Reyhane Shahrokhian 99521361

HomeWork3 of Computer Vision Course

Dr. Mohammadi

Q1:

1.1:

The gradient of image I(x, y) would be a 2-dimension vector that can be calculated as below:

$$\partial I(x,\ y) = [\frac{\partial I}{\partial x},\ \frac{\partial I}{\partial y}]$$

1.2:

Calculating the gradient has many advantages such as:

- Feature extraction
- Edge detection
- Sharpening or embossing
- Analyzing the texture or objects shape

1.3:

$$||\partial f|| = mag(\partial f) = \sqrt{g_x^2 + g_y^2} \approx |g_x| + |g_y|$$

1.4:

$$\alpha(x,\ y) = dir(\partial f) = atan2(g_x,\ g_y)$$

1.5:

- Image smoothing using Gaussian filter

- Gradient calculation
- Remove non-maximum values
- Two-stage thresholding

Any pixel that has non-maximum values along its gradient is removed.

The gradient direction is divided into 4 groups and the neighborhood is 3x3.

Any pixel whose gradient size is smaller than T_1 is defined as non-edge and Any pixel whose gradient size is greater than T_2 is defined as an edge. Pixels whose gradient size is between T1 and T2 are defined as edges only if they are connected to an edge pixel directly or through pixels whose gradient size is between T1 and T2.

Canny algorithm has many advantages compared to other methods like:

- Suppresses noise effectively
- Detect both strong and weak edges
- Track edge connectivity
- Providing accurate edge orientation
- Two-stage thresholding

1.6:

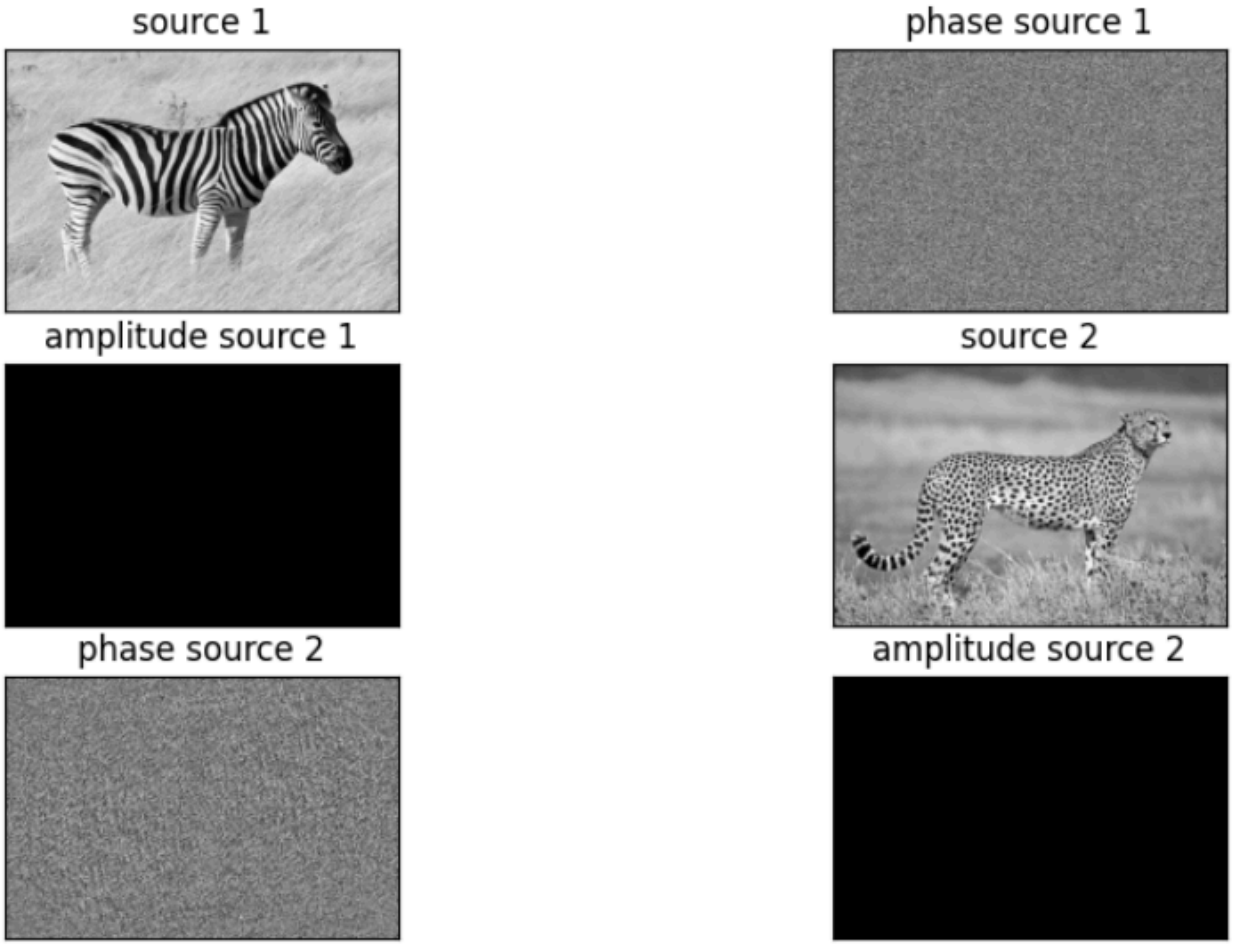
- Lack of thresholding
- Noise sensitivity
- Single kernel

