CO543: Image Processing
Lab 6 – Feature Extraction

Ranage R.D.P.R. - E/19/310

The following is the original image that was used to implement the functions in the lab tasks.

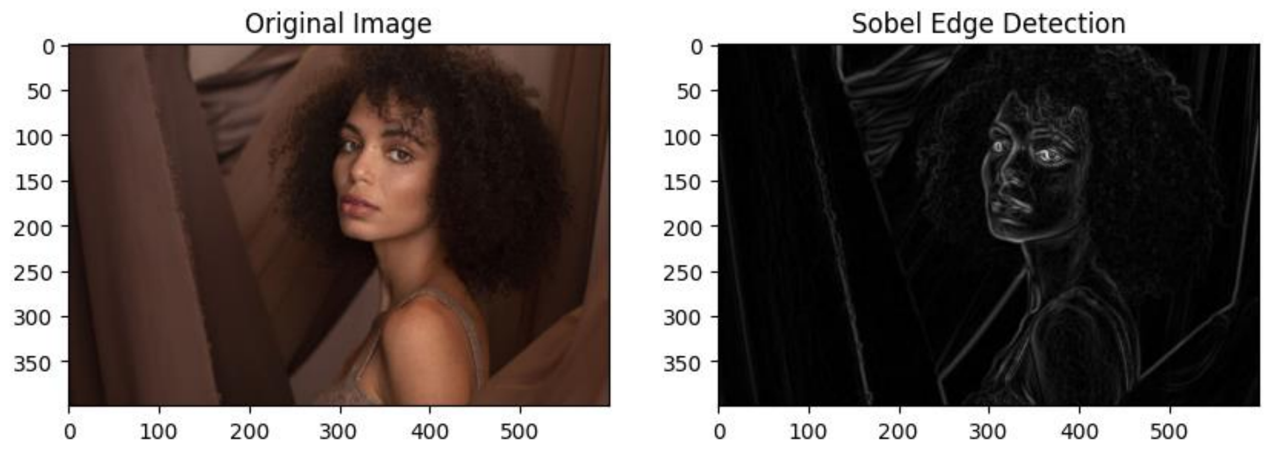
The following functions are used throughout the lab to plot each figure to show the results.

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

def show_2_images(image1, image2, title1, title2):
    fig, ax = plt.subplots(1, 2, figsize=(10, 5))

    ax[0].imshow(image1, cmap='gray')
    ax[0].set_title(title1)

    ax[1].imshow( image2, cmap='gray')
    ax[1].set_title(title2)

    plt.show()

def show_3_images(image1, image2, image3, title1, title2, title3):
    fig, ax = plt.subplots(1, 3, figsize=(15, 5))

    ax[0].imshow(image1, cmap='gray')
    ax[0].set_title(title1)

    ax[1].imshow( image2, cmap='gray')
    ax[1].set_title(title2)

    ax[2].imshow( image3, cmap='gray')
    ax[2].set_title(title3)

