CO 543 Image Processing

Lab 06

Jayawardena H.D.N.S.

E/19/167

Feature Extraction

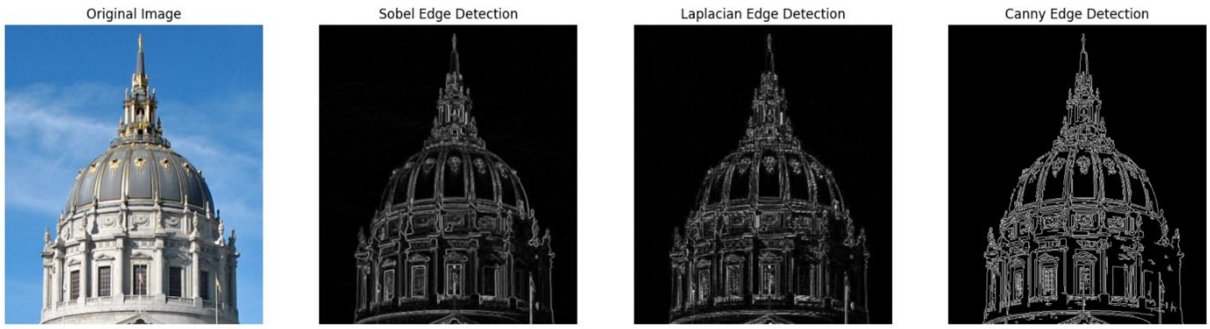
Lab Task 1: Edge Detection

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.

Code

```
import cv2
import matplotlib.pyplot as plt

# Load the image
image_path = '/content/city_hall_zoom.png'
image = cv2.imread(image_path, cv2.IMREAD_COLOR)

# Convert to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply Sobel edge detection
sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=3)
sobel_combined = cv2.magnitude(sobel_x, sobel_y)

# Apply Laplacian edge detection
laplacian = cv2.convertScaleAbs(cv2.Laplacian(gray_image, cv2.CV_64F))

# Apply Canny edge detection
canny = cv2.Canny(gray_image, 100, 200)

# Display the results
fig, axs = plt.subplots(1, 4, figsize=(20, 5))

# Original image
axs[0].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
axs[0].set_title('Original Image')
axs[0].axis('off')

# Sobel edge detection
axs[1].imshow(sobel_combined, cmap='gray')
axs[1].set_title('Sobel Edge Detection')
axs[1].axis('off')

# Laplacian edge detection
axs[2].imshow(laplacian, cmap='gray')
axs[2].set_title('Laplacian Edge Detection')
axs[2].axis('off')

# Canny edge detection
axs[3].imshow(canny, cmap='gray')
axs[3].set_title('Canny Edge Detection')
axs[3].axis('off')

plt.show()
```

Output



Sobel Edge Detection:

- Uses two kernels (Gx and Gy) to detect changes in intensity in horizontal and vertical directions.
- Highlights edges in a specific direction (usually horizontal or vertical).

Laplacian Edge Detection:

- Uses a single kernel that detects edges in all directions.
- Sensitive to noise but highlights areas of rapid intensity change.

Canny Edge Detection:

- Uses a multi-stage process involving Gaussian filtering, gradient calculation, non-maximum suppression, and edge tracking by hysteresis.
- Known for its ability to detect true edges while minimizing noise and avoiding false positives.

1.2 Using the provided image jigsaw.jpg, identify the boundary lines of the puzzle piece.

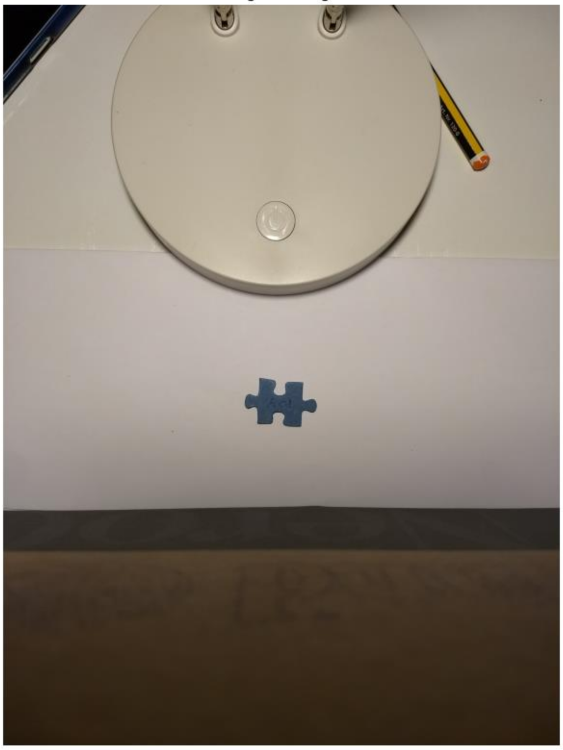
Code

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

# Load the image
image_path = '/content/jigsaw.jpg'
image = cv2.imread(image_path, cv2.IMREAD_COLOR)
#print(image.shape)

# Display the original image
plt.figure(figsize=(10, 10))
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')
plt.show()