Q2:

2.1:

To show the fourier transform and the phase and amplitude of the images we were supposed to complete the draw_phase_amplitude function. To compute the fourier transform, I called np.fft.fft2() and after that the phase and amplitude can be easily extracted from that:

```
    # Writer your code here
    # Fourier transform
    fourier_t = np.fft.fft2(image)

    phase = np.angle(fourier_t)
    amplitude = np.abs(fourier_t)

    return phase, amplitude
```



2.2:

In the first step, the fourier transform of each image should be calculated and after that the phases of images should be swapped. Then the inverse fourier transform can be calculated using np.fft.ifft2():

```
# Write your code here
# Fourier transform
fourier_t1 = np.fft.fft2(img1)
fourier_t2 = np.fft.fft2(img2)

# Changing phases
fourier_t1_new = np.abs(fourier_t1) * np.exp(1j * np.angle(fourier_t2))
fourier_t2_new = np.abs(fourier_t2) * np.exp(1j * np.angle(fourier_t1))

# Inverse fourier transform
img1 = np.fft.ifft2(fourier_t1_new).real
img2 = np.fft.ifft2(fourier_t2_new).real

return img1, img2
```

new image 1

new image 2

Changing the phases of images, changes the spatial arrangement or phase relationships of frequency components. This can lead to changes in the spatial pattern or texture of the resulting images.

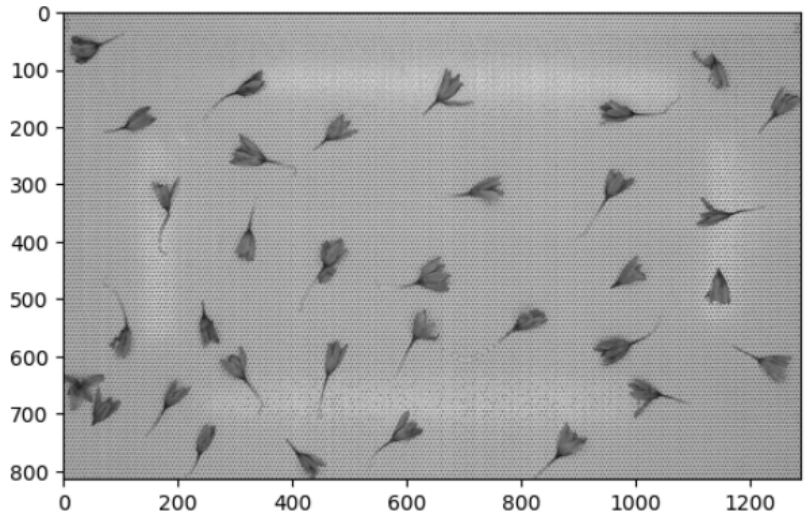
The obtained images may show different visual effects that include changes in texture, contrast, brightness, and spatial arrangement of features.