    plt.show()
```

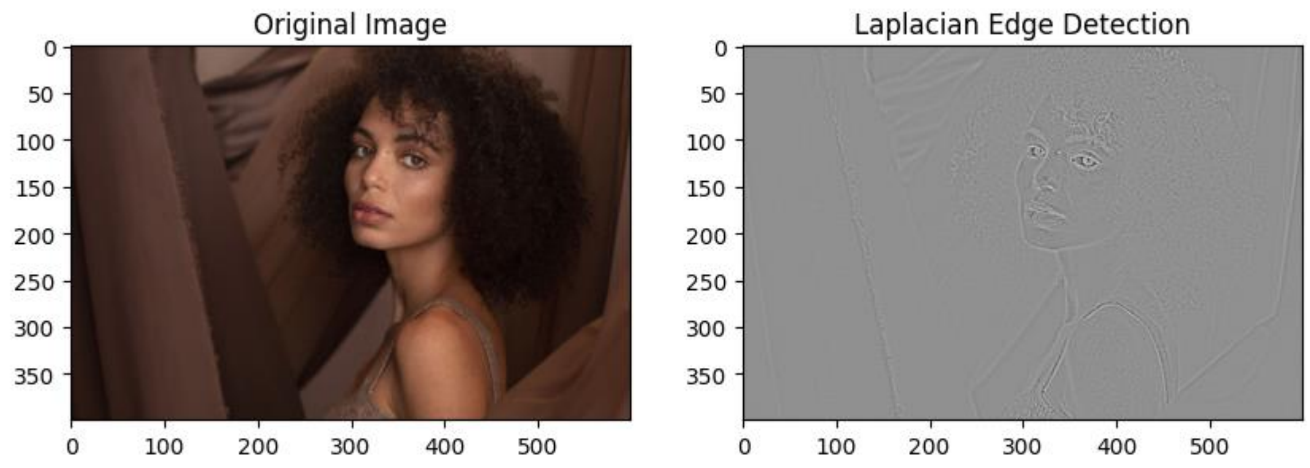
Lab Task 1: Edge Detection

1.1) Identify the different edges present in an image using Sobel, Laplacian, and Canny edge detection algorithms, and discuss the differences in their outputs.

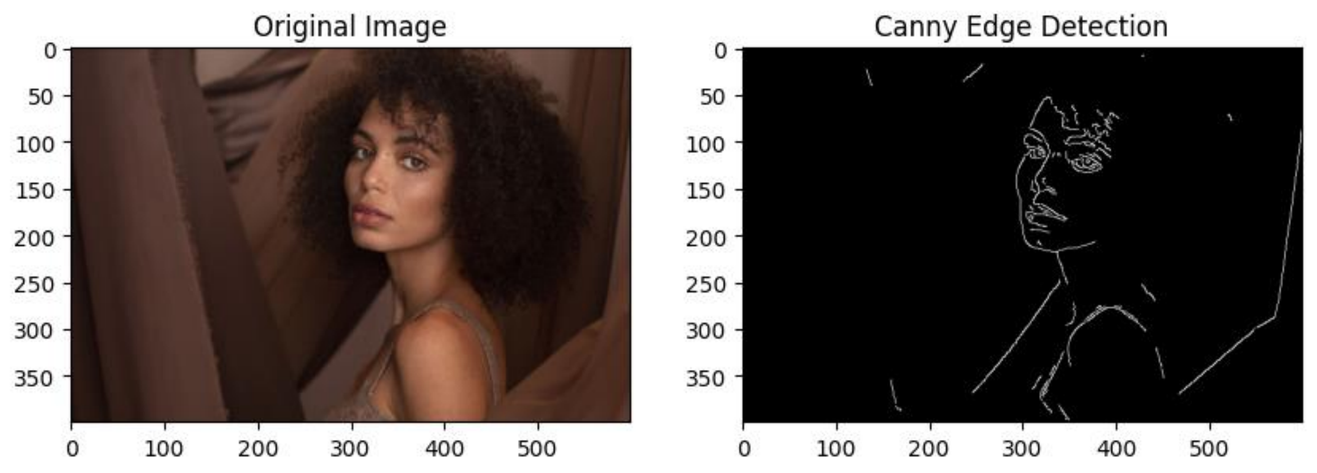
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image
img = plt.imread('image.jpg')

# Sobel Edge Detection
def sobel_edge_detection(img):
    # Convert the image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian Blur
    blur = cv2.GaussianBlur(gray, (3, 3), 0)

    # Apply Sobel operator in X direction
    sobelx = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=5)

    # Apply Sobel operator in Y direction
    sobely = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=5)

    # Calculate the gradient magnitude
    magnitude = np.sqrt(sobelx**2 + sobely**2)

    return magnitude

sobel_img = sobel_edge_detection(img)
show_2_images(img, sobel_img, 'Original Image', 'Sobel Edge Detection')
```



```
# Laplacian Edge Detection
def laplacian_edge_detection(img):
    # Convert the image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian Blur
    blur = cv2.GaussianBlur(gray, (3, 3), 0)

    # Apply Laplacian operator
    laplacian = cv2.Laplacian(blur, cv2.CV_64F)

    return laplacian

laplacian_img = laplacian_edge_detection(img)
show_2_images(img, laplacian_img, 'Original Image', 'Laplacian Edge Detection')
```



```
# Canny Edge Detection
def canny_edge_detection(img):

    # Apply Gaussian Blur
    blur = cv2.GaussianBlur(img, (3, 3), 0)

    # Apply Canny Edge Detection
    canny = cv2.Canny(blur, 100, 150)

    return canny

canny_img = canny_edge_detection(img)
show_2_images(img, canny_img, 'Original Image', 'Canny Edge Detection')
```