# Step 1: Crop the image
x, y, w, h = 1300, 2024, 450, 300
cropped_image = image[y:y+h, x:x+w]

# Display the cropped image
plt.figure(figsize=(10, 10))
plt.imshow(cv2.cvtColor(cropped_image, cv2.COLOR_BGR2RGB))
plt.title('Cropped Image')
plt.axis('off')
plt.show()
```

```
# Step 2: Preprocess the image
gray_image = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2GRAY)
blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 0)
_, binary_image = cv2.threshold(blurred_image, 127, 255, cv2.THRESH_BINARY_INV)

# Display the binary image
plt.figure(figsize=(10, 10))
plt.imshow(binary_image, cmap='gray')
plt.title('Binary Image')
plt.axis('off')
plt.show()

# Step 3: Perform edge detection
edges = cv2.Canny(binary_image, 50, 150, apertureSize=3)

# Display the edges
plt.figure(figsize=(10, 10))
plt.imshow(edges, cmap='gray')
plt.title('Edges Detected')
plt.axis('off')
plt.show()

# Step 4: Apply Hough Transform to detect lines
lines = cv2.HoughLines(edges, 4, np.pi / 180, 100)

# Create a copy of the cropped image to draw lines on
line_image = cropped_image.copy()

# Draw the detected lines on the image
if lines is not None:
    for rho, theta in lines[:, 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(line_image, (x1, y1), (x2, y2), (0, 0, 255), 2)