Q3:

3.1:

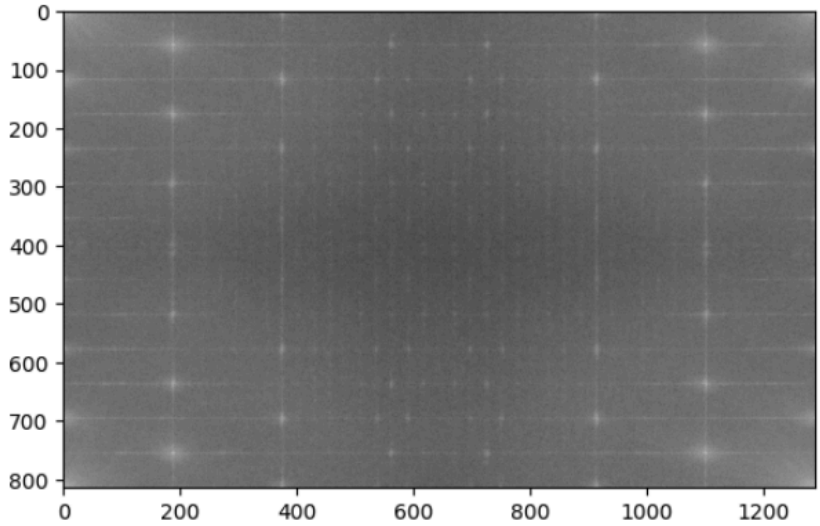
The first step is reading the image:

```
# Read original image
original_image = cv2.imread("/content/saffrun.jpg", cv2.IMREAD_GRAYSCALE)
w, h = original_image.shape[0] , original_image.shape[1]
plt.imshow(original_image, cmap='gray')
plt.show()
```



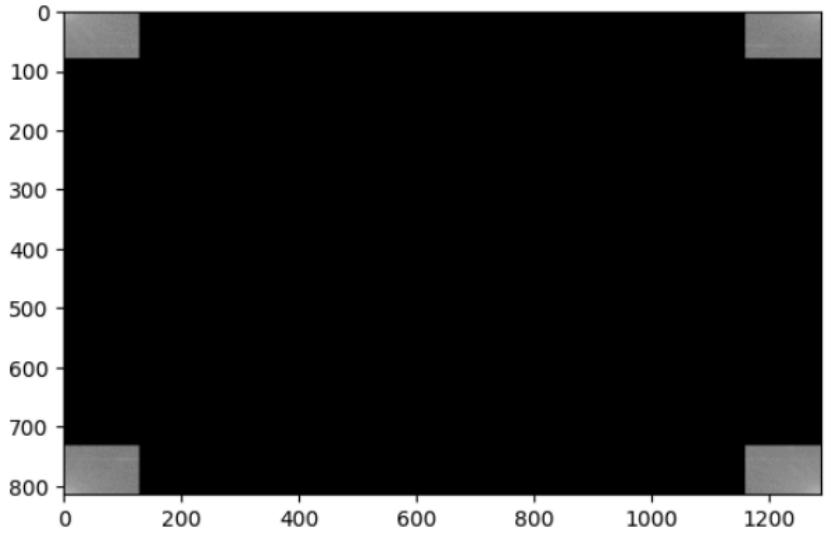
Then, the fourier transform of the image is calculated with np.fft.fft2 to detect the noisy points of the image.

```
# Applying 2D fourier transform
fourier_transform = np.fft.fft2(original_image)
plt.imshow(np.log(1+np.abs(fourier_transform)), cmap='gray')
plt.show()
```

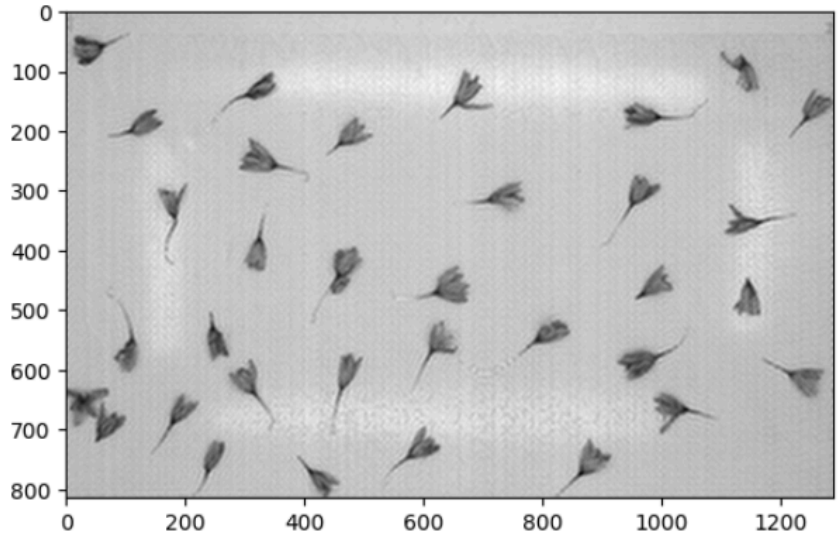


Then by zeroing these noise values, which include a horizontal rectangle and a vertical rectangle, the image noise is removed. I didn't zero the entire screen because there is a lot of information in the corners of the image.

```
# Setting the noisy points to zero
k = 0.1
fourier_transform[int(w*k):int(w*(1-k))] = 0
fourier_transform[:,int(h*k):int(h*(1-k))] = 0
plt.imshow(np.log(1+np.abs(fourier_transform)), cmap='gray')
plt.show()
```



Finally, the image is denoised using an inverse Fourier transform.



