VARUN NAGARAJ UB Person number : 50290761

COMPUTER VISION PROJECT -1

# TASK 1

Implement a SOBEL Edge detector from scratch.

## Sub tasks

### Implement edge detection using the X-Direction kernel

### mage result for sobel operator exampleImplement edge detection using the Y-Direction kernel

### What is Sobel operator?

### The Sobel operator, sometimes called the Sobel–Feldman operator or Sobel filter, is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasising edges.

## Source code

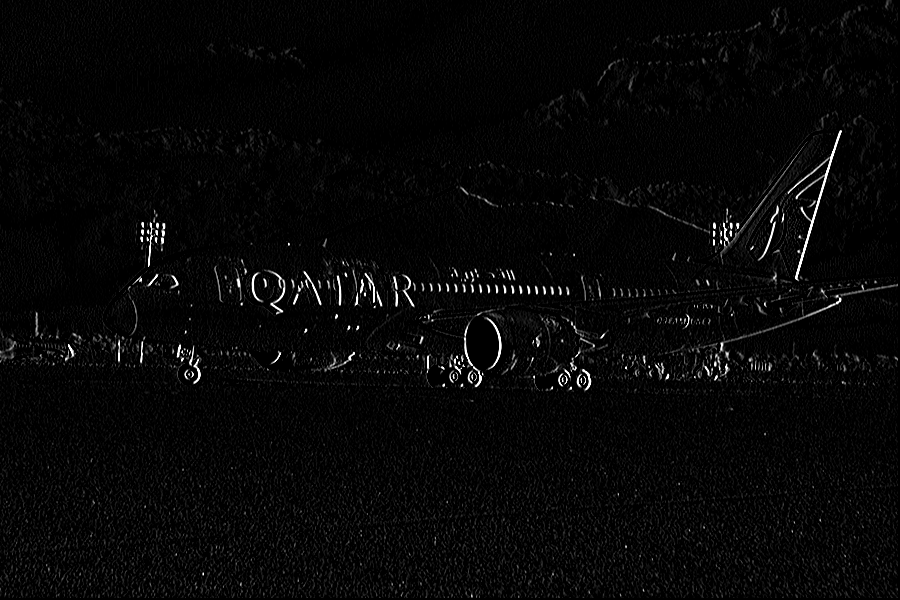


C. Output Images

Horizontal Edge Detection



Vertical Edge Detection



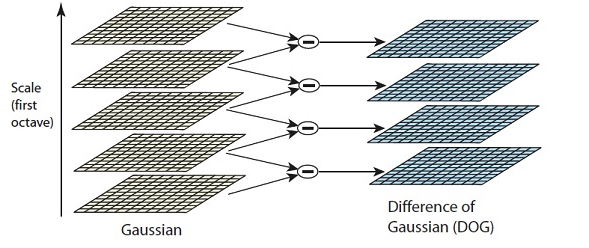
# Task 2

Program to detect keypoints in an image. This task implements the first three steps of Scale Invarient Feature Transform (SIFT)

* Generate four octaves. Each octave is composed of five images blurred using Gaussian kernels.

Here we implement the Gaussian kernel and convolve with the image to produce a blur effect in each stage. These are represented as octaves.

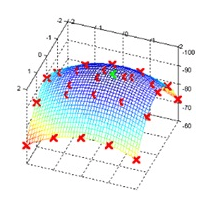
Each of these five octaves are then subjected to Difference of Gaussian (DoG) where we find the differences between the gaussians in sets of 2.



After we find the 4 images of DoGs per octave, we scale down the image to find the next set of DoGs.

We then find the maxima or minima of the DoGs by comparing it with the top DoG and bottom DoG.

Once we get the set of maxima and minima, we plot them back on the original image. I have produced the keypoints on the black image for better clarity.



Similar to the above figure we find the minima’s too from the image.

**2nd Octave:**

**Image dimensions: 344 X 563**



**3rd Octave Images:Dimensions: 258 X 423**

**Difference of Gaussian (DoGs)**

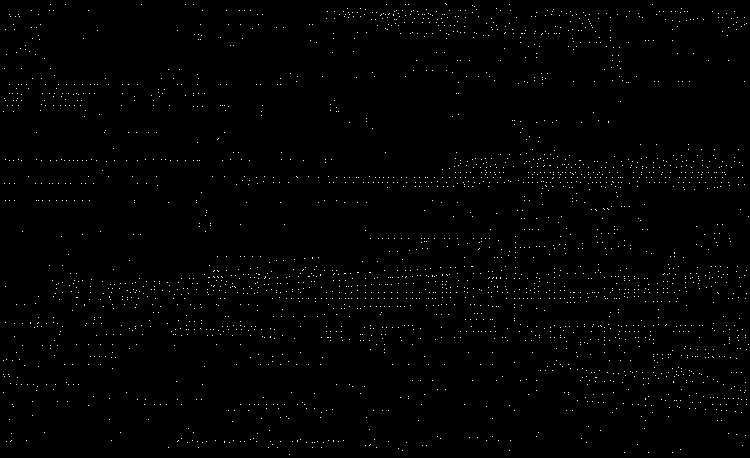
**2nd Octave DoG set:**



**3rd Octave DoG set:**



* **Final Keypoint Image:**

****



Source Code for SIFT:

**from** \_\_future\_\_ **import** division  
**import** cv2  
**import** numpy **as** np  
  
white\_image = cv2.imread(**'white.png'**)  
black\_image = ~white\_image  
black\_image = cv2.cvtColor(black\_image, cv2.COLOR\_BGR2GRAY)  
image\_original = cv2.imread(**'task2.jpg'**)  
image = cv2.imread(**'task2.jpg'**, 0)  
cv2.imshow(**"original"**, image)  
**print**(image.shape)  
  
*# Initializing the 20 Guassian Kernels, dimension = (7x7)*G1 = [([0] \* 7) **for** g **in** range(7)]  
G2 = [([0] \* 7) **for** g **in** range(7)]  
G3 = [([0] \* 7) **for** g **in** range(7)]  
G4 = [([0] \* 7) **for** g **in** range(7)]  
G5 = [([0] \* 7) **for** g **in** range(7)]  
G6 = [([0] \* 7) **for** g **in** range(7)]  
G7 = [([0] \* 7) **for** g **in** range(7)]  
G8 = [([0] \* 7) **for** g **in** range(7)]  
G9 = [([0] \* 7) **for** g **in** range(7)]  
G10 = [([0] \* 7) **for** g **in** range(7)]  
G11 = [([0] \* 7) **for** g **in** range(7)]  
G12 = [([0] \* 7) **for** g **in** range(7)]  
G13 = [([0] \* 7) **for** g **in** range(7)]  
G14 = [([0] \* 7) **for** g **in** range(7)]  
G15 = [([0] \* 7) **for** g **in** range(7)]  
G16 = [([0] \* 7) **for** g **in** range(7)]  
G17 = [([0] \* 7) **for** g **in** range(7)]  
G18 = [([0] \* 7) **for** g **in** range(7)]  
G19 = [([0] \* 7) **for** g **in** range(7)]  
G20 = [([0] \* 7) **for** g **in** range(7)]  
  
gaussian\_kernel = [[G1, G2, G3, G4, G5], [G6, G7, G8, G9, G10], [G11, G12, G13, G14, G15], [G16, G17, G18, G19, G20]]  
maxima = {}  
  
  
**def** show\_images(image\_arrays, count):  
 **for** i **in** range(0,5):  
 cv2.imwrite(**"Octave\_{}{}.jpg"**.format(count,i),np.asarray(image\_arrays[i]))  
  
  
**def** scale\_down\_image(image, orig\_height, orig\_width, scaled\_width, scaled\_height):  
  
 *# Using Nearest Neighbor Algorithm to scale the image* orig\_width = int(orig\_width)  
 orig\_height = int(orig\_height)  
 scaled\_height = int(scaled\_height)  
 scaled\_width = int(scaled\_width)  
 scaled\_image = [([0] \* scaled\_width) **for** s **in** range(scaled\_height)]  
 **for** h **in** range(0, scaled\_height):  
 **for** w **in** range(0, scaled\_width):  
 n\_w = int(round(float(w) / float(scaled\_width) \* float(orig\_width)))  
 n\_h = int(round(float(h) / float(scaled\_height) \* float(orig\_height)))  
 index\_w = min(n\_w, orig\_width - 1)  
 index\_h = min(n\_h, orig\_height - 1)  
 scaled\_image[h][w] = image[index\_h][index\_w]  
  
 **return** scaled\_image  
  
  
**def** kernel\_generation(rows, columns, image, gaussian\_kernel\_list):  
 rows = int(rows)  
 columns = int(columns)  
 h1 = [([0] \* columns) **for** h **in** range(rows)]  
 h2 = [([0] \* columns) **for** h **in** range(rows)]  
 h3 = [([0] \* columns) **for** h **in** range(rows)]  
 h4 = [([0] \* columns) **for** h **in** range(rows)]  
 h5 = [([0] \* columns) **for** h **in** range(rows)]  
 **for** i **in** range(3, rows - 3):  
 **for** j **in** range(3, columns - 3):  
 **for** u **in** range(7):  
 **for** v **in** range(7):  
 h1[i][j] += float(gaussian\_kernel\_list[0][u][v] \* image[i + u - 3][j + v - 3])  
 h2[i][j] += float(gaussian\_kernel\_list[1][u][v] \* image[i + u - 3][j + v - 3])  
 h3[i][j] += float(gaussian\_kernel\_list[2][u][v] \* image[i + u - 3][j + v - 3])  
 h4[i][j] += float(gaussian\_kernel\_list[3][u][v] \* image[i + u - 3][j + v - 3])  
 h5[i][j] += float(gaussian\_kernel\_list[4][u][v] \* image[i + u - 3][j + v - 3])  
  
 **for** i **in** range(3, rows - 3):  
 **for** j **in** range(3, columns - 3):  
 h1[i][j] = (h1[i][j] / find\_max(h1[i]))\*255.0  
 h2[i][j] = (h2[i][j] / find\_max(h2[i]))\*255.0  
 h3[i][j] = (h3[i][j] / find\_max(h3[i]))\*255.0  
 h4[i][j] = (h4[i][j] / find\_max(h4[i]))\*255.0  
 h5[i][j] = (h5[i][j] / find\_max(h5[i]))\*255.0  
 **return** [h1, h2, h3, h4, h5]  
  
**def** find\_max(arr):  
 max\_of\_items = 0  
 **for** item **in** arr:  
 **if** item > max\_of\_items:  
 max\_of\_items = item  
 **if** max\_of\_items:  
 **return** max\_of\_items  
 **else**:  
 **return** 1  
  
  
**def** gaussian\_filter(sigma, gaussian\_kernel\_list):  
 c\_value = [0 **for** g **in** range(5)]  
 **for** z **in** range(-3, 4):  
 **for** k **in** range(-3, 4):  
 gaussian\_kernel\_list[0][z + 3][k + 3] = np.math.exp(-1 \* ((z \*\* 2 + k \*\* 2) / (2 \* (sigma[0] \*\* 2)))) / (  
 2 \* np.math.pi \* (sigma[0] \*\* 2))  
 c\_value[0] += gaussian\_kernel\_list[0][z+3][k+3]  
 gaussian\_kernel\_list[1][z + 3][k + 3] = np.math.exp(-1 \* ((z \*\* 2 + k \*\* 2) / (2 \* (sigma[1] \*\* 2)))) / (  
 2 \* np.math.pi \* (sigma[1] \*\* 2))  
 c\_value[1] += gaussian\_kernel\_list[1][z + 3][k + 3]  
 gaussian\_kernel\_list[2][z + 3][k + 3] = np.math.exp(-1 \* ((z \*\* 2 + k \*\* 2) / (2 \* (sigma[2] \*\* 2)))) / (  
 2 \* np.math.pi \* (sigma[2] \*\* 2))  
 c\_value[2] += gaussian\_kernel\_list[2][z + 3][k + 3]  
 gaussian\_kernel\_list[3][z + 3][k + 3] = np.math.exp(-1 \* ((z \*\* 2 + k \*\* 2) / (2 \* (sigma[3] \*\* 2)))) / (  
 2 \* np.math.pi \* (sigma[3] \*\* 2))  
 c\_value[3] += gaussian\_kernel\_list[3][z + 3][k + 3]  
 gaussian\_kernel\_list[4][z + 3][k + 3] = np.math.exp(-1 \* ((z \*\* 2 + k \*\* 2) / (2 \* (sigma[4] \*\* 2)))) / (  
 2 \* np.math.pi \* (sigma[4] \*\* 2))  
 c\_value[4] += gaussian\_kernel\_list[4][z + 3][k + 3]  
 **return** gaussian\_kernel\_list, c\_value  
  
  
**def** diff\_of\_gaussian(im\_oct\_1, im\_oct\_2, row, col):  
 dog = [([0]\*int(col)) **for** d **in** range(int(row))]  
 **for** i **in** range(0, int(row)):  
 **for** j **in** range(0, int(col)):  
 dog[i][j] = im\_oct\_1[i][j] - im\_oct\_2[i][j]  
 **return** np.asarray(dog)  
  
  
**def** find\_keypoints(main\_dog, top\_dog, bottom\_dog, rows, columns, count):  
 rows = int(rows)  
 columns = int(columns)  
 maxima\_list = []  
 **for** row **in** range(1, rows - 1):  
 **for** col **in** range(1, columns - 1):  
 **if** (main\_dog[row][col] > main\_dog[row - 1][col - 1]) **and** (main\_dog[row][col] > main\_dog[row - 1][col]) **and** (  
 main\_dog[row][col] > main\_dog[row - 1][col + 1]) **and** (  
 main\_dog[row][col] > main\_dog[row][col - 1]) **and** (main\_dog[row][col] > main\_dog[row][col + 1]) **and** (  
 main\_dog[row][col] > main\_dog[row + 1][col - 1]) **and** (  
 main\_dog[row][col] > main\_dog[row + 1][col]) **and** (main\_dog[row][col] > main\_dog[row + 1][col + 1]):  
 **if** (main\_dog[row][col] > top\_dog[row - 1][col - 1]) **and** (  
 main\_dog[row][col] > top\_dog[row - 1][col]) **and** (  
 main\_dog[row][col] > top\_dog[row - 1][col + 1]) **and** (  
 main\_dog[row][col] > top\_dog[row][col - 1]) **and** (main\_dog[row][col] > top\_dog[row][col]) **and** (  
 main\_dog[row][col] > top\_dog[row][col + 1]) **and** (  
 main\_dog[row][col] > top\_dog[row + 1][col - 1]) **and** (  
 main\_dog[row][col] > top\_dog[row + 1][col]) **and** (  
 main\_dog[row][col] > top\_dog[row + 1][col + 1]):  
 **if** (main\_dog[row][col] > bottom\_dog[row - 1][col - 1]) **and** (  
 main\_dog[row][col] > bottom\_dog[row - 1][col]) **and** (  
 main\_dog[row][col] > bottom\_dog[row - 1][col + 1]) **and** (  
 main\_dog[row][col] > bottom\_dog[row][col - 1]) **and** (  
 main\_dog[row][col] > bottom\_dog[row][col]) **and** (  
 main\_dog[row][col] > bottom\_dog[row][col + 1]) **and** (  
 main\_dog[row][col] > bottom\_dog[row + 1][col - 1]) **and** (  
 main\_dog[row][col] > bottom\_dog[row + 1][col]) **and** (  
 main\_dog[row][col] > bottom\_dog[row + 1][col + 1]):  
 **if** count != 0:  
 maxima\_list.append((int(row\*count/0.75),int(col\*count/0.75)))  
 **else**:  
 maxima\_list.append((row, col))  
   
 **elif** (main\_dog[row][col] < main\_dog[row - 1][col - 1]) **and** (main\_dog[row][col] < main\_dog[row - 1][col]) **and** (  
 main\_dog[row][col] < main\_dog[row - 1][col + 1]) **and** (  
 main\_dog[row][col] < main\_dog[row][col - 1]) **and** (main\_dog[row][col] < main\_dog[row][col + 1]) **and** (  
 main\_dog[row][col] < main\_dog[row + 1][col - 1]) **and** (  
 main\_dog[row][col] < main\_dog[row + 1][col]) **and** (main\_dog[row][col] < main\_dog[row + 1][col + 1]):  
 **if** (main\_dog[row][col] < top\_dog[row - 1][col - 1]) **and** (  
 main\_dog[row][col] < top\_dog[row - 1][col]) **and** (  
 main\_dog[row][col] < top\_dog[row - 1][col + 1]) **and** (  
 main\_dog[row][col] < top\_dog[row][col - 1]) **and** (main\_dog[row][col] < top\_dog[row][col]) **and** (  
 main\_dog[row][col] < top\_dog[row][col + 1]) **and** (  
 main\_dog[row][col] < top\_dog[row + 1][col - 1]) **and** (  
 main\_dog[row][col] < top\_dog[row + 1][col]) **and** (  
 main\_dog[row][col] < top\_dog[row + 1][col + 1]):  
 **if** (main\_dog[row][col] < bottom\_dog[row - 1][col - 1]) **and** (  
 main\_dog[row][col] < bottom\_dog[row - 1][col]) **and** (  
 main\_dog[row][col] < bottom\_dog[row - 1][col + 1]) **and** (  
 main\_dog[row][col] < bottom\_dog[row][col - 1]) **and** (  
 main\_dog[row][col] < bottom\_dog[row][col]) **and** (  
 main\_dog[row][col] < bottom\_dog[row][col + 1]) **and** (  
 main\_dog[row][col] < bottom\_dog[row + 1][col - 1]) **and** (  
 main\_dog[row][col] < bottom\_dog[row + 1][col]) **and** (  
 main\_dog[row][col] < bottom\_dog[row + 1][col + 1]):  
 **if** count != 0:  
 maxima\_list.append((int((row/0.75)\*count),int((col/0.75)\*count)))  
 **else**:  
 maxima\_list.append((row, col))  
 maxima.update({count: maxima\_list})  
  
  
**def** normalize\_kernel(kernel\_array, c\_value, count):  
 **for** row **in** range(5):  
 **for** col **in** range(5):  
 kernel\_array[row][col] /= c\_value  
 **return** kernel\_array  
  
  
octaves = np.asarray([[1.0 / np.sqrt(2), 1.0, np.sqrt(2.0), 2.0, 2.0 \* np.sqrt(2)],  
 [np.sqrt(2.0), 2.0, 2.0 \* np.sqrt(2), 4.0, 4.0 \* np.sqrt(2)],  
 [2.0 \* np.sqrt(2), 4.0, 4.0 \* np.sqrt(2), 8.0, 8.0 \* np.sqrt(2)],  
 [4.0 \* np.sqrt(2), 8.0, 8.0 \* np.sqrt(2), 16.0, 16.0 \* np.sqrt(2)]], dtype=np.float)  
count = 0  
**for** sigma\_values **in** octaves:  
 gaussian\_kernel[count], c\_value = gaussian\_filter(sigma\_values, gaussian\_kernel[count])  
 **for** values **in** c\_value:  
 gaussian\_kernel[count] = normalize\_kernel(gaussian\_kernel[count], values, c\_value.index(values))  
 count += 1  
  
filtered\_images = {}  
count = 0  
  
*# Reduce the image to 0.75 of the previous output*filtered\_images[count] = kernel\_generation(458, 750,image, gaussian\_kernel[count])  
image\_out = scale\_down\_image(image, 458, 750, 563, 344)  
show\_images(filtered\_images[count], count)  
count += 1  
  
*# Reduce the image to 0.75 of the previous output*filtered\_images[count] = kernel\_generation(344, 563, image\_out, gaussian\_kernel[count])  
image\_out\_1 = scale\_down\_image(image\_out, 344, 563, 423, 258)  
show\_images(filtered\_images[count], count)  
count += 1  
  
*# Reduce the image to 0.75 of the previous output*filtered\_images[count] = kernel\_generation(258, 423, image\_out\_1, gaussian\_kernel[count])  
image\_out\_2 = scale\_down\_image(image\_out\_1, 258, 423, 318, 194)  
show\_images(filtered\_images[count], count)  
count += 1  
  
*# Reduce the image to 0.75 of the previous output*filtered\_images[count] = kernel\_generation(194, 318, image\_out\_2, gaussian\_kernel[count])  
image\_out\_3 = scale\_down\_image(image\_out\_1, 194, 318, 239, 146)  
show\_images(filtered\_images[count], count)  
count += 1  
  
  
**def** normalize(arr, row, col):  
 max\_value = 0  
 **for** i **in** range(row):  
 **for** j **in** range(col):  
 **if** max\_value < arr[i][j]:  
 max\_value = arr[i][j]  
 **for** i **in** range(row):  
 **for** j **in** range(col):  
 arr[i][j] = (arr[i][j] / max\_value)\*255  
 **return** arr  
  
*# Calculate the DoG's between images*L\_11 = diff\_of\_gaussian(filtered\_images[0][0], filtered\_images[0][1], 458, 750)  
L\_12 = diff\_of\_gaussian(filtered\_images[0][1], filtered\_images[0][2], 458, 750)  
L\_13 = diff\_of\_gaussian(filtered\_images[0][2], filtered\_images[0][3], 458, 750)  
L\_14 = diff\_of\_gaussian(filtered\_images[0][3], filtered\_images[0][4], 458, 750)  
normalize(L\_11, 458,750)  
normalize(L\_12, 458, 750)  
normalize(L\_13,458, 750)  
normalize(L\_14, 458, 750)  
cv2.imwrite(**"dog1.jpg"**,np.asarray(L\_11))  
cv2.imwrite(**"dog2.jpg"**,np.asarray(L\_12))  
cv2.imwrite(**"dog3.jpg"**,np.asarray(L\_13))  
cv2.imwrite(**"dog4.jpg"**,np.asarray(L\_14))  
  
L\_21 = diff\_of\_gaussian(filtered\_images[1][0], filtered\_images[1][1], 344,563)  
L\_22 = diff\_of\_gaussian(filtered\_images[1][1], filtered\_images[1][2], 344,563)  
L\_23 = diff\_of\_gaussian(filtered\_images[1][2], filtered\_images[1][3], 344,563)  
L\_24 = diff\_of\_gaussian(filtered\_images[1][3], filtered\_images[1][4], 344,563)  
normalize(L\_21, 344,563)  
normalize(L\_22, 344, 563)  
normalize(L\_23,344, 563)  
normalize(L\_24, 344, 563)  
cv2.imwrite(**"dog5.jpg"**,np.asarray(L\_21))  
cv2.imwrite(**"dog6.jpg"**,np.asarray(L\_22))  
cv2.imwrite(**"dog7.jpg"**,np.asarray(L\_23))  
cv2.imwrite(**"dog8.jpg"**,np.asarray(L\_24))  
  
L\_31 = diff\_of\_gaussian(filtered\_images[2][0], filtered\_images[2][1], 258, 423)  
L\_32 = diff\_of\_gaussian(filtered\_images[2][1], filtered\_images[2][2], 258, 423)  
L\_33 = diff\_of\_gaussian(filtered\_images[2][2], filtered\_images[2][3], 258, 423)  
L\_34 = diff\_of\_gaussian(filtered\_images[2][3], filtered\_images[2][4], 258, 423)  
normalize(L\_31, 258,423)  
normalize(L\_32, 258, 423)  
normalize(L\_33,258, 423)  
normalize(L\_34, 258, 423)  
cv2.imwrite(**"dog9.jpg"**,np.asarray(L\_31))  
cv2.imwrite(**"dog10.jpg"**,np.asarray(L\_32))  
cv2.imwrite(**"dog11.jpg"**,np.asarray(L\_33))  
cv2.imwrite(**"dog12.jpg"**,np.asarray(L\_34))  
  
L\_41 = diff\_of\_gaussian(filtered\_images[3][0], filtered\_images[3][1], 194, 318)  
L\_42 = diff\_of\_gaussian(filtered\_images[3][1], filtered\_images[3][2], 194, 318)  
L\_43 = diff\_of\_gaussian(filtered\_images[3][2], filtered\_images[3][3], 194, 318)  
L\_44 = diff\_of\_gaussian(filtered\_images[3][3], filtered\_images[3][4], 194, 318)  
normalize(L\_41, 194,318)  
normalize(L\_42, 194, 318)  
normalize(L\_43,194, 318)  
normalize(L\_44, 194, 318)  
cv2.imwrite(**"dog13.jpg"**,np.asarray(L\_41))  
cv2.imwrite(**"dog14.jpg"**,np.asarray(L\_42))  
cv2.imwrite(**"dog15.jpg"**,np.asarray(L\_43))  
cv2.imwrite(**"dog16.jpg"**,np.asarray(L\_44))  
  
*# Find Keypoints of the image*find\_keypoints(L\_12,L\_11,L\_13, 458, 750, 0)  
find\_keypoints(L\_13,L\_12,L\_14, 458, 750, 0)  
  
find\_keypoints(L\_22,L\_21,L\_23, 344, 563, 1)  
find\_keypoints(L\_23,L\_22,L\_24, 344, 563, 1)  
  
find\_keypoints(L\_32,L\_31,L\_33, 258, 423, 2)  
find\_keypoints(L\_33,L\_32,L\_34, 258, 423, 2)  
  
find\_keypoints(L\_42,L\_41,L\_43, 194, 318, 3)  
find\_keypoints(L\_43,L\_42,L\_44, 194, 318, 3)  
  
counter = 0  
**for** count,loc **in** maxima.items():  
 counter += len(loc)  
 **for** point **in** loc:  
 **try**:  
 image\_original[point[0]][point[1]] = 255  
 black\_image[point[0]][point[1]] = 255  
 **except** IndexError:  
 **continue**cv2.imwrite(**"keypoints.png"**,image\_original)  
cv2.imwrite(**"blacked.png"**,black\_image)  
*# cv2.waitKey()***print**(**"Number of Keypoints: {}"**.format(counter))  
count = 0  
first\_five\_points = []  
**for** i **in** range(black\_image.shape[0]):  
 **for** j **in** range(6):  
 **if** black\_image[i][j] == 255 **and** len(first\_five\_points)<5:  
 first\_five\_points.append((i, j))  
 count += 1  
 **if** count >= 5:  
 **break  
print**(**"The first five keypoints from the left are : {}"**.format(first\_five\_points))

# Task 3

## Cursor detection in images.

## In this task I made use of matchTemplate method using the Normalized cross correlation function.

## This task although allowed the usage of library functions, was experimental and I got to learn about the morphological operations and smoothening, sharpening etc. kernels.

## Methods I used for matching template with image:

#### 1. Method 1

#### Converted the image and template to grayscale images.

#### Applied binary threshold combined with OTSU thresholding to get rid of unwanted noise in the background of the image.

#### Used match template on the threshold image and template.

#### 2. Method 2

#### Converted the image and template to grayscale images.

#### Applied Gaussian Blur on the source.

#### Used Canny edge detector to get the edges of the cursor both in the image and the template.

#### Then used the matchTemplate method using the output of canny.

#### 3. Method 3

#### Converted the image and template into grayscale images.

#### Used scaling operations on the template to make it run from a small size to the actual size based on the scale factor.

#### Compared each of the scaled template with the image and looked for the maximum result value from match template and plotted it on the image.

#### 4. Method 4

#### Converted the image and template to grayscale images

#### Applied Gaussian Blur on the image

#### Applied Laplacian operator on the image and the template

#### Used matchTemplate method with the NCC function to find the cursors in the image.

#### The Method 4 discussed above gave best results out of the 4 methods that I tried. I was able to get max True positive cursor detection in this method. While there were a few false positives, I could understand that template matching is not that accurate and we have to move towards feature based matching.

#### Below is the sample output of the cursor detection.

#### ../../../Desktop/Screenshot%202018-10-08%20at%2012.52.48%20AM

#### Source Code:

#### 