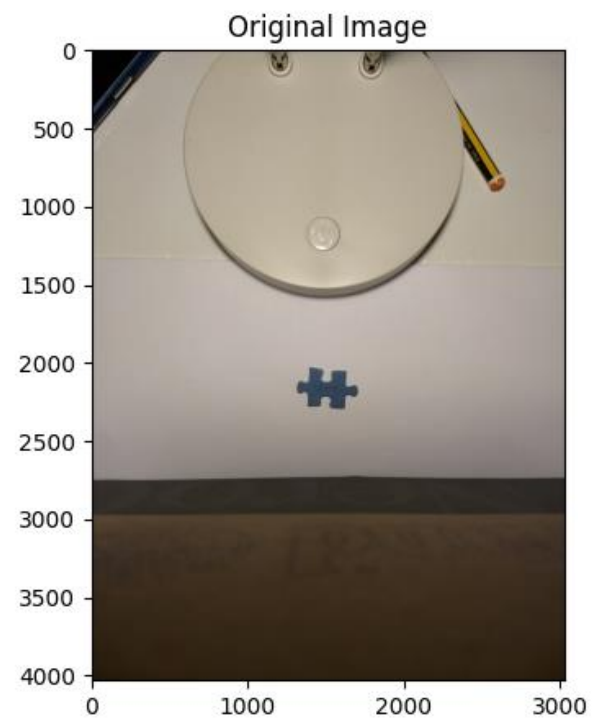
Sobel Edge Detection: Calculates gradients in both x and y directions to capture edge information, particularly in specific orientations. It is sensitive to noise, resulting in less clean edges compared to other methods like Canny.

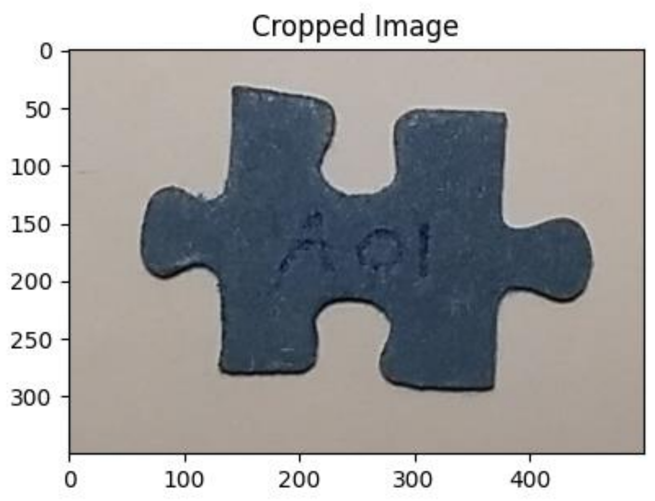
Laplacian Edge Detection: Highlights regions of rapid intensity change and detects edges regardless of orientation. However, it is more sensitive to noise, which can lead to the detection of spurious edges.

Canny Edge Detection: A sophisticated method that reduces noise, resulting in cleaner edges. It uses steps like Gaussian blur, gradient calculation, non-maximum suppression, and edge tracking by hysteresis, making it accurate and widely preferred for detecting true edges.

1.2) Using the provided image jigsaw.jpg, identify the boundary lines of the puzzle piece. Follow the below steps to obtain the lines:
i. Crop the region containing the puzzle piece using a simple python matrix manipulation.