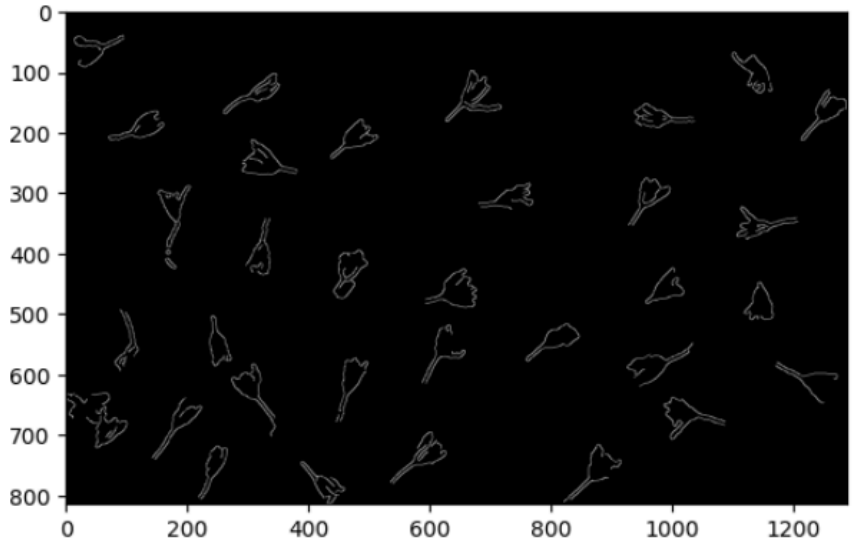
```
# Applying ifft2 for 2D fourier transform
ifft2_image = np.real(np.fft.ifft2(fourier_transform))
plt.imshow(ifft2_image, cmap='gray')
plt.show()
```

I used this source.

3.2:

To detect the edges, I used cv2.Canny() that gets 5 arguments. First argument is our input image. Second and third arguments are our minVal and maxVal respectively. Fourth argument is Sobel kernel size(By default it is 3). Last argument is L2gradient which specifies the equation for finding gradient magnitude. If it is True, it uses the $Edge\ Gradient(G) = \sqrt{G_x^2 + G_y^2}$ which is more accurate, otherwise it uses this equation: $Edge\ Gradient(G) = |G_x| + |G_y|$ (By default, it is False).

```
canny_image = cv2.Canny(np.uint8(ifft2_image), 50, 100, L2gradient=True)
plt.imshow(canny_image, cmap='gray')
plt.show()
```



I used this source.

3.3:

I used a derivative filter to calculate the gradient. Depending on whether the derivative is in the y direction or in the x direction, the filter changes. So. the gradient in both x and y directions should be calculated . For the gradient, I used the sobel filter and then convolved using the cv2.filter2D().

```
# derivative kernels
derivative_kernel_x_axis = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1],
])
derivative_kernel_y_axis = np.array([
    [-1, -2, -1],
    [0, 0, 0],
    [1, 2, 1],
])
```

```
Ix = cv2.filter2D(ifft2_image, -1, derivative_kernel_x_axis)
Iy = cv2.filter2D(ifft2_image, -1, derivative_kernel_y_axis)

arctan = np.arctan2(Iy, Ix)
```

```
arctan
```

```
array([[ 0.        ,  0.        ,  3.14159265, ...,  0.        ,
         0.        ,  0.        ],
       [ 1.57079633,  1.53360547,  2.10410905, ...,  0.62933669,
         0.6962573 ,  1.57079633],
       [ 1.57079633,  1.66174152,  2.07772489, ...,  0.88115437,
         0.95778539,  1.57079633],
       ...,
       [-1.57079633, -1.96280045, -2.6174476 , ..., -0.82276487,
        -0.90450392, -1.57079633],
       [-1.57079633, -1.81284234, -3.08188142, ..., -0.54963116,
        -0.60109731, -1.57079633],
       [ 0.        ,  0.        ,  3.14159265, ...,  0.        ,
         0.        ,  3.14159265]])
```