# Display the image with detected lines
plt.figure(figsize=(10, 10))
plt.imshow(cv2.cvtColor(line_image, cv2.COLOR_BGR2RGB))
plt.title('Detected Lines')
plt.axis('off')
plt.show()
```

Input image

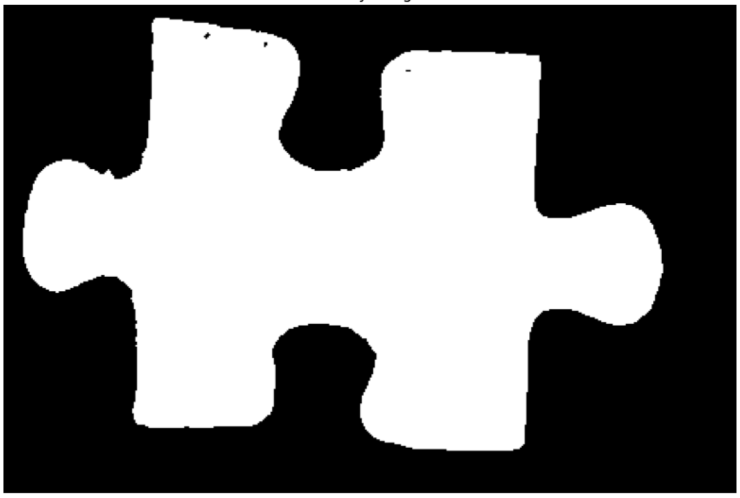
Original Image



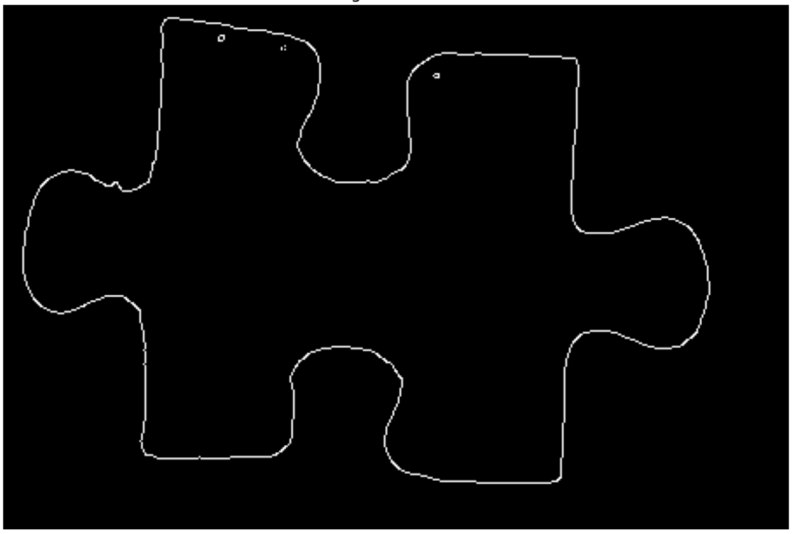
Output

Cropped Image

Binary Image



Edges Detected



Detected Lines

V. Explain the impact of the rho, theta, and threshold parameters of Hough transformation in detecting lines.

Rho (rho):

- **Description**: The distance resolution of the accumulator in pixels.
- **Impact**:
    - Smaller values of rho result in a higher resolution of the parameter space, allowing for more precise detection of line positions but increasing computational complexity and potential noise.
    - Larger values of rho result in a lower resolution of the parameter space, reducing computational complexity and noise but potentially decreasing the precision of detected line positions.

Theta (theta):

- **Description**: The angle resolution of the accumulator in radians.
- **Impact**:
    - Smaller values of theta (higher angular resolution) increase the precision of line orientation detection but also increase computational complexity.
    - Larger values of theta (lower angular resolution) reduce computational complexity and noise but may lead to less accurate line orientation detection.

Threshold (threshold):

- **Description**: The minimum number of intersections in the accumulator required to detect a line.
- **Impact**:
    - Higher values of threshold result in fewer lines being detected, as only the most prominent lines with a significant number of intersections in the accumulator are considered.
    - Lower values of threshold result in more lines being detected, including weaker lines with fewer intersections, which may increase noise and false positives.

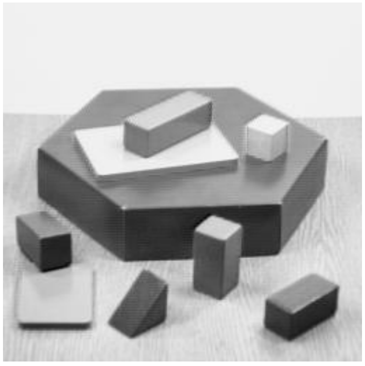
Lab Task 2: Corner Detection

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

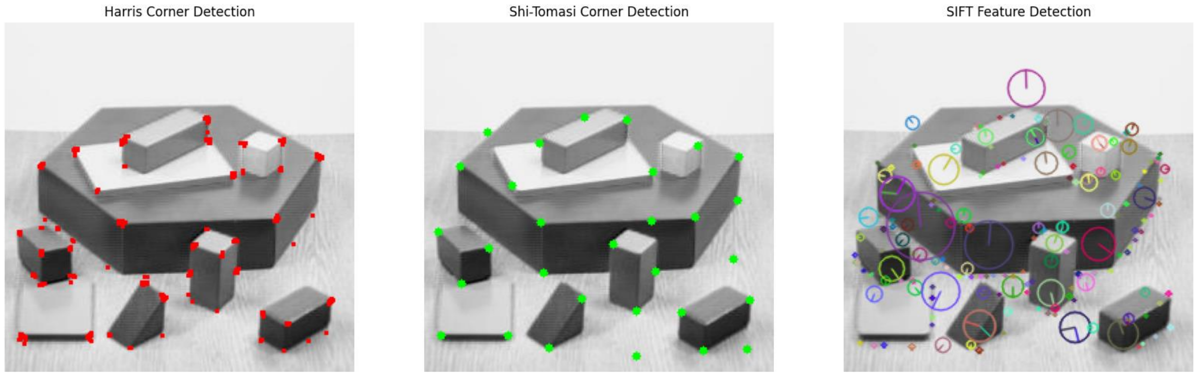
Code

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

# Load the image
image_path = '/content/blox.jpg'
image = cv2.imread(image_path, cv2.IMREAD_COLOR)
cropped_image = image[0:500, 0:500]

# Convert the image to grayscale
gray_image = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2GRAY)

# Step 2: Apply Harris Corner Detection
gray = np.float32(gray_image)
harris_corners = cv2.cornerHarris(gray, 2, 3, 0.04)
harris_corners = cv2.dilate(harris_corners, None)
harris_image = cropped_image.copy()
harris_image[harris_corners > 0.01 * harris_corners.max()] = [0, 0, 255]

# Step 3: Apply Shi-Tomasi Corner Detection
corners = cv2.goodFeaturesToTrack(gray_image, 100, 0.01, 31)
corners = np.int0(corners)
shi_tomasi_image = cropped_image.copy()
for corner in corners:
    x, y = corner.ravel()
    cv2.circle(shi_tomasi_image, (x, y), 3, (0, 255, 0), -1)

# Step 4: Apply SIFT Feature Detection
sift = cv2.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(gray_image, None)
sift_image = cv2.drawKeypoints(cropped_image, keypoints, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Display the results
plt.figure(figsize=(20, 20))

plt.subplot(1, 3, 1)
plt.imshow(cv2.cvtColor(harris_image, cv2.COLOR_BGR2RGB))
plt.title('Harris Corner Detection')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(cv2.cvtColor(shi_tomasi_image, cv2.COLOR_BGR2RGB))
plt.title('Shi-Tomasi Corner Detection')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(cv2.cvtColor(sift_image, cv2.COLOR_BGR2RGB))
plt.title('SIFT Feature Detection')
plt.axis('off')

plt.show()
```

Input image



Output



1. Harris Corner Detection

Methodology:

- Harris Corner Detection is based on the idea of analyzing the local gradient in an image to find points where the gradient changes significantly in all directions.
- It uses the eigenvalues of the second moment matrix (also known as the structure tensor) to measure corner strength.

Advantages:

- Simple and computationally efficient.
- Good for detecting corners where the intensity gradient changes in all directions.

Disadvantages:

- Sensitive to noise.
- Lacks scale invariance; corners detected at one scale may not be detected at another.
- Does not provide orientation information.

Typical Use Cases:

- Basic corner detection in images where computational efficiency is a priority.

2. Shi-Tomasi Corner Detection (Good Features to Track)

Methodology:

- An improvement over Harris Corner Detection.
- Uses the minimum eigenvalue of the second moment matrix rather than a combined function of the eigenvalues.
- Selects corners based on their quality measure (minimum eigenvalue).

Advantages:

- More selective and robust compared to Harris.
- Fewer false positives and better localization of corners.
- Still computationally efficient.

Disadvantages:

- Like Harris, it lacks scale and orientation information.
- Also sensitive to noise, though less so than Harris.

Typical Use Cases:

- Feature tracking (e.g., in optical flow algorithms).
- Applications requiring more accurate and reliable corner detection.

3. SIFT (Scale-Invariant Feature Transform)

Methodology:

- SIFT detects keypoints in scale space by finding extrema in the Difference of Gaussians (DoG) of the image at multiple scales.
- Each keypoint is assigned a descriptor that is invariant to scale and rotation.
- Uses histograms of gradient orientations around the keypoint to create a distinctive descriptor.

Advantages:

- Scale and rotation invariant, robust to changes in illumination and perspective.
- Provides detailed descriptors for each keypoint, making it suitable for matching across different images.
- High accuracy and robustness.

Disadvantages:

- Computationally more intensive compared to Harris and Shi-Tomasi.
- More complex to implement and understand.

Typical Use Cases:

- Object recognition.
- Image stitching and panorama creation.

Feature matching and tracking across different views.

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

Code

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

# Load the image
image_path = '/content/jigsaw.jpg'
image = cv2.imread(image_path, cv2.IMREAD_COLOR)

# Crop the region containing the puzzle piece
x, y, w, h = 1250, 1950, 500, 450
cropped_image = image[y:y+h, x:x+w]

# Step 2: Preprocess the image
gray_image = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2GRAY)
blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 0)
_, binary_image = cv2.threshold(blurred_image, 127, 255, cv2.THRESH_BINARY_INV)

# Use morphological operations to remove small noise
kernel = np.ones((5, 5), np.uint8)
cleaned_binary = cv2.morphologyEx(binary_image, cv2.MORPH_OPEN, kernel)

# Perform edge detection
edges = cv2.Canny(cleaned_binary, 50, 150, apertureSize=3)

# Find contours and filter out small contours
contours, _ = cv2.findContours(cleaned_binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
filtered_contours = [cnt for cnt in contours if cv2.contourArea(cnt) > 500]

# Smooth the contours by approximating them
smoothed_contours = []
for contour in filtered_contours:
    epsilon = 0.01 * cv2.arcLength(contour, True)
    approx_contour = cv2.approxPolyDP(contour, epsilon, True)
    smoothed_contours.append(approx_contour)

# Create an empty image to draw the smoothed contours
smoothed_image = np.zeros_like(edges)
cv2.drawContours(smoothed_image, smoothed_contours, -1, 255, thickness=cv2.FILLED)

# Apply Shi-Tomasi Corner Detection on the smoothed image
corners = cv2.goodFeaturesToTrack(smoothed_image, 4, 0.01, 180)
corners = np.int0(corners)

# Create a copy of the cropped image to draw corners on
corner_image = cropped_image.copy()
for corner in corners:
    x, y = corner.ravel()
    cv2.circle(corner_image, (x, y), 5, (255, 0, 0), -1)  # Red color

# Display the image with detected corners
plt.figure(figsize=(10, 10))
plt.imshow(cv2.cvtColor(corner_image, cv2.COLOR_BGR2RGB))
plt.title('Smoothed and Filtered Jigsaw Shape')
plt.axis('off')
plt.show()
```

Input Image

Original Image

Output

Smoothed and Filtered Jigsaw Shape

In this task, I have used several preprocessing techniques to achieve smoother edges to detect the given corners. Without preprocessing steps, the shi-tomasi corner detection identify incorrect corners.