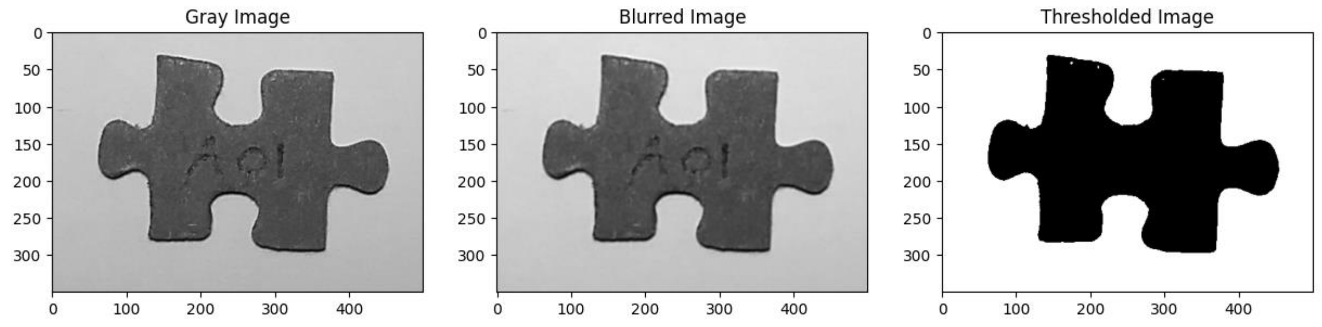
```python
# Load the image
image = cv2.imread('jigsaw.jpg')
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Define the cropping coordinates
x_start, y_start, x_end, y_end = 1250, 2000, 1750, 2350

# Crop the image
cropped_image = image_rgb[y_start:y_end, x_start:x_end]
```





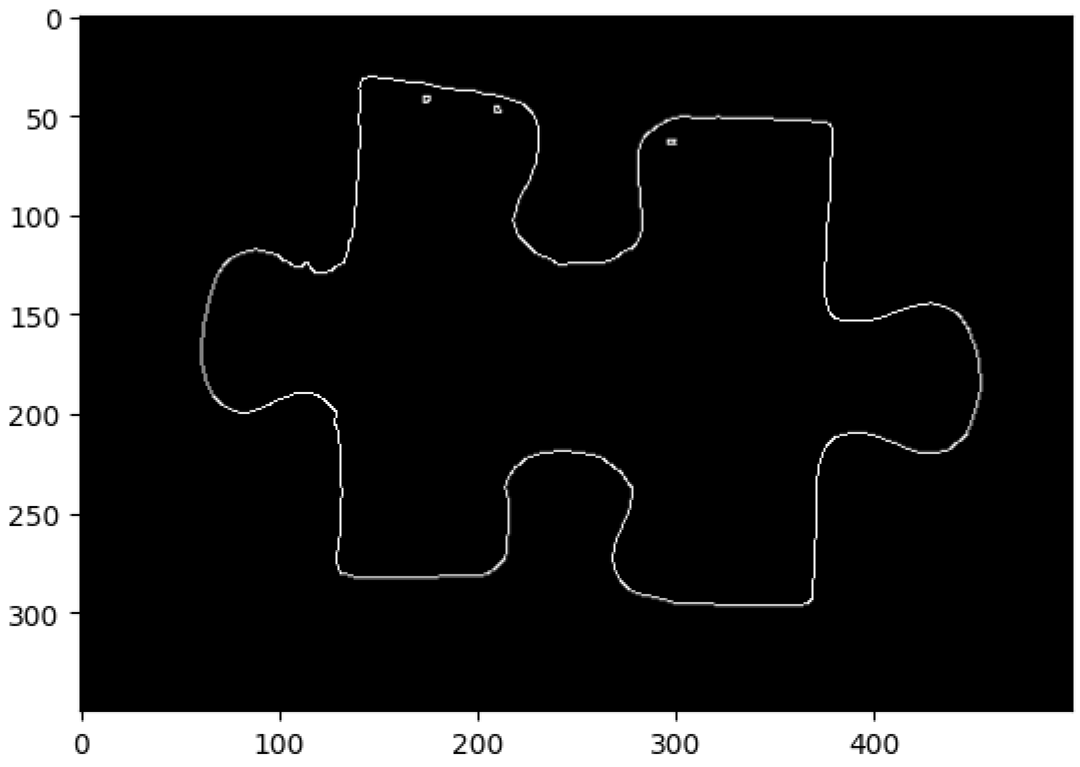
ii. Apply simple preprocessing techniques to convert the cropped image to grayscale and remove noises (e.g: Blurring, Thresholding).

```
gray_image = cropped_image.mean(axis=2)

# Apply Gaussian Blurring
blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 0)

# Apply Thresholding
_, thresholded_image = cv2.threshold(blurred_image, 127, 255, cv2.THRESH_BINARY)

thresholded_image = thresholded_image.astype('uint8')

show_3_images(gray_image, blurred_image, thresholded_image, 'Gray Image', 'Blurred
Image', 'Thresholded Image')
```



iii. Perform edge detection on the binarized image using a suitable edge detection algorithm (e.g: Canny Edge Detection).

