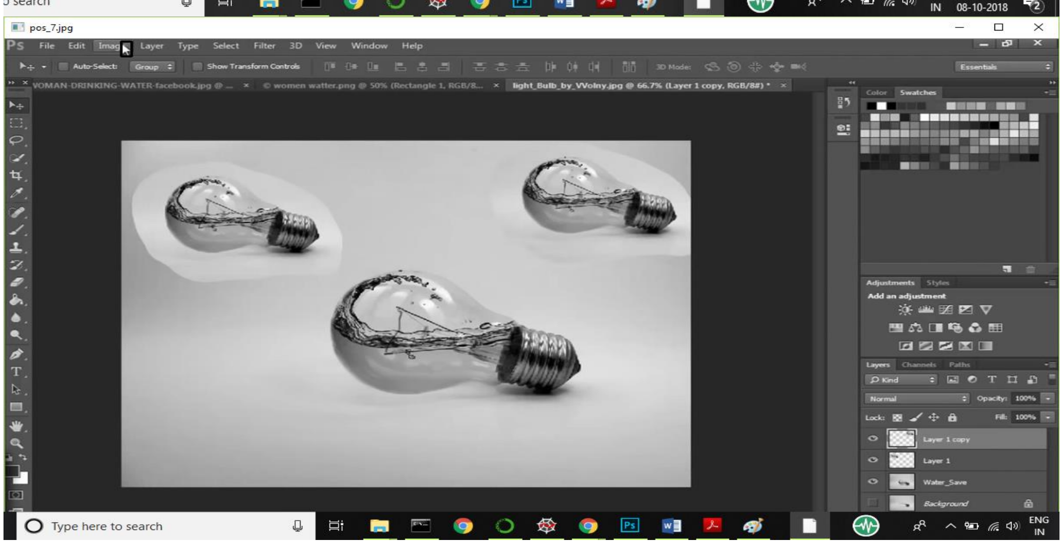
Source Code:

```
#Detecting Normal White Cursor
import cv2
from math import exp
from scipy.ndimage.filters import gaussian_filter
import numpy as np
images = [ 'pos_4.jpg']
for y in images:
    img = cv2.imread(y,0)
    temp = cv2.imread('template_1_white.png',0)
    w, h = temp.shape[::-1]
    blur = gaussian_filter(img,0)
    lap_i = cv2.Laplacian(blur, cv2.CV_32F)
    lap_t = cv2.Laplacian(temp, cv2.CV_32F)
    res = cv2.matchTemplate(lap_i, lap_t, cv2.TM_CCORR_NORMED)
    threshold = 0.5
    loc = np.where( res >= threshold)
    for pt in zip(*loc[::-1]):
        cv2.rectangle(img, pt, (pt[0] + w, pt[1] + h), (0,255,0), 2)
    cv2.imshow(y,img)
    cv2.waitKey(0)
```

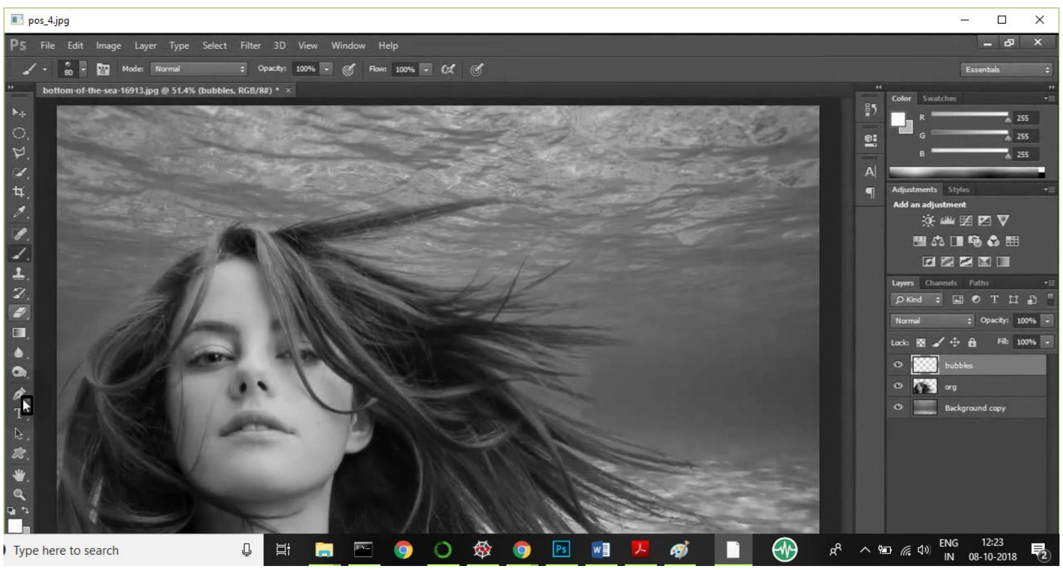
Following are the few sample images where the normal white cursor is detected:



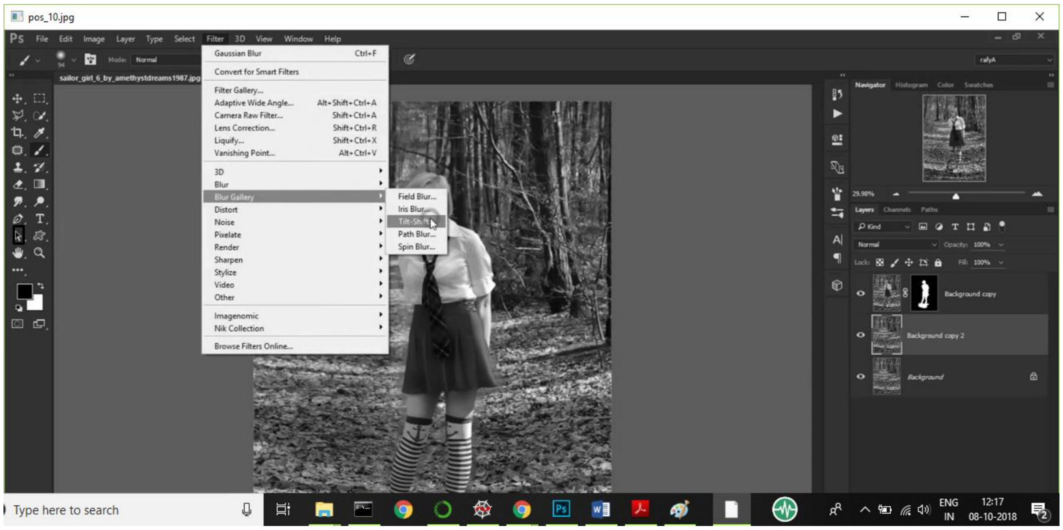


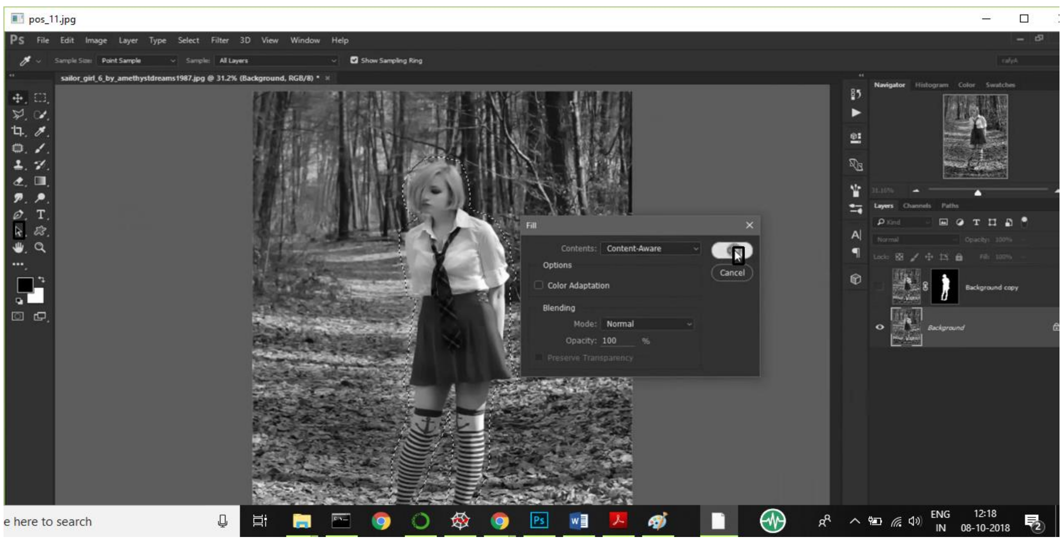


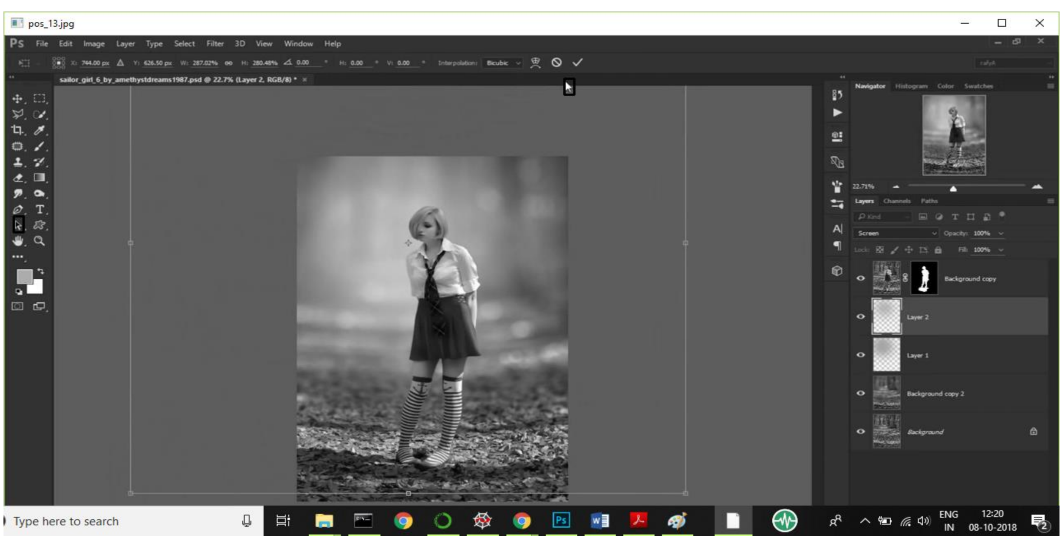

pos_3.jpg
File Edit Image Layer Type Select Filter 3D View Window Help
Mode: Normal
Opacity: 100%
Flow: 100%
Essentials
bottom-of-the-sea-16913.jpg @ 51.4% (bubbles, RGB/8#) *
Color Swatches
Adjustments Styles
Add an adjustment
Layers Channels Paths
Overlay
refection
bubbles
org
Background copy
Type here to search
ENG
12:23


pos_4.jpg
File Edit Image Layer Type Select Filter 3D View Window Help
Mode: Normal
Opacity: 100%
Flow: 100%
Essentials
bottom-of-the-sea-16913.jpg @ 51.4% (bubbles, RGB/8#) *
Color Swatches
Adjustments Styles
Add an adjustment
Layers Channels Paths
Normal
bubbles
org
Background copy
Type here to search
ENG
IN
12:23
08-10-2018

False Positive Match:







3.3 Detecting Hand Cursor

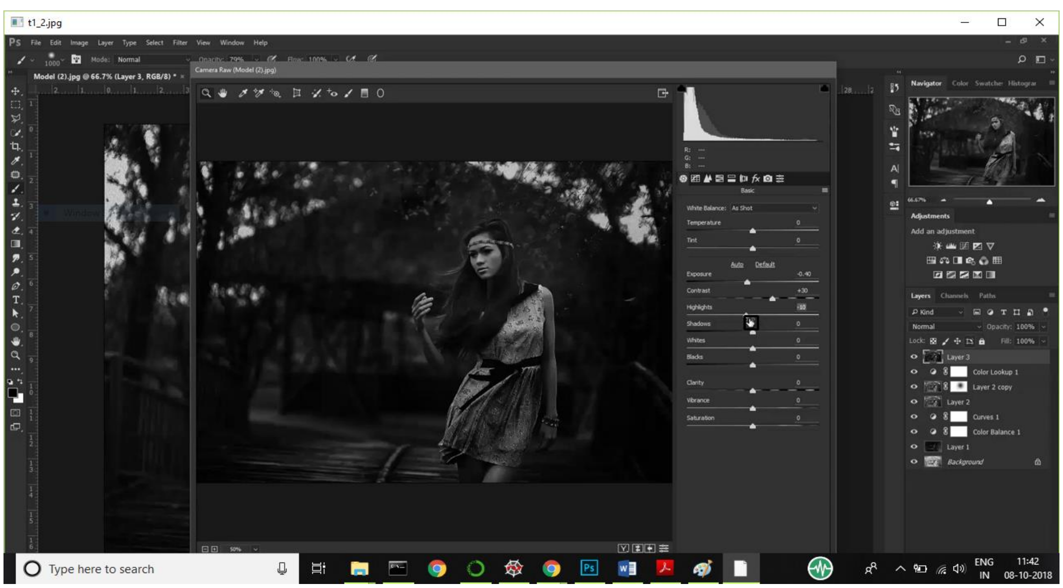
Source Code:

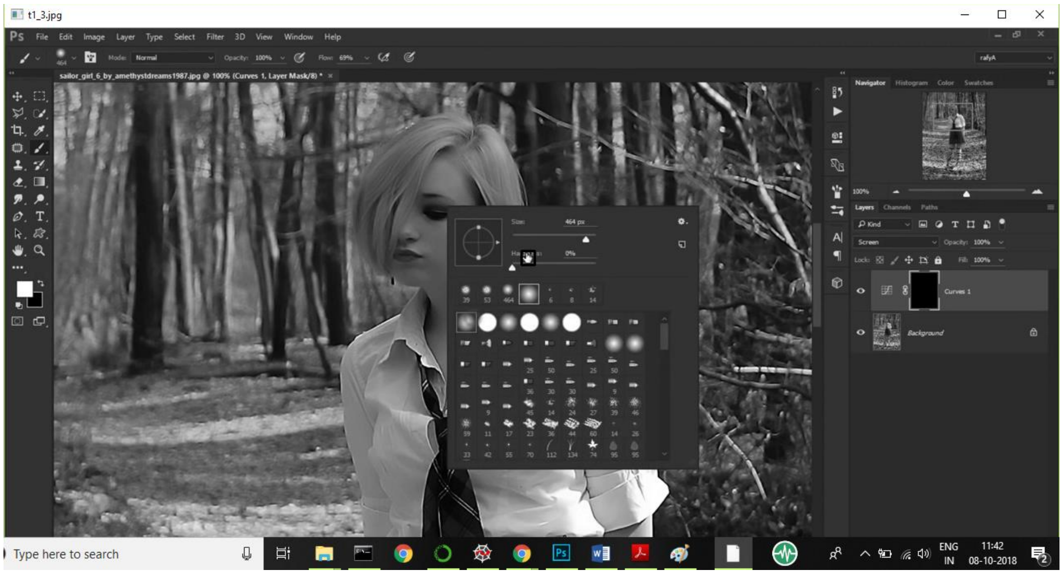
#Detecting hand cursor

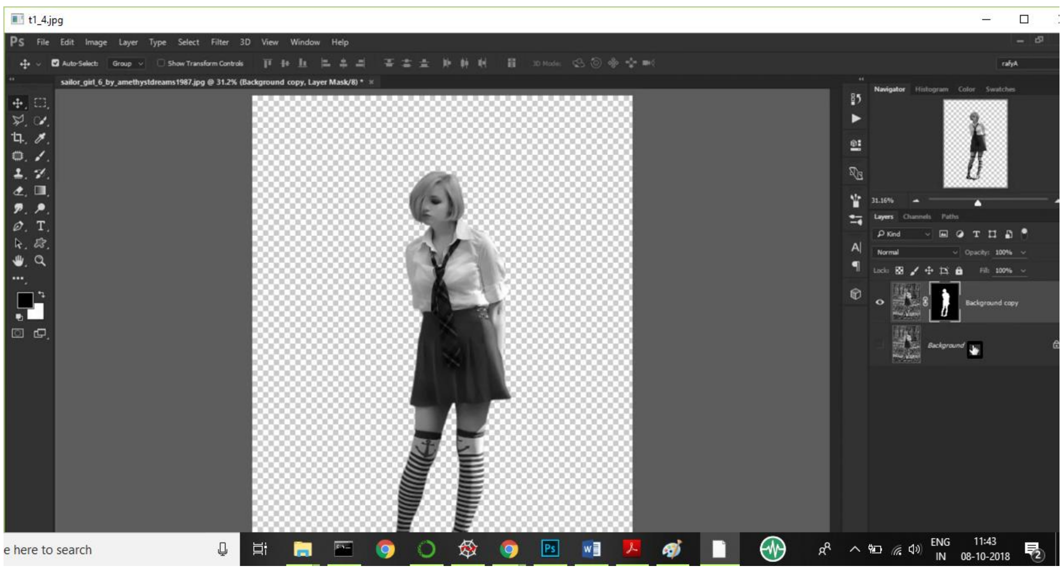
#Successfully detected - 't1_2.jpg','t1_3.jpg','t1_4.jpg' and 't1_6.jpg'

```
import cv2
from math import exp
from scipy.ndimage.filters import gaussian_filter
import numpy as np
images = ['t1_1.jpg','t1_2.jpg','t1_3.jpg','t1_4.jpg','t1_5.jpg','t1_6.jpg']
for y in images:
    img = cv2.imread(y,0)
    temp = cv2.imread('template_1_hand.png',0)
    temp = temp[::4, ::4]
    w, h = temp.shape[::-1]
    blur = gaussian_filter(img,0)
    lap_i = cv2.Laplacian(blur, cv2.CV_32F)
