CSE 573 - Computer Vision and Image processing - Project 1 Report

Due date: 8 Oct 2018

Name: *Saketh Varma Pericherla*

UBIT number: *50288206*

1. Edge Detection:

Description:

The steps to perform Edge Detection are:

- Sobel operators - one for horizontal changes and the other for vertical changes - are chosen as kernels for horizontal and vertical edge detection.
- The image is convolved with the two sobel operators.
- The resultant images are normalized to show the detected horizontal and vertical edges.
- Comments in the below code elaborate more on this.

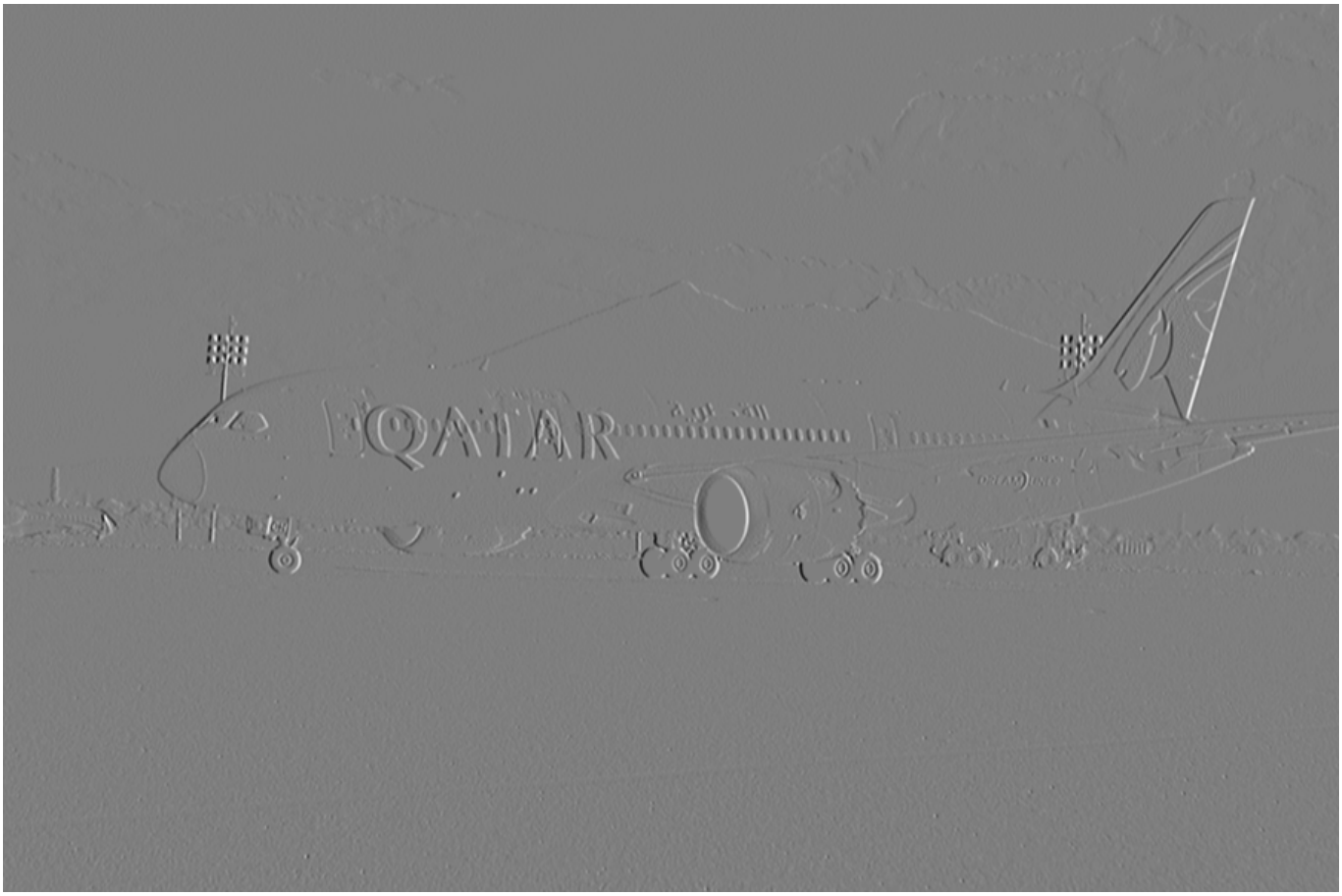
Python 3 Code:

```
import cv2
# only for np.asarray() function
import numpy as np

# used by conv function to do element wise multiplication
def elmwise_mul(a,b):
    c = []
    for i in range(0,len(a)):
        temp=[]
        for j in range(0,len(a[0])):
            temp.append(a[i][j] * b[i][j])
        c.append(temp)
    return c

# used to flatten multi dimensional array to single dimension to find sum, max,
min etc.
def flatten_arr(arr):
    result = []
    for i in range(len(arr)):
        for j in range(len(arr[0])):
            result.append(arr[i][j])
    return result

# convolve image with a kernel of size kernel_size, stride of 1
def conv(img, kernel, kernel_size):
    result = []
```