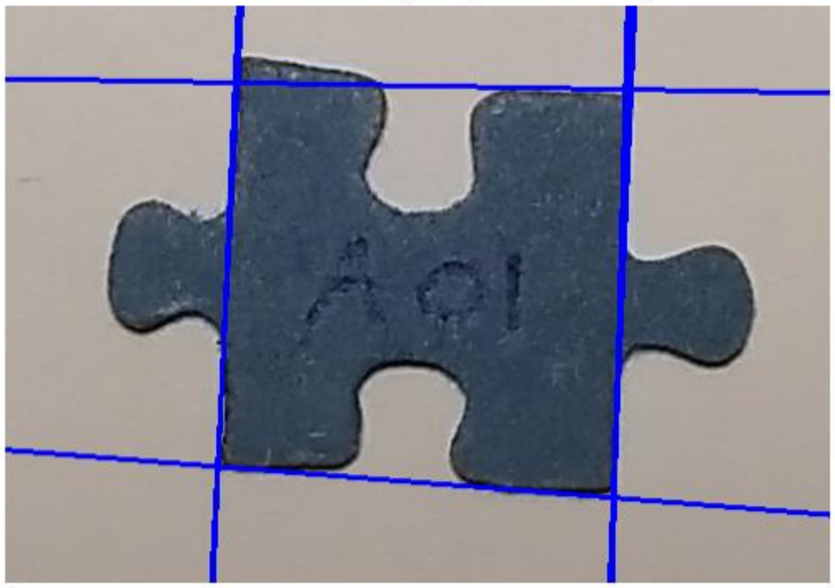
```
cannyImg = canny_edge_detection(thresholded_image)
plt.imshow(cannyImg, cmap='gray')
```



iv. Apply the Hough Transform (cv2.HoughLines)to detect lines that bound the four main corners of the jigsaw piece.

```
def hough_line_detection(edges, img):
    lines = cv2.HoughLines(edges, rho=2, theta=np.pi/360, threshold=80)

    img_copy = img.copy()
    if lines is not None:
        for line in lines:
            rho, theta = line[0]
            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))
            cv2.line(img_copy, (x1, y1), (x2, y2), (0, 0, 255), 2)

    return img_copy

img_with_lines = hough_line_detection(cannyImg, cropped_image)

plt.imshow(img_with_lines, cmap='gray')
plt.title('Detected Lines (Hough Transform)')
plt.axis('off')
plt.show()
```

Detected Lines (Hough Transform)



v. Explain the impact of the rho, theta, and threshold parameters of Hough transformation in detecting lines.
Rho (ρ) determines how finely distances are measured from the origin to detect lines. Smaller rho values give more precise results but require more computation. Larger rho values are faster but might miss details.