    lap_t = cv2.Laplacian(temp, cv2.CV_32F)
    res = cv2.matchTemplate(lap_i, lap_t, cv2.TM_CCORR_NORMED)
    threshold = 0.45
    loc = np.where( res >= threshold)
    for pt in zip(*loc[::-1]):
        cv2.rectangle(img, pt, (pt[0] + w, pt[1] + h), (0,255,0), 2)
    cv2.imshow(y,img)
    cv2.waitKey(0)
```

Few Sample Images Detected with Hand Cursor:







3.4 Detecting Black Point Cursor

Source Code:

#Detecting Black Point Cursor

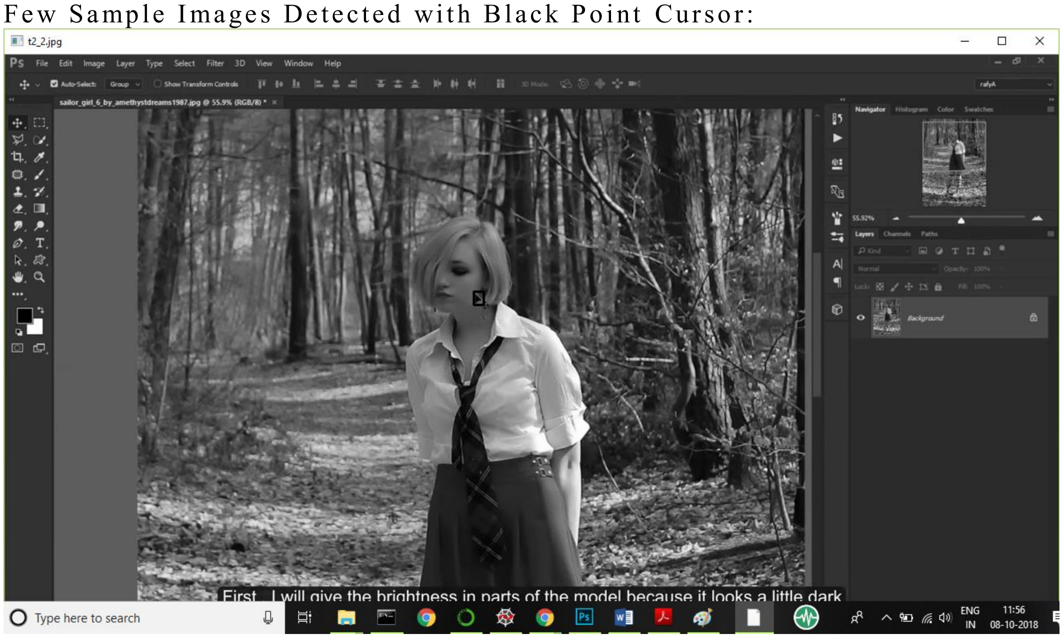
#Successfully Detected: 't2_2.jpg','t2_3.jpg','t2_4.jpg','t2_5.jpg','t2_6.jpg'

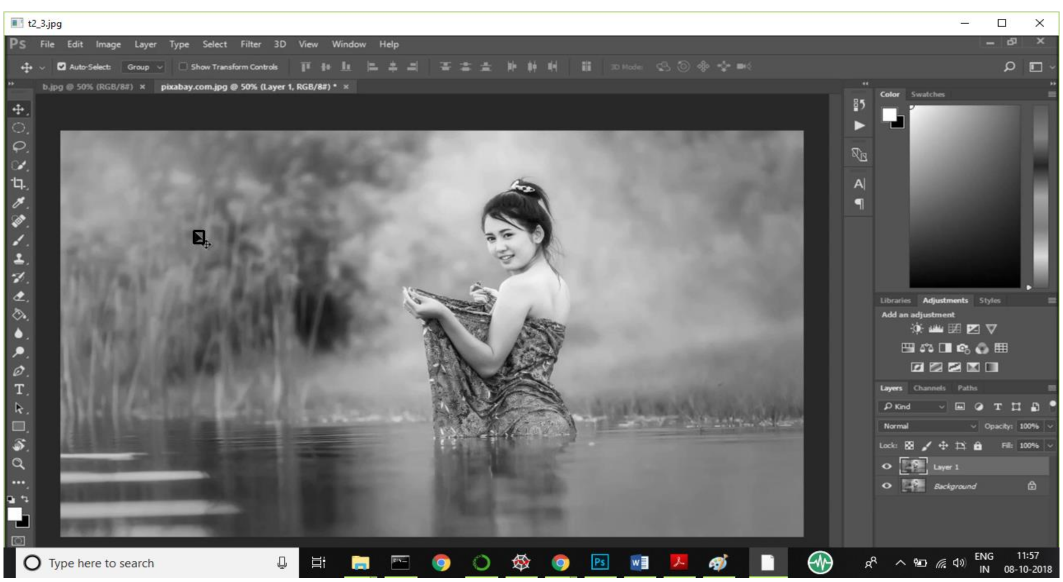
#Detecting False Positive Matches: 't2_6.jpg'

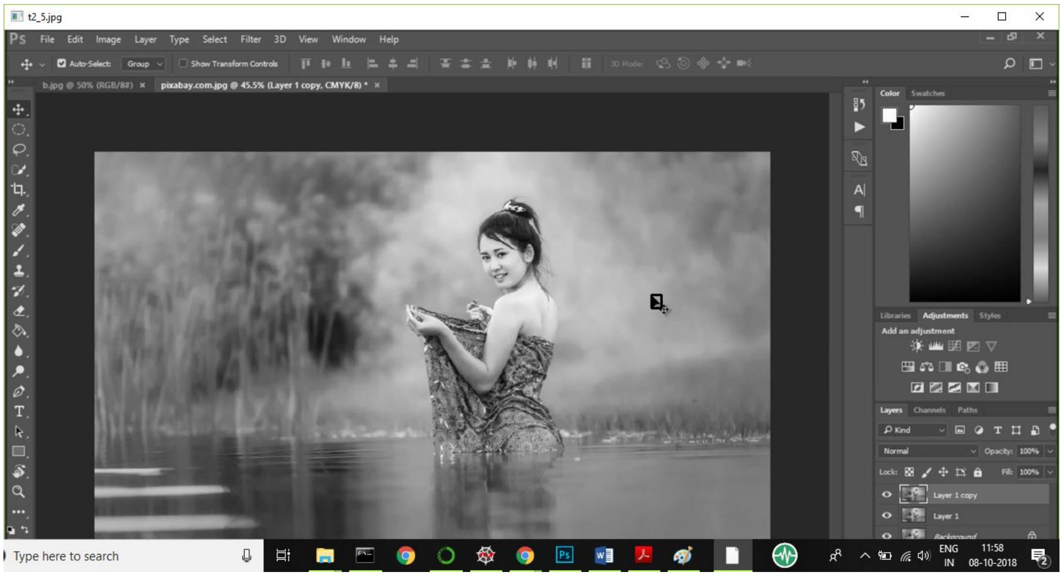
```
import cv2
from math import exp
from scipy.ndimage.filters import gaussian_filter
import numpy as np
images = ['t2_1.jpg','t2_2.jpg','t2_3.jpg','t2_4.jpg','t2_5.jpg','t2_6.jpg']
```

```
for y in images:
   img = cv2.imread(y,0)
   temp = cv2.imread('template_1_blk.png',0)
   w, h = temp.shape[::-1]
   blur = gaussian_filter(img,0)
   lap_i = cv2.Laplacian(blur, cv2.CV_32F)
   lap_t = cv2.Laplacian(temp, cv2.CV_32F)
   res = cv2.matchTemplate(lap_i, lap_t, cv2.TM_CCORR_NORMED)
   threshold = 0.45
   loc = np.where( res >= threshold)
   for pt in zip(*loc[::-1]):
      cv2.rectangle(img, pt, (pt[0] + w, pt[1] + h), (0,255,0), 2)
   cv2.imshow(y,img)
   cv2.waitKey(0)
```

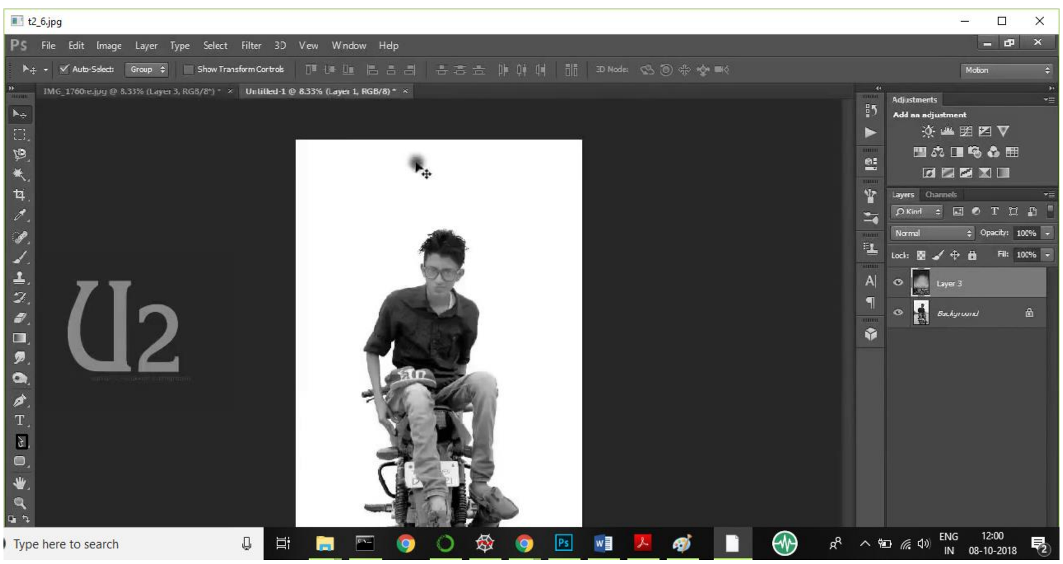
Few Sample Images Detected with Black Point Cursor:







False Positive Match:



4 References

1. http://aishack.in/tutorials/sift-scale-invariant-feature-transform-scale-space/

2. https://en.wikipedia.org/wiki/Sobel_operator

3. https://docs.opencv.org/3.4/d5/db5/tutorial_laplace_operator.html

4.https://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.ndimage.filters.gaussian_filter.html