Now with using np.arctan2(), we can have the direction of obtained gradients. As the outputs are in radians, I used the np.rad2deg() method to convert them to degree.

```
theta = np.array([np.rad2deg(x) for x in arctan])
```

```
1 theta
```

```
array([[   0.        ,    0.        ,  180.        , ...,    0.        ,
           0.        ,    0.        ],
       [  90.        ,   87.86912086,  120.55656795, ...,   36.05833625,
          39.8926048 ,   90.        ],
       [  90.        ,   95.21077586,  119.04486729, ...,   50.48642654,
          54.87706073,   90.        ],
       ...,
       [ -90.        , -112.46018183, -149.96870037, ...,  -47.14095456,
         -51.82425736,  -90.        ],
       [ -90.        , -103.86821509, -176.57879833, ...,  -31.49154576,
         -34.44033874,  -90.        ],
       [   0.        ,    0.        ,  180.        , ...,    0.        ,
           0.        ,  180.        ]])
```

3.4:

First, we obtain a suitable binary image of the saffron flower using the sobel operator. Then, using the voting method and the obtained gradient direction, the places where the gradient changes very quickly are extracted, which is actually the same as the saffron stem. Then, using the hough method, the beginning and end points of these lines are found and the beginning is cut.

Q4:

4.1:

Fourier analysis is widely used in computer vision for various tasks such as image filtering, image reconstruction, image compression, and edge detection.

- Image filtering:

  The Fourier transform can be used to apply filters to an image in the frequency domain, which is more efficient than applying filters in the spatial domain.

- edge detection:

  By applying a logarithmic transformation to the Fourier transform of an image, the

image's frequency components can be visualized, which can be used to detect edges and textures in the image.

- Image reconstruction:

  Fourier analysis can be used in this context to analyze the frequency content of the depth images and to reconstruct the 3D scene from the depth information.

4.2:

The fourier transform of an image is:

$$F(u,\ v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,\ y)\, e^{-j2\pi(ux/M+vy/N)}$$

So:

$$F(0,\ 0) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(0,\ 0)$$

Q5:

As the TA mentioned, there is no need for the explanation. Code is attached.

Q6:

This the formula:

$N = \frac{log(1-P)}{log(1-\omega^{k})}$ in which p is the probability(here 0.99), ω is the probability of selecting an inlier and k is the minimum number of points needed to estimate(here for the circle is 3). So:

$$N = \frac{log(1-0.99)}{log(1-0.4^{3})} = 69.6$$

So we need 70 iterations.

Q7:

7.1:

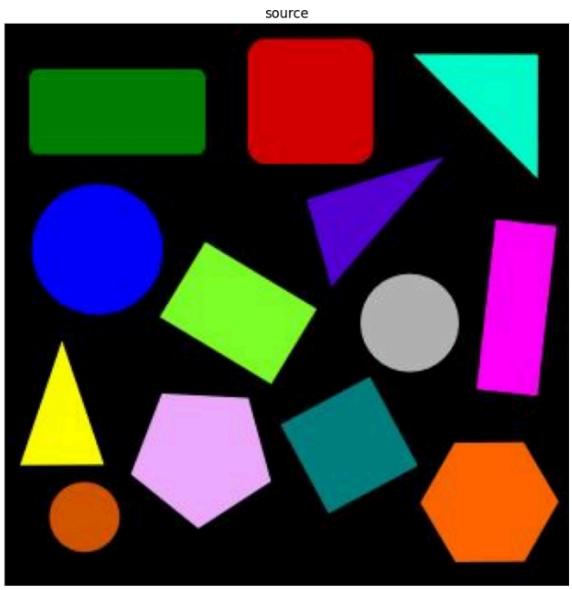
The Hough Transform and Line Segment Detector (LSD) are both widely used methods for line detection, but they have different strengths and weaknesses.