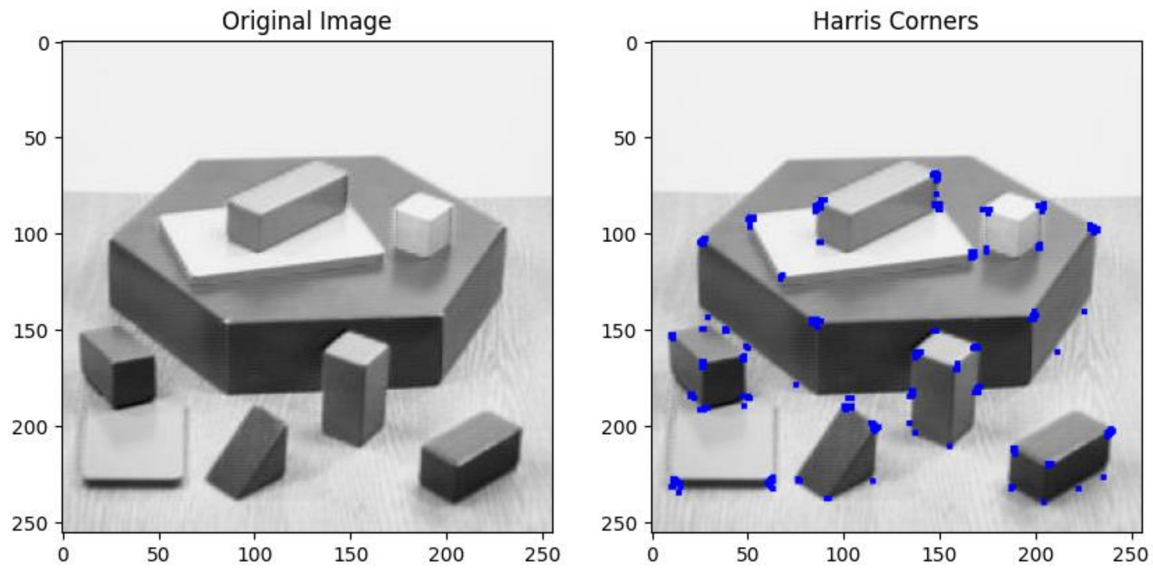
Theta (θ) decides the angular resolution for detecting lines of different orientations. Smaller theta values detect lines more accurately but are slower. Larger theta values are faster but might confuse lines that are nearly parallel or perpendicular.

Threshold sets the minimum votes needed in the Hough space to confirm a line. A higher threshold avoids detecting noise but might miss faint lines. A lower threshold detects more lines, including noise.

Lab Task 2: Corner Detection

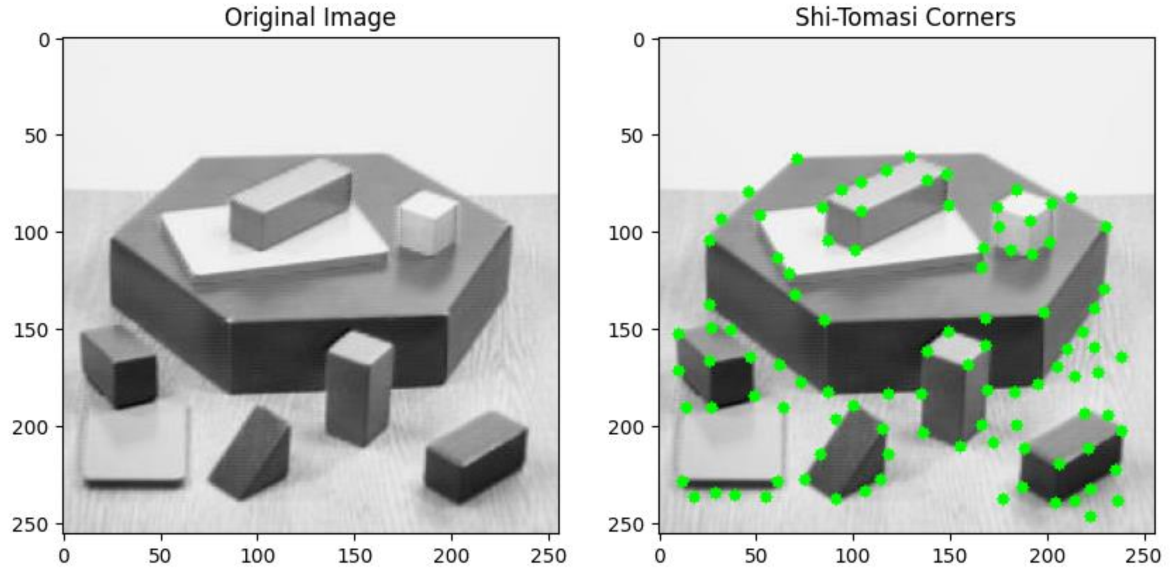
2.1) Apply Harris, Shi-Tomasi, and SIFT algorithms on an image to identify corners and discuss the differences in these algorithms.

```
def apply_harris(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    gray = np.float32(gray)
    harris_corners = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)
    harris_corners = cv2.dilate(harris_corners, None)
    img_copy = img.copy()  # Create a writable copy of the image
    img_copy[harris_corners > 0.01 * harris_corners.max()] = [0, 0, 255]
    return img_copy

img = plt.imread('blox.jpg')

harris_img = apply_harris(img)
show_2_images(img, harris_img, 'Original Image', 'Harris Corners')
```

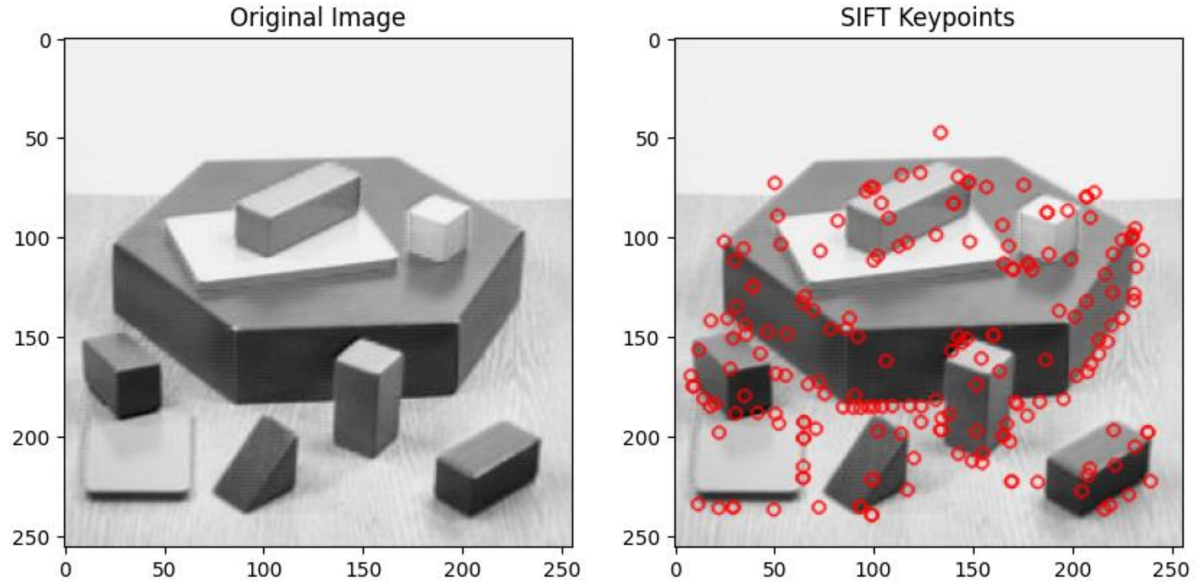


```
def apply_shi_tomasi(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    corners = cv2.goodFeaturesToTrack(gray, maxCorners=100, qualityLevel=0.01,
minDistance=10)
    corners = np.intp(corners)
    img_copy = img.copy()  # Create a writable copy of the image
    for corner in corners:
        x, y = corner.ravel()
        cv2.circle(img_copy, (x, y), 3, (0, 255, 0), -1)
    return img_copy
```

```python
shi_tomasi_img = apply_shi_tomasi(img)
show_2_images(img, shi_tomasi_img, 'Original Image', 'Shi-Tomasi Corners')
```



```python
def apply_sift(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create()
    keypoints, descriptors = sift.detectAndCompute(gray, None)
    img_copy = img.copy()  # Create a writable copy of the image
    img_copy = cv2.drawKeypoints(img_copy, keypoints, None, color=(255, 0, 0))
    return img_copy

sift_img = apply_sift(img)
show_2_images(img, sift_img, 'Original Image', 'SIFT Keypoints')
```



2.2) Using the provided image jigsaw.jpg, identify the corners present in the puzzle piece.

```
gray = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2GRAY)

# Shi-Tomasi Corner Detection
shi_tomasi_corners = cv2.goodFeaturesToTrack(gray, 10, 0.01, 10)
shi_tomasi_corners = np.int0(shi_tomasi_corners)

# Plot the results
for corner in shi_tomasi_corners:
    x = corner[0][0]
    y = corner[0][1]

    #filtration to get the required points only based on observation
    if x < 120 or (x>145 and x<360) or (y>100 and y<250) :
        continue
    x, y = corner.ravel()
    cv2.circle(cropped_image, (x, y), 5, 255, -1)
plt.imshow(cropped_image, cmap='gray')
```