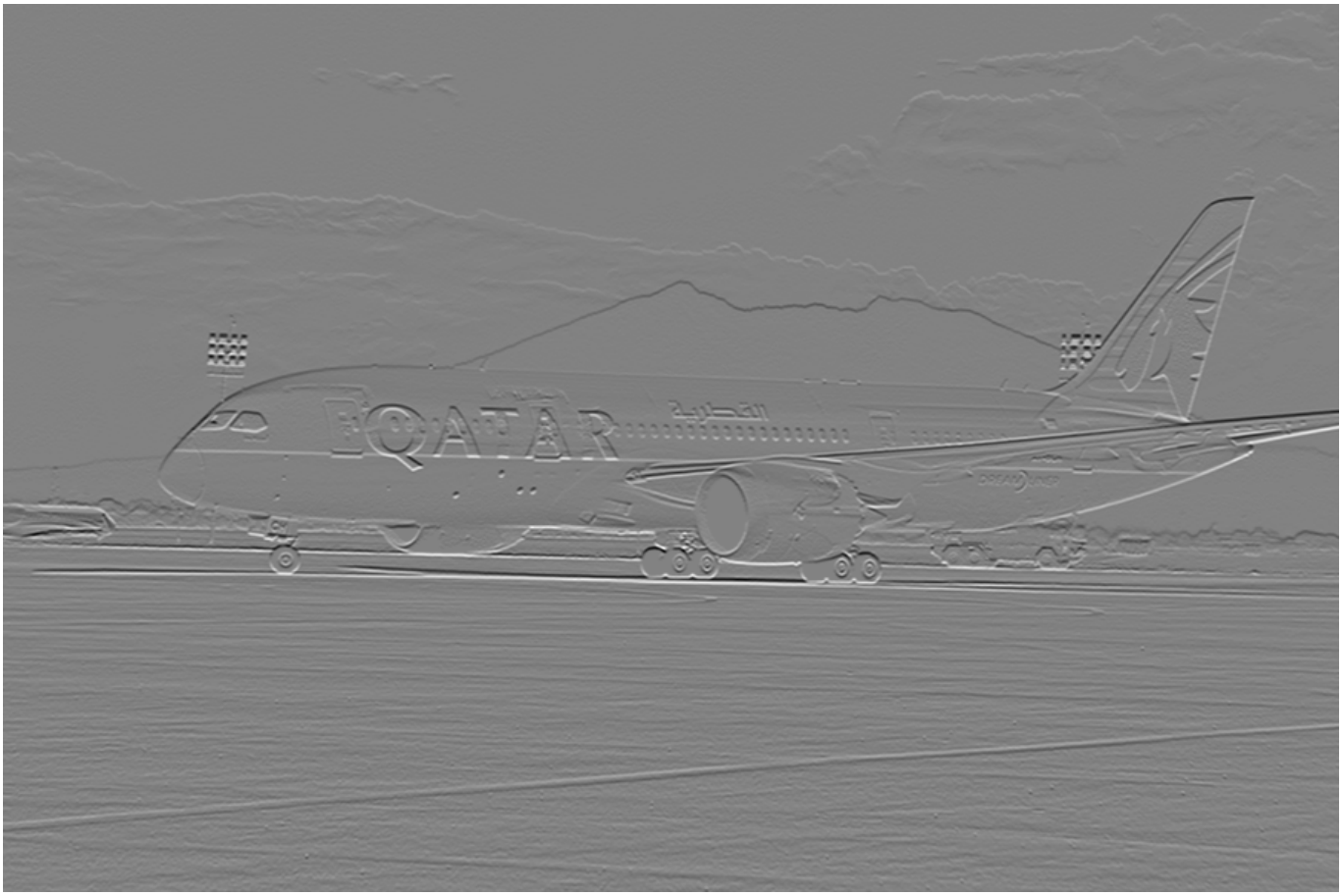
```
    # flips the kernel for convolution
    kernel = kernel[::-1][::-1]
    for i in range(img.shape[0]):
        if i <= img.shape[0] - kernel_size:
            row = []
            for j in range(img.shape[1]):
                if j <= img.shape[1] - kernel_size:
                    # element wise multiplication with kernel and calculates sum
and appends to the result
                    img_arr = np.asarray(img[i:i+kernel_size, j:j+kernel_size])
                    temp = elmwise_mul(kernel, img_arr)
                    temp = sum(flatten_arr(temp))
                    row.append(temp)
            result.append(row)
    return result

def normalize_and_display(result, window_name):
    # flatten the array to calculate maximum and minimum
    arr = flatten_arr(result)
    min_val = min(arr)
    max_val = max(arr)

    # normalize the resultant image before displaying it
    result = (result - min_val) / (max_val - min_val)

    cv2.namedWindow(window_name, cv2.WINDOW_NORMAL)
    cv2.imshow(window_name, result)
    cv2.waitKey(0)

if __name__ == "__main__":
    img = cv2.imread("./task1.png", 0)

    # kernel used to detect horizontal edges
    sobel_x = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]

    # kernel used to detect vertical edges
    sobel_y = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]

    # convolves the kernel with image to detect horizontal edges
    horiz_result = conv(img, sobel_x, 3)

    # normalizes image, displays it and saves image to disk as ./Horiz_conv.png
    normalize_and_display(horiz_result, "Horiz_conv")

    # convolves the kernel with image to detect vertical edges
    vert_result = conv(img, sobel_y, 3)

    # normalizes image, displays it and saves image to disk as ./Vert_conv.png
    normalize_and_display(vert_result, "Vert_conv")
```

Image Results:

Horizontal Edge Detection:



Vertical Edge Detection:



2. Keypoint Detection:

Description

The steps to perform keypoint detection are:

- Generate four octaves of decreasing scales (by half) from the original image.
- Apply guassian blurring for different values of sigma to generate 5 guassian blurred images for each of the four octaves
- Find the difference between the guassians to generate four Differences of Guassians for each of the four octaves
- Generate keypoints for each octave by picking pixels in the image that have higher intensity than the surrounding pixels across the DoGs.
- Finally, scale up the lower octaves and overlay the keypoints from different octaves onto the color image to display the keypoints.
- Below code demonstrates the entire process.

Python Code

```
import cv2
import numpy as np
import math

def scaledown_half(img):
    # subsample every other pixel in the image
    resized_img = []
    for i in range(0, img.shape[0], 2):
        row = []
        for j in range(0, img.shape[1], 2):
            row.append(img[i, j])
        resized_img.append(row)
    return np.array(resized_img)

def scaleup_twice(img):
    # scales the image up by twice its dimensions by redundancy
    enlarged_img = []
    for i in range(img.shape[0]):
        if i+1 < img.shape[0]:
            row = []
            for j in range(img.shape[1]):
                if j+1 < img.shape[1]:
                    row.append(img[i, j])
                    row.append(img[i, j+1])
            enlarged_img.append(row)
            enlarged_img.append(row)
    return np.array(enlarged_img)

def flatten_arr(arr):
    result = []
    for i in range(len(arr)):
        for j in range(len(arr[0])):
```

```
            result.append(arr[i][j])
    return result

# convolution operation
def conv(img, kernel, kernel_size):
    result = []
    kernel = kernel[::-1, ::-1]
    for i in range(img.shape[0]):
        if i <= img.shape[0] - kernel_size:
            row = []
            for j in range(img.shape[1]):
                if j <= img.shape[1] - kernel_size:
                    arr = flatten_arr( kernel * img[i:i+kernel_size,
j:j+kernel_size])
                    row.append(sum(arr))
            result.append(row)
    return np.array(result)


def get_guassian(x, y, sigma):
    return (1 / (sigma**2 * (2 * math.pi))) * (math.exp(- (x**2 + y**2) / (2 *
sigma**2)))


def guassian_blur(img, ksize, sigma):
    # blurs the input image using guassian distribution
    guassian = []
    a = []
    for j in range(2, -3, -1):
        temp = []
        for i in range(-2, 3):
            b = get_guassian(i, j, sigma)
            a.append(b)
            temp.append(b)
        guassian.append(temp)
    guassian = np.array(guassian) / sum(a)
    # now convolve guassian with img
    return conv(img, guassian, ksize[0])

def get_keypoint_indices(key_img):
    mask = (key_img == 255)
    points_arr = []
    [points_arr.append([i, j]) for i in range(mask.shape[0]) for j in
range(mask.shape[1]) if mask[i][j] == True]
    return points_arr

def overlay_keypoints(img, key_arr):
    # gets keypoint indices and paints those in red color on color image
    for key_img in key_arr:
        for [x, y] in get_keypoint_indices(key_img):
            img[x, y] = [0, 0, 255]

    return img
```

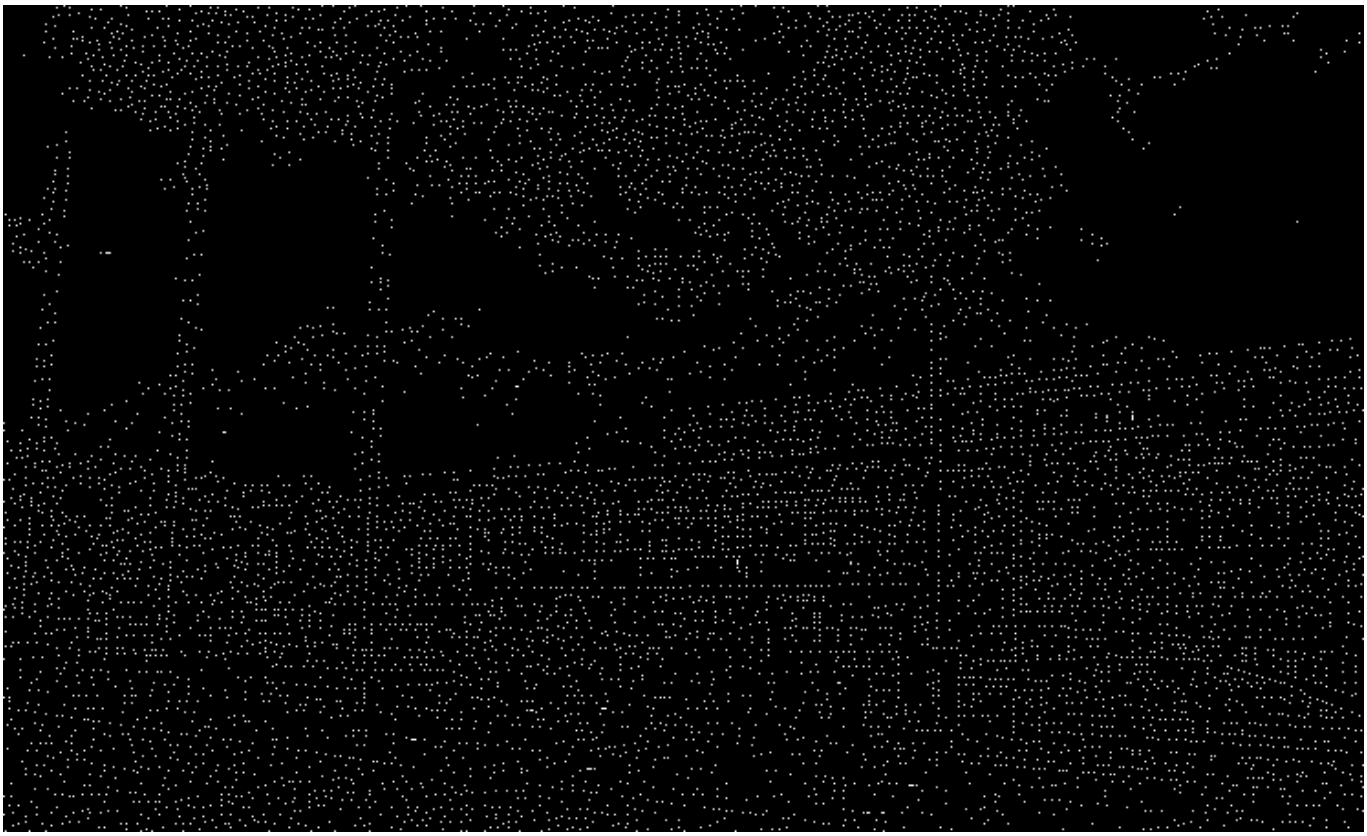
```
def generate_octaves(img):
    # generates 4 octaves
    octaves = []
    octaves.append(img)
    current_img = img
    for i in range(3):
        resized_img = scaledown_half(current_img)
        octaves.append(resized_img)
        current_img = resized_img
        # cv2.namedWindow("", cv2.WINDOW_AUTOSIZE)
        # cv2.imshow("", resized_img)
        # cv2.waitKey(0)
    return octaves

def generate_guassians(octaves, sigmas):
    # create gaussian blurs for images in each octave
    octave_guassians = []
    for octave, sigma_list in zip(octaves, sigmas):
        guassian_blurs = []
        for sigma in sigma_list:
            blur = guassian_blur(octave, (5, 5), sigma)
            guassian_blurs.append(blur)
        octave_guassians.append(guassian_blurs)
    return octave_guassians

def generate_dogs(octave_guassians):
    # takes difference between guassian blurred images
    octave_dogs = []
    for guassian_group in octave_guassians:
        dog_list = []
        i = 0
        while i < len(guassian_group) - 1:
            dog = guassian_group[i] - guassian_group[i - 1]
            dog_flatten = flatten_arr(dog)
            dog = (dog - min(dog_flatten)) / (max(dog_flatten) - min(dog_flatten))
            i += 1
            dog_list.append(dog)
        octave_dogs.append(dog_list)
    return octave_dogs


def generate_keypoints(m1, m2, m3):
    # generates keypoints on the image by comparing maxima between DoGs
    result = []
    for i in range(m1.shape[0]):
        if i <= m1.shape[0] - 3:
            row = []
            for j in range(m1.shape[1]):
                if j <= m1.shape[1] - 3:
                    m1_flatten = flatten_arr(m1[i:i+3, j:j+3])
                    m2_flatten = flatten_arr(m2[i:i+3, j:j+3])
                    m3_flatten = flatten_arr(m3[i:i+3, j:j+3])

                    m2_max = max(m2_flatten)
```

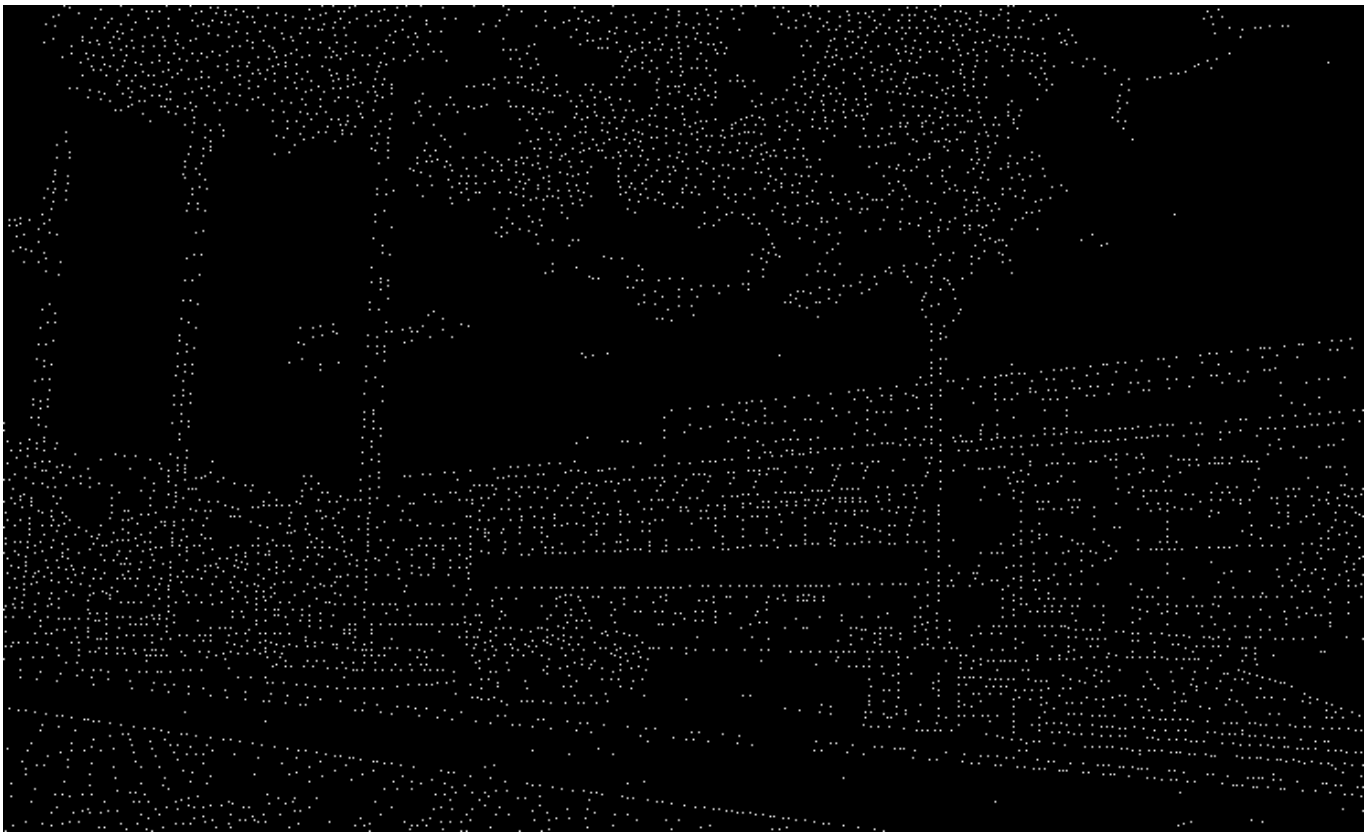
```
                    m3_max = max(m3_flatten)
                    m1_max = max(m1_flatten)
                    if m1[i+1, j+1] != m1_max or m1[i+1, j+1] < m2_max or m1[i+1,
j+1] < m3_max:
                        row.append(0)
                    else:
                        row.append(255)
            result.append(row)
    return result

if __name__ == "__main__":

    # read source image
    img = cv2.imread("./task2.jpg", 0)

    print("Generating 4 octaves ....")
    octaves = generate_octaves(img)

    sigmas = [[2**-0.5, 1, 2**0.5, 2, 2**1.5],
            [2**0.5, 2, 2**1.5, 4, 2**2.5],
            [2**1.5, 4, 2**2.5, 8, 2**3.5],
            [2**2.5, 8, 2**3.5, 16, 2**4.5]
            ]
    print("Generating 5 Guassian blurs each for 4 octaves .... (takes approx. 1
minute)")
    octave_guassians = generate_guassians(octaves, sigmas)

    print("Generating (5-1)=4 DoGs each for 4 octaves .....")
    octave_dogs = generate_dogs(octave_guassians)

    print("Generating 2 keypoint images each for 4 octaves .....")
    keypoint_groups = []
    for dog_group in octave_dogs:
        temp1 = generate_keypoints(dog_group[0], dog_group[1], dog_group[2])
        temp2 = generate_keypoints(dog_group[0], dog_group[2], dog_group[3])

        keypoint_groups.append([np.array(temp1), np.array(temp2)])

    for kg in keypoint_groups:
        for kp in kg:
            cv2.namedWindow("", cv2.WINDOW_AUTOSIZE)
            cv2.imshow("", np.array(kp, dtype=np.uint8))
            cv2.waitKey(0)


    merged_keypoints = []
    merged_keypoints.append(keypoint_groups[0][0])
    merged_keypoints.append(keypoint_groups[0][1])

    for key_group in keypoint_groups[1:]:
        for key_img in key_group:
            merged_keypoints.append(scaleup_twice(key_img))

    color_img = cv2.imread("./task2.jpg")
```

```
    print("Overlaying keypoints on the color image .....")
    key_img = overlay_keypoints(color_img, merged_keypoints)

    cv2.namedWindow("", cv2.WINDOW_AUTOSIZE)
    cv2.imshow("", key_img)
    cv2.waitKey(0)
```

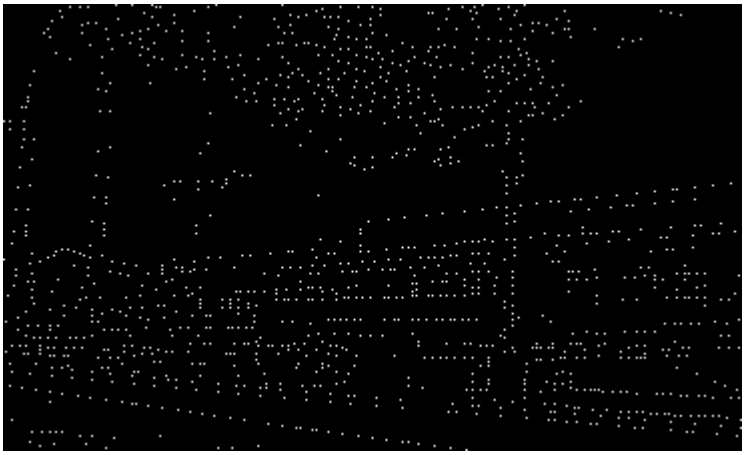
Image Results

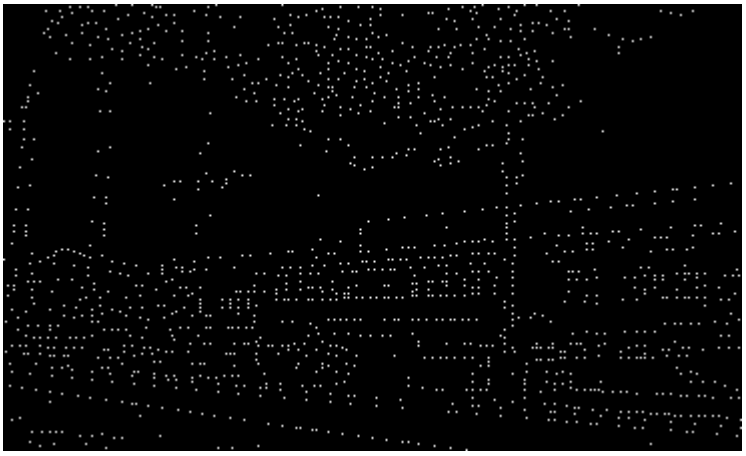
Keypoints for Octave 1:



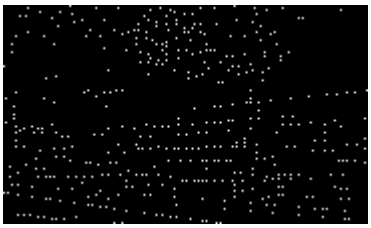


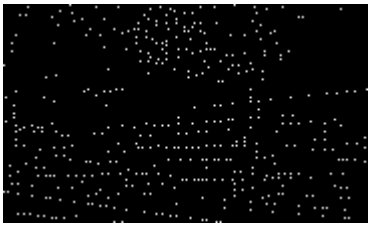
Keypoints for Octave 2:





Keypoints for Octave 3:





Result:

3. Cursor Detection:

Description

The steps to perform cursor detection are:

- Read the input image and template image.
- Apply guassian blur on the input image for smoothing.
- Now calculate laplacian of the guassian image.
- Apply different amounts of scaling on the template image.
- For each scaled template image compare the laplacian of the scaled template to that of the laplacian of guassian image using correlation.
- Find the region having maximum correlation value.

- Repeat the above steps for all scaled templates and find the maximum among the different scales and draw a box around the image patch that has the maximum correlation value.
- Below code demonstrates the idea of template matching by finding cursors in the input images.

Python Code:

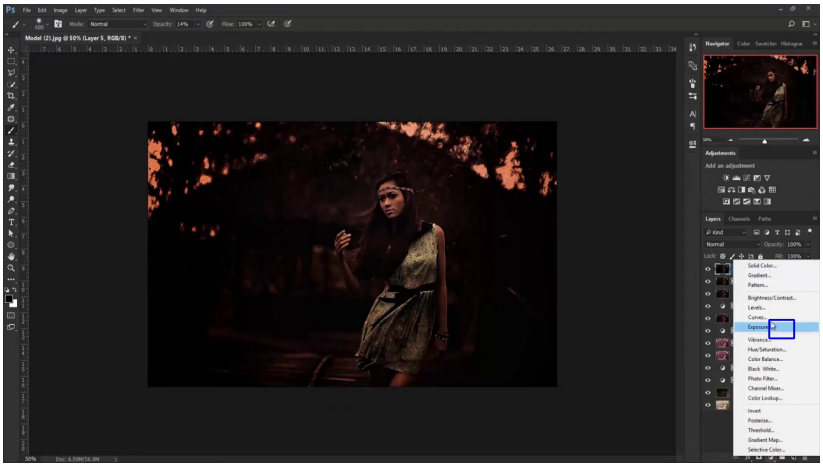
```
import cv2
import numpy as np
import math

def template_matching(img_name, template, threshold=0):
        # input image
        image = cv2.imread(img_name)

        gray_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # template image

        template_gray = cv2.cvtColor(template, cv2.COLOR_BGR2GRAY)

        # apply laplacian of guassian on img
        blurred_img = cv2.GaussianBlur(gray_img, (3, 3), sigmaX=0)
        laplacian_img = cv2.Laplacian(blurred_img, cv2.CV_32F)

        scales = np.linspace(0.05,0.6,20)


        matches = []
        # template scaled to different sizes
        for scale in scales:
            resized_template = cv2.resize(template_gray, (0, 0), fx=scale,
fy=scale)
            laplacian_template = cv2.Laplacian(resized_template, cv2.CV_32F)

            if laplacian_img.shape[0] > laplacian_template.shape[0] and
laplacian_img.shape[1] > laplacian_template.shape[1]:
                # if input size greater than template call match template using
correlation coefficient
                result = cv2.matchTemplate(laplacian_img, laplacian_template,
cv2.TM_CCORR)
                (_, maxVal, _, maxLoc) = cv2.minMaxLoc(result)
                # get the maximum value and index correspongin to the match
                matches.append([maxVal, maxLoc])

        maxVal, maxLoc = max(matches, key=lambda elm: elm[0])
        print(img_name, maxVal)
        (startX, startY) = (int(maxLoc[0]), int(maxLoc[1]))
        (endX, endY) = (int((maxLoc[0] + template.shape[0])), int((maxLoc[1] +
template.shape[1])))

        if maxVal > threshold:
            cv2.rectangle(image,(startX, startY), (endX, endY), 255, 2)
```

```
        cv2.imshow("Image", image)
        cv2.waitKey(0)
        cv2.imwrite("c:/Users/socket_var/Documents/UB
Courses/CVIP/proj1_cse573/"+img_name+"_boxed.jpg", image)

if __name__ == "__main__":
    template = cv2.imread("./task3/template.png")
    for i in range(1, 16):
        template_matching("./task3/pos_%d.jpg" % i, template)
    for j in range(1, 7):
        template_matching("./task3/neg_%d.jpg" % j, template, threshold=450000)
    for k in range(8, 11):
        template_matching("./task3/neg_%d.jpg" % k, template, threshold=450000)
    for i in range(1, 7):
        template_matching(("./task3_bonus/t3_%d.jpg" % i), template)
    template = cv2.imread("./task3_bonus/black_pointer.png")
    for i in range(1, 7):
        template_matching(("./task3_bonus/t2_%d.jpg" % i), template)
    template = cv2.imread("./task3_bonus/hand_pointer.png")
    for i in range(1, 7):
        template_matching(("./task3_bonus/t1_%d.jpg" % i), template)
    template = cv2.imread("./task3_bonus/hand_pointer.png")
    for j in range(1, 7):
        template_matching("./task3_bonus/neg_%d.jpg" % j, template,
threshold=290000)
    for k in range(8, 11):
        template_matching("./task3_bonus/neg_%d.jpg" % k, template,
threshold=290000)
```

Image Results:

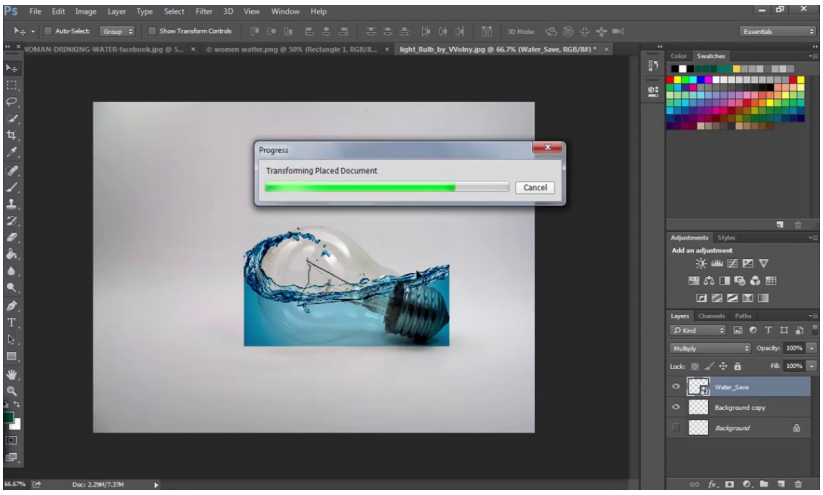
Positive Images:

Accuracy: 14 out of 15: 93.3%



Negative Images:

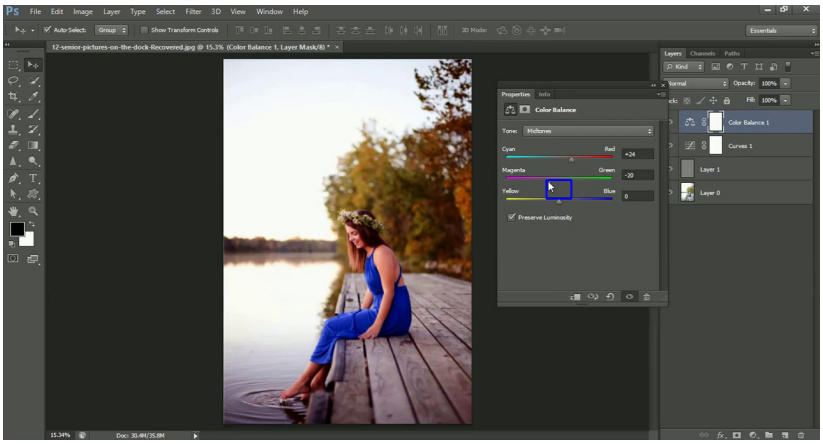
Accuracy: 7 out of 9: 77.7%



Bonus Image Results:

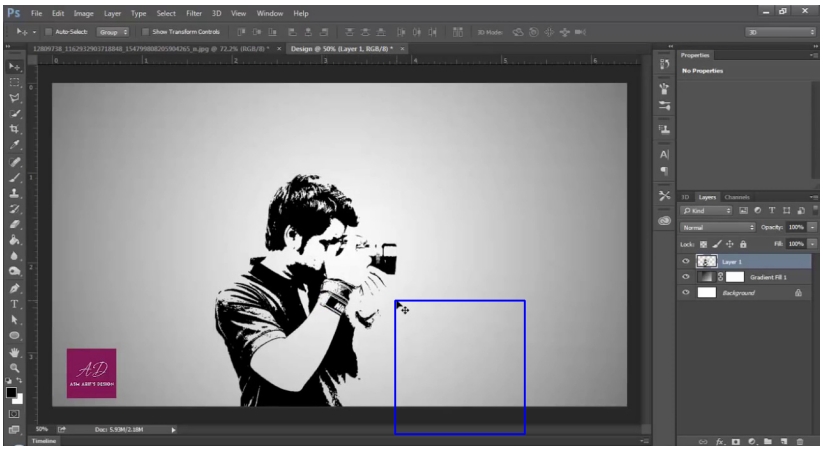
White Cursor :

Accuracy: 6 out of 6: 100%



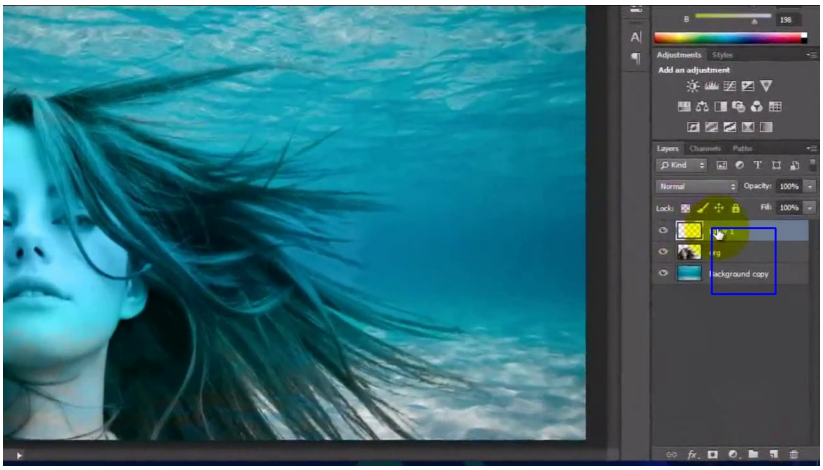
Black Arrow Cursor:

Accuracy: 5 out of 6: 83.33%



Hand Cursor:

Accuracy: 6 out of 6: 100%



Negative Cases:

Accuracy: 9 out of 9: 100%