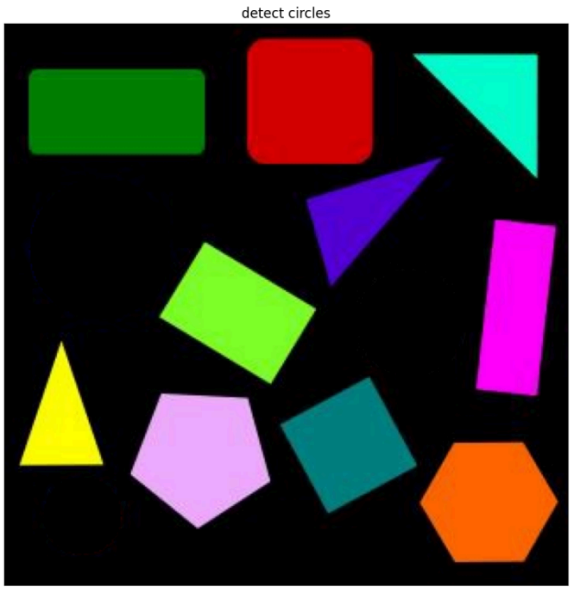
- The Hough transform is robust to noise but LSD can be sensitive to noise and can produce false detections, especially when the edge density is low.
- The Hough transform involves voting in a parameter space to find the best fitting lines. The LSD is based on the gradient of the image.
- LSD is a fast and efficient method. On the other hand, Hough transform can be slow and memory-intensive for large images.
- The Hough Transform requires the setting of several parameters and the performance of that can be sensitive to these parameter settings. But LSD does not require the setting of any parameters

7.2:

After converting the image to grayscale, the circles can be detected using the cv2.HoughCircles() method. Then we can put a black circle with the cv2.circle() method on each in order to remove them. I choose the radius of the black circles a bit larger than what is detected to solve mistakes.

```
# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Detect circles using the Hough Transform
circles = cv2.HoughCircles(gray, cv2.HOUGH_GRADIENT, 1, 20, param1=45, param2=40, minRadius=0, maxRadius=0)

# If circles are detected
if circles is not None:
    # Convert the circles to integer coordinates
    circles = np.round(circles[0, :]).astype("int")

    # Loop over the detected circles
    for (x, y, r) in circles:
        cv2.circle(out_img, (x, y), r + 2, (0, 0, 0), -1)

return out_img
```

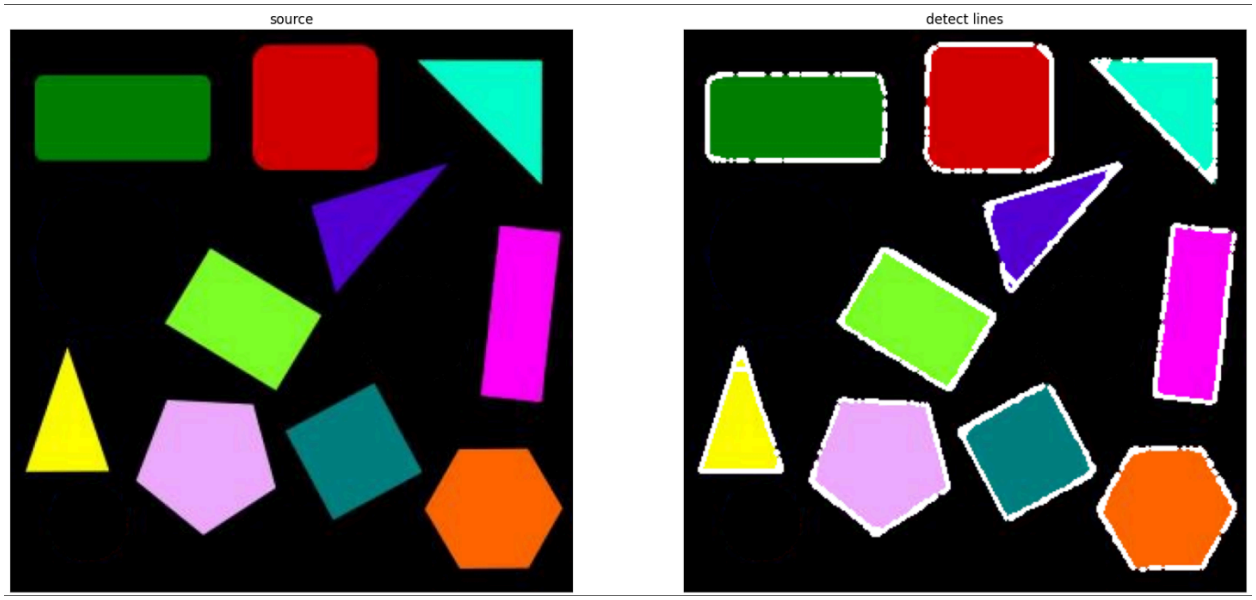




7.3:

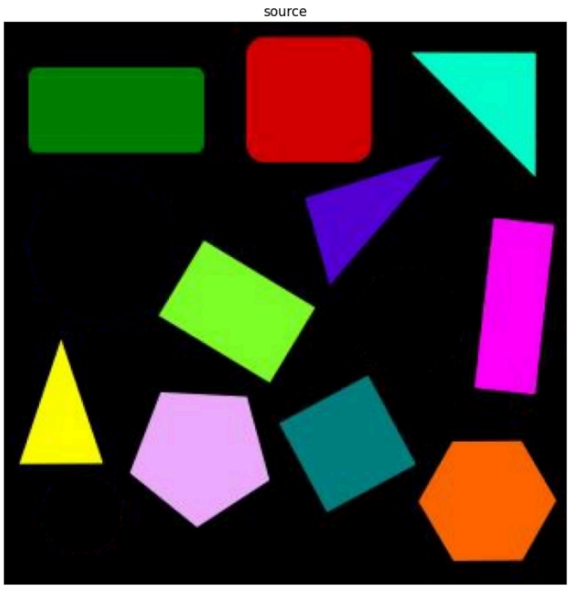
After converting the image to grayscale, I applied the Canny edge detection using cv2.Canny() method and then Hough Transform was performed using cv2.HoughLinesP() method that has many parameters, such as threshold which is minimum count of votes, minLineLength that shows the minimum length of line we wanted, and maxLineGap that indicates the gap that is linked. Finally the detected lines are drawn using cv2.line() method.

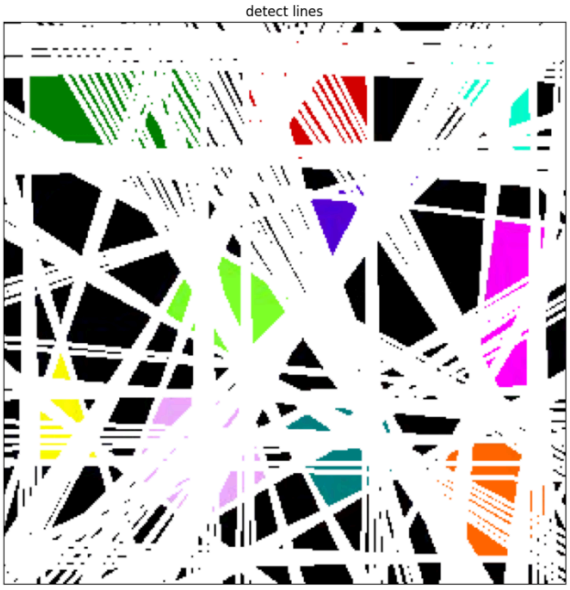
```
#Writer your code here
# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply Canny edge detection
edges = cv2.Canny(gray, 22, 105, L2gradient=True)

# Perform Hough Line Transform
lines = cv2.HoughLinesP(edges, 1, np.pi/180, threshold=2, minLineLength=1, maxLineGap=7)

# Draw the detected lines
if lines is not None:
    for line in lines:
        x1, y1, x2, y2 = line[0]
        cv2.line(out_img, (x1, y1), (x2, y2), (255, 255, 255), 2)
return out_img
```



I also tried to implement it with the cv2.HoughLines() method but I couldn't get the satisfying result:(.

```python
def detect_lines_hough(image):
    out_img = image.copy()
    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Canny edge detection
    edges = cv2.Canny(gray, 22, 105, L2gradient=True)

    # Detect lines using Hough Line Transform
    lines = cv2.HoughLines(edges, 1, np.pi/180, 30)

    # Iterate over the detected lines
    if lines is not None:
      for line in lines:
          rho, theta = line[0]

          # Convert the line to Cartesian coordinates
          a = np.cos(theta)
          b = np.sin(theta)
          x0 = a*rho
          y0 = b*rho
          x1 = int(x0 + 1000*(-b))
          y1 = int(y0 + 1000*(a))
          x2 = int(x0 - 1000*(-b))
          y2 = int(y0 - 1000*(a))

          # Draw the line on the image
          cv2.line(out_img, (x1, y1), (x2, y2), (255, 255, 255), 2)

    return out_img
```

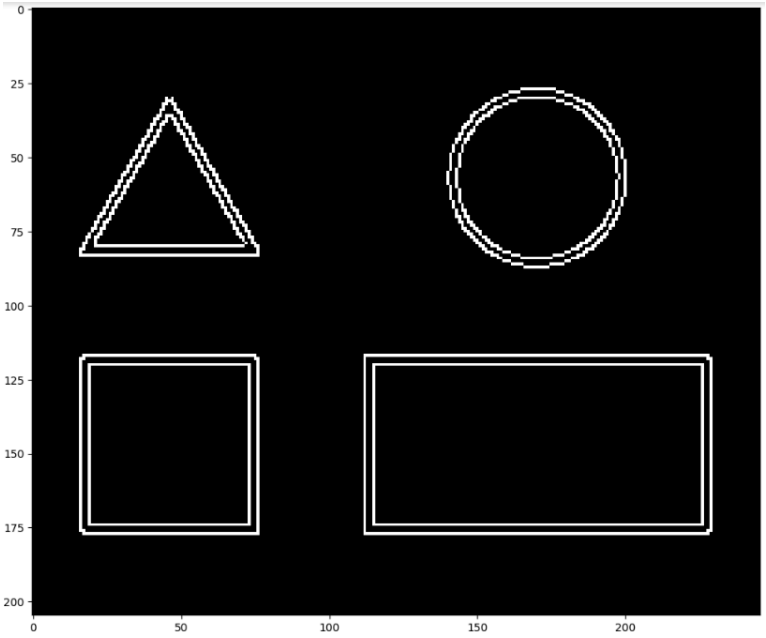




7.4:

First, the image should be converted to grayscale and then the Canny must be applied on that to detect the edges.

```
image_d = plt.imread("/content/7.jpg")
# Convert the image to grayscale
gray_d = cv2.cvtColor(image_d, cv2.COLOR_BGR2GRAY)
# Apply Canny edge detection
edges_d = cv2.Canny(gray_d, 50, 150, L2gradient=True)

plt.imshow(edges_d, cmap="gray")
```



Now to find contours in the image, I used cv2.findContours() method.

```
contours, hierarchy = cv2.findContours(edges_d//255, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
```

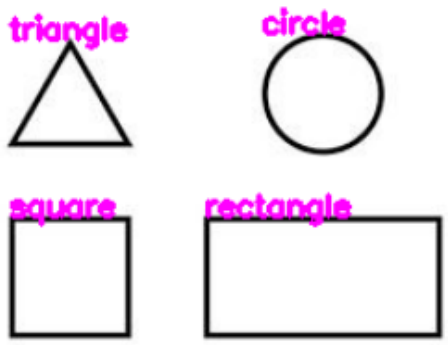
Then I wrote a function that takes the image and a list of contours and it approximates each contour, draws the approximated contours on the image, and returns the modified image along with a list of the approximated contours. With the approximated contours for each shape we can find what type of each using the classify_shapes() function that I have written.

```
def draw_contour(img, contours):
    approximates = []
    # Calculating an approximation of each contour
    for cnt in contours:
        epsilon = 0.03 * cv2.arcLength(cnt, True) # The epsilon parameter controls the approximation accuracy
        approx = cv2.approxPolyDP(cnt, epsilon, True)
        approximates.append(approx)
        cv2.drawContours(img, [approx], 0, (255, 0, 0), 2)
    return img, approximates
```

```
def classify_shapes(approximates):
    result = []
    for approx in approximates:
        vertice_cnt = len(approx)
        if vertice_cnt == 3:
            shape = "triangle"
        elif vertice_cnt == 4:
            # Cooridinates of the smallest rectangle that covers these points
            x, y, w, h = cv2.boundingRect(approx)
            w2h_ratio = float(w)/h
            if w2h_ratio >= 0.95 and w2h_ratio <= 1.05:
                shape = "square"
            else:
                shape = "rectangle"
        else:
            shape = "circle"
        x,y,w,h = cv2.boundingRect(approx)
        result.append((x,y,shape))
    return result
```

Now by calling this function for each shape we can get their type:

```
from google.colab.patches import cv2_imshow

result = classify_shapes(approximates)
for item in result:
  x,y,shape = item[0], item[1], item[2]
  cv2.putText(image_d, shape, (x,y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 255), 2)

cv2_imshow(image_d)
cv2.waitKey(0)
cv2.destroyAllWindows()
```



Q8:

This is the added code:

```
# Transform each point (x, y) in image
# Find rho corresponding to values in thetas
# and increment the accumulator in the corresponding coordinate.
### YOUR CODE HERE
for x, y in zip(xs, ys):
    for i in range(num_thetas):
        rho = round(x * cos_t[i] + y * sin_t[i])
        accumulator[rho + diag_len, i] += 1
### END YOUR CODE

return accumulator, rhos, thetas
```

Output: