National University of Computer & Emerging Sciences, Karachi
Artificial Intelligence-School of Computing
Fall 2024, Lab Manual - 06

Course Code (AI4002)	**Course: Computer Vision Lab**
Instructor(s):	**Sohail Ahmed**

Objectives:

- Wavelet Transformation
- Boundary Detection
- Hough Transformation
- Line Detection by Hough Transformation
- SIFT Feature Extraction

1. Introduction to Wavelet Transformation

Wavelet transformation is a mathematical technique used in signal processing, data analysis, and image processing. It allows you to analyze a signal or image in terms of its frequency components at different scales. Wavelet transforms provide a time-frequency analysis that is particularly useful for non-stationary signals. There are several types of wavelet transforms, with the most common being the Continuous Wavelet Transform (CWT) and the Discrete Wavelet Transform (DWT).

Continuous Wavelet Transform (CWT):

Mathematical Function: The CWT involves continuous wavelet functions that are dilated and translated across the signal. The wavelet function, often denoted as $\psi(t)$, is scaled by different frequencies and translated across time.

Wavelet Families: Different wavelet families can be used, such as the Morlet wavelet for analyzing oscillatory components in signals.

Applications: CWT is useful for analyzing time-frequency properties of continuous signals and is commonly used in applications like seismic signal analysis and image analysis.

Discrete Wavelet Transform (DWT):

Mathematical Function: The DWT involves discrete wavelet functions that are used to transform discrete signal data. The wavelet function is scaled and shifted across the signal in a discrete manner.

Wavelet Families: DWT uses various wavelet families, including Daubechies, Haar, and Symlet wavelets.

Applications: DWT is widely used in image compression, image denoising, and feature extraction. It's also used in signal compression and analysis.

Wavelet Packet Transform (WPT):

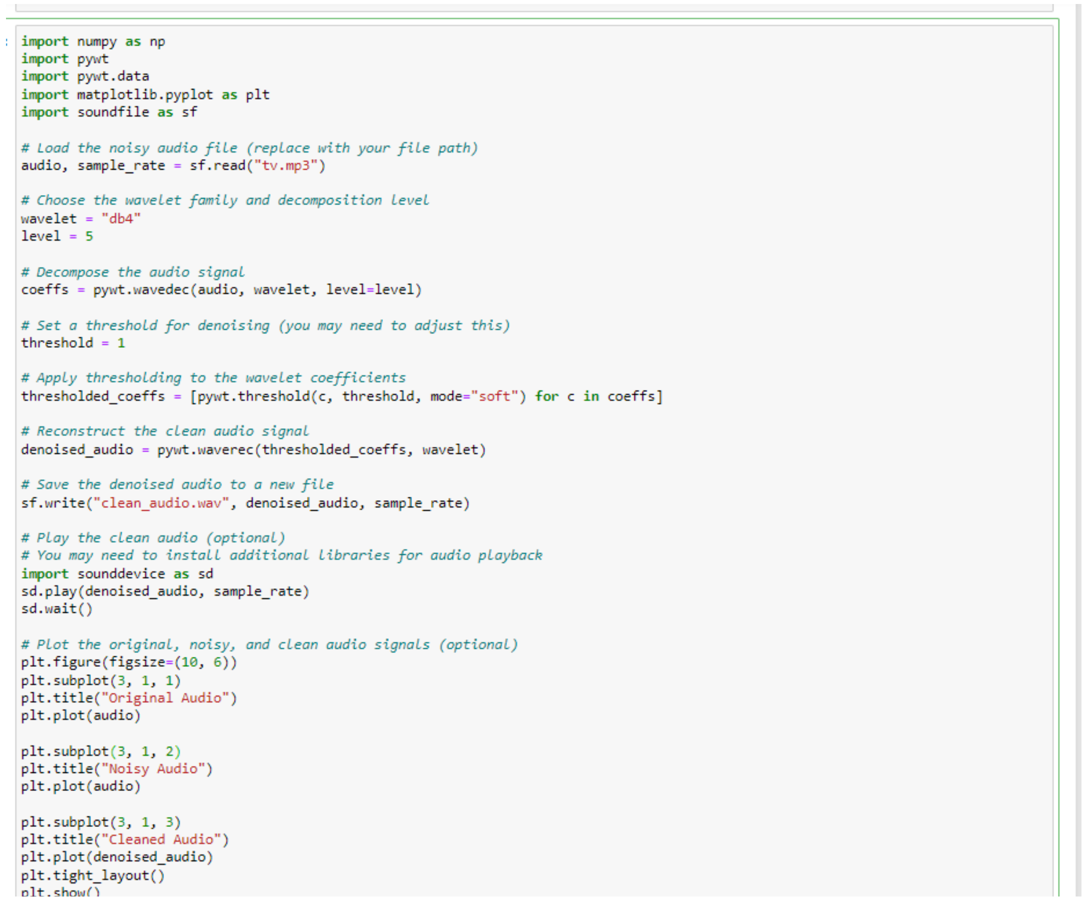
Mathematical Function: Wavelet packet transform is an extension of the DWT that allows for more flexible decomposition of signals. It decomposes signals into multiple subbands.

Applications: WPT can be used for analyzing signals with complex frequency content and for feature extraction in machine learning.

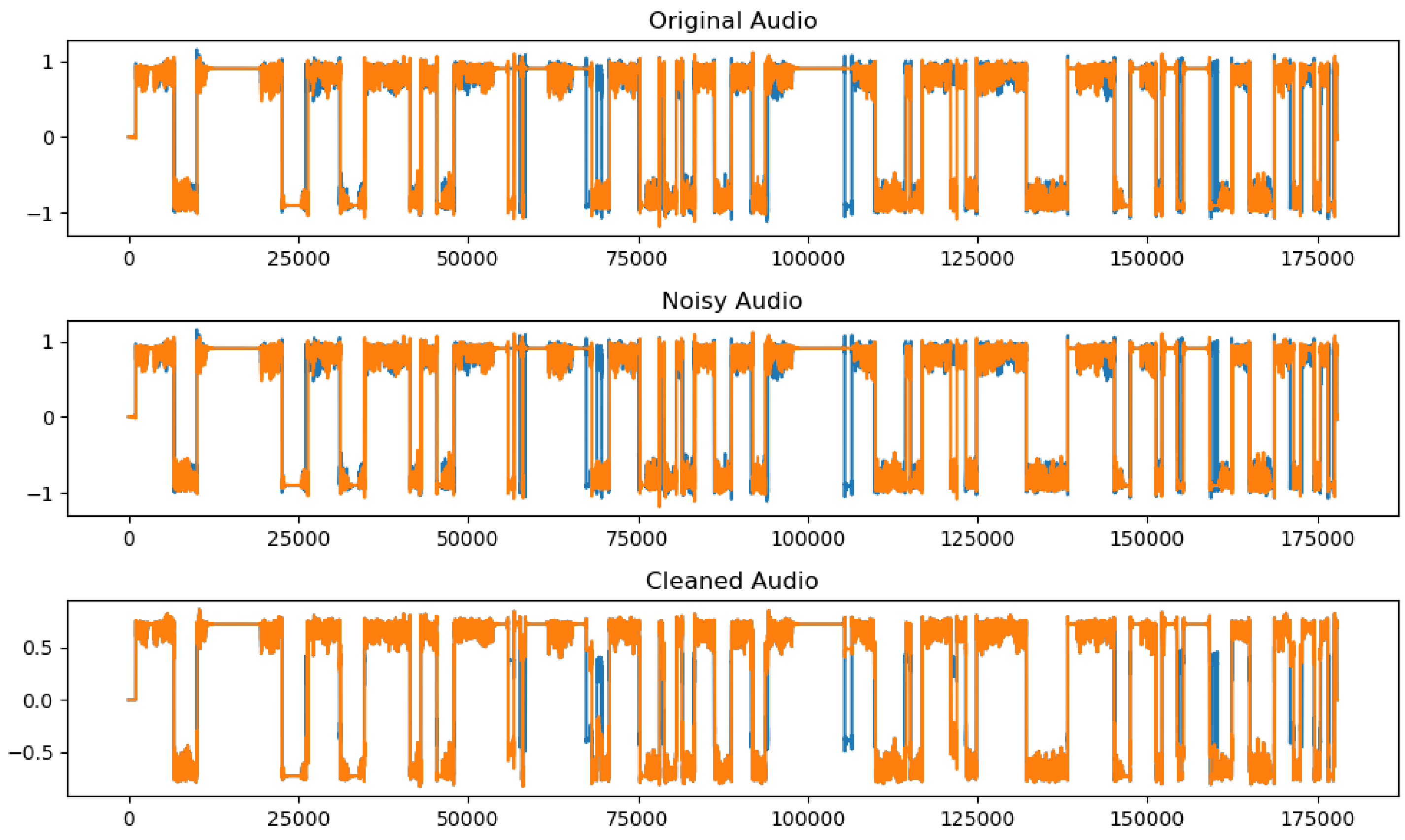
Inverse Wavelet Transform:

Mathematical Function: To reconstruct a signal or image from its wavelet coefficients, an inverse wavelet transform is applied. It uses the original data along with the transformed coefficients to reconstruct the original signal.

Applications: Inverse wavelet transforms are crucial for signal and image reconstruction in applications like image compression and de-noising.



```
import numpy as np
import pywt
import pywt.data
import matplotlib.pyplot as plt
import soundfile as sf

# Load the noisy audio file (replace with your file path)
audio, sample_rate = sf.read("tv.mp3")

# Choose the wavelet family and decomposition level
wavelet = "db4"
level = 5

# Decompose the audio signal
coeffs = pywt.wavedec(audio, wavelet, level=level)

# Set a threshold for denoising (you may need to adjust this)
threshold = 1

# Apply thresholding to the wavelet coefficients
thresholded_coeffs = [pywt.threshold(c, threshold, mode="soft") for c in coeffs]

# Reconstruct the clean audio signal
denoised_audio = pywt.waverec(thresholded_coeffs, wavelet)

# Save the denoised audio to a new file
sf.write("clean_audio.wav", denoised_audio, sample_rate)

# Play the clean audio (optional)
# You may need to install additional libraries for audio playback
import sounddevice as sd
sd.play(denoised_audio, sample_rate)
sd.wait()

# Plot the original, noisy, and clean audio signals (optional)
plt.figure(figsize=(10, 6))
plt.subplot(3, 1, 1)
plt.title("Original Audio")
plt.plot(audio)

plt.subplot(3, 1, 2)
plt.title("Noisy Audio")
plt.plot(audio)

plt.subplot(3, 1, 3)
plt.title("Cleaned Audio")
plt.plot(denoised_audio)
plt.tight_layout()
plt.show()
```



2. Boundary Detection

Boundary detection, often referred to as edge detection, is a fundamental concept in computer vision. It plays a crucial role in identifying the boundaries of objects within an image. These boundaries represent significant changes in intensity or color and are essential for various computer vision tasks, such as object recognition, image segmentation, and feature extraction.

Edges in an image are critical because they contain important information about the shapes and structures present. By detecting edges, computer vision algorithms can focus on these key regions, making it easier to analyze and interpret the content of an image.

Sobel Edge Detection:

Sobel edge detection is a basic gradient-based method. It applies convolution with two 3x3 kernels to calculate the gradient in the horizontal and vertical directions. The magnitude of the gradient represents the strength of the edge, and its direction indicates the edge's orientation.

Canny Edge Detection:

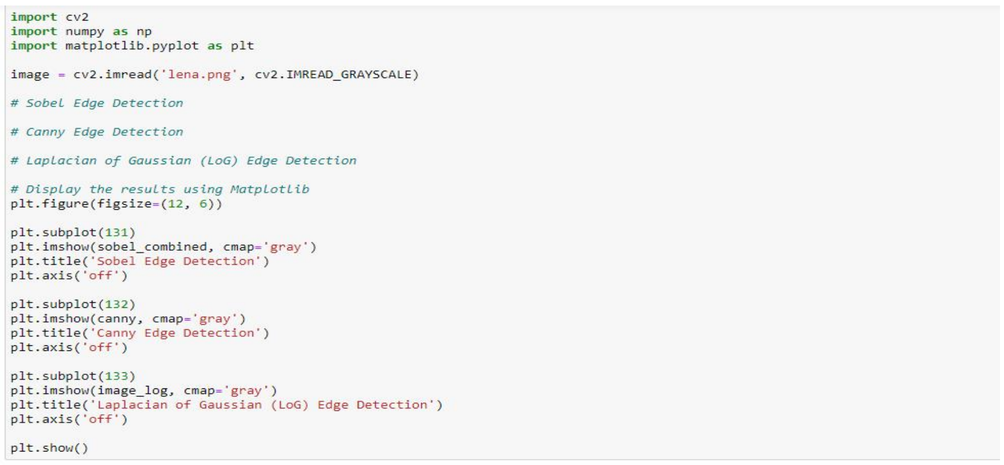
Canny edge detection is a more sophisticated method that consists of multiple steps

- Gaussian smoothing: Reduces noise in the image.
- Gradient calculation: Calculates the gradient magnitude and direction.
- Non-maximum suppression: Eliminates weak edge pixels.
- Hysteresis thresholding: Tracks edges by linking strong edge pixels.

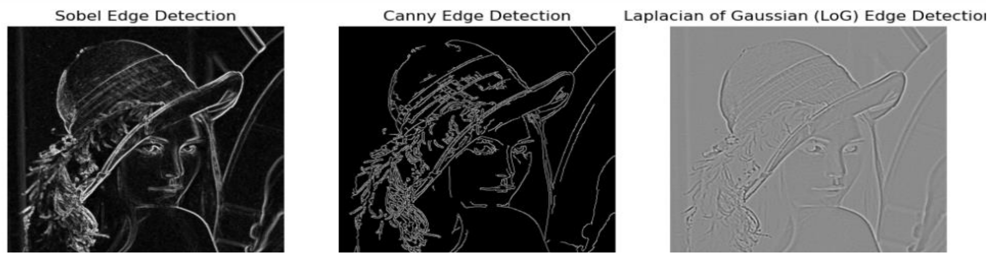
The Canny detector is known for its high accuracy and ability to detect thin and continuous edges.

Laplacian of Gaussian (LoG):

The LoG operator is used to detect edges by first applying a Gaussian smoothing filter to the image and then computing the Laplacian of the smoothed image. The zero-crossings of the resulting image indicate the edges. The LoG operator is effective at detecting edges with varying scales.



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

image = cv2.imread('lena.png', cv2.IMREAD_GRAYSCALE)

# Sobel Edge Detection

# Canny Edge Detection

# Laplacian of Gaussian (LoG) Edge Detection

# Display the results using Matplotlib
plt.figure(figsize=(12, 6))

plt.subplot(131)
plt.imshow(sobel_combined, cmap='gray')
plt.title('Sobel Edge Detection')
plt.axis('off')

plt.subplot(132)
plt.imshow(canny, cmap='gray')
plt.title('Canny Edge Detection')
plt.axis('off')

plt.subplot(133)
plt.imshow(image_log, cmap='gray')
plt.title('Laplacian of Gaussian (LoG) Edge Detection')
plt.axis('off')

plt.show()
```



3. Introduction to Hough Transformation

The Hough Transform is a fundamental technique in computer vision and image processing used to detect simple shapes within an image, most commonly lines and circles. It was first introduced by Paul Hough in 1962 for detecting lines in binary images, and later extended to detect other shapes like circles and ellipses. The Hough Transform is particularly useful when traditional methods like gradient-based edge detection are insufficient for detecting specific shapes. It's widely employed in various computer vision applications, such as image analysis, object recognition, and feature extraction.

The Hough Transform works by transforming the spatial domain of an image into a parameter space, where each pixel in the image corresponds to a curve or a point in this parameter space. The primary purpose of this transformation is to identify patterns that are represented as lines or curves in the parameter space.

Hough Transform for Line Detection:

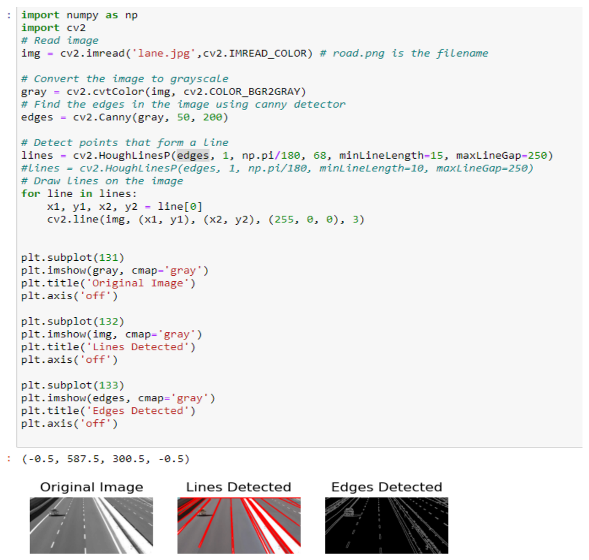
Parametric Representation of Lines- The Hough Transform starts with the parametric representation of lines, where a line in Cartesian coordinates (x, y) is represented as **$y = mx + b$.**
Here, 'm' is the slope of the line, and 'b' is the y-intercept

- **Parameter Space-** In the Hough Transform, we transform this representation into a parameter space with 'm' and 'b' as axes. Each line in the image corresponds to a point in this parameter space.
- **Voting-** For each edge pixel (typically obtained from edge detection methods like Canny), we calculate the 'm' and 'b' values that could generate the line passing through that pixel. This is essentially a voting process where we increment the corresponding (m, b) cell in the parameter space.
- **Peak Detection-** After processing all edge pixels, the parameter space is examined to find peaks. These peaks represent the lines in the image.
- **Back-Transformation-** Once the lines are detected in the parameter space, they can be back-transformed into the image space, revealing the position and orientation of the lines.

Hough Transform for Circle Detection:

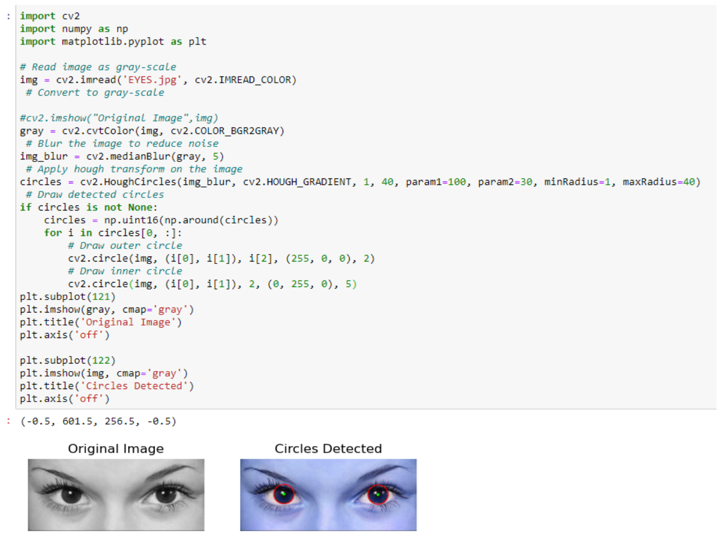
Parametric Representation of Circles: In the case of circle detection, a circle in Cartesian coordinates (x, y) is represented as **$(x - a)^2 + (y - b)^2 = r^2$.**
Here, (a, b) represents the center of the circle, and 'r' is its radius.

- **Parameter Space-** The parameter space for circle detection has three dimensions, corresponding to 'a,' 'b,' and 'r.'
- **Voting-** Similar to line detection, we vote for potential circle centers (a, b) and radii (r) based on edge pixel information in the image.
- **Peak Detection-** After processing all edge pixels, peaks in the parameter space correspond to potential circle centers and radii.
- **Back-Transformation-** Detected circles in the parameter space can be back-transformed into the image space, revealing the position and size of the circles.



```
import numpy as np
import cv2
# Read image
img = cv2.imread('lane.jpg',cv2.IMREAD_COLOR) # road.png is the filename

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Find the edges in the image using canny detector
edges = cv2.Canny(gray, 50, 200)

# Detect points that form a line
lines = cv2.HoughLinesP(edges, 1, np.pi/180, 68, minLineLength=15, maxLineGap=250)
#lines = cv2.HoughLinesP(edges, 1, np.pi/180, minLineLength=10, maxLineGap=250)
# Draw lines on the image
for line in lines:
    x1, y1, x2, y2 = line[0]
    cv2.line(img, (x1, y1), (x2, y2), (255, 0, 0), 3)


plt.subplot(131)
plt.imshow(gray, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(132)
plt.imshow(img, cmap='gray')
plt.title('Lines Detected')
plt.axis('off')

plt.subplot(133)
plt.imshow(edges, cmap='gray')
plt.title('Edges Detected')
plt.axis('off')
```

```
(-0.5, 587.5, 300.5, -0.5)
```



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

# Read image as gray-scale
img = cv2.imread('EYES.jpg', cv2.IMREAD_COLOR)
 # Convert to gray-scale

#cv2.imshow("Original Image",img)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
 # Blur the image to reduce noise
img_blur = cv2.medianBlur(gray, 5)
 # Apply hough transform on the image
circles = cv2.HoughCircles(img_blur, cv2.HOUGH_GRADIENT, 1, 40, param1=100, param2=30, minRadius=1, maxRadius=40)
 # Draw detected circles
if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0, :]:
        # Draw outer circle
        cv2.circle(img, (i[0], i[1]), i[2], (255, 0, 0), 2)
        # Draw inner circle
        cv2.circle(img, (i[0], i[1]), 2, (0, 255, 0), 5)
plt.subplot(121)
plt.imshow(gray, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(122)
plt.imshow(img, cmap='gray')
plt.title('Circles Detected')
plt.axis('off')
```

```
(-0.5, 601.5, 256.5, -0.5)
```

4. SIFT Feature Extraction

SIFT (Scale-Invariant Feature Transform) is a powerful computer vision technique used for feature extraction and matching in images. It is particularly useful for tasks such as object recognition, image stitching, and tracking.

SIFT Feature Extraction

Scale-Space Extrema Detection

SIFT starts by detecting keypoint candidates at multiple scales within the image. This is achieved by applying a Difference of Gaussians (DoG) operator to identify local extrema in the scale space. The DoG is calculated by subtracting two Gaussian-blurred versions of the image.

Keypoint Localization:

For each candidate keypoint, SIFT refines the location by fitting a 3D quadratic function to the nearby data points in the scale space. This step helps to accurately localize the keypoints.

Orientation Assignment:

To make the keypoints invariant to rotation, SIFT assigns an orientation to each keypoint based on the dominant gradient direction in its local neighborhood. This results in a canonical orientation for each keypoint.

Keypoint Descriptor Generation:

Around each keypoint, SIFT constructs a descriptor that characterizes the local image region. The descriptor is created by considering the gradient magnitudes and orientations in a specific pattern of sub-regions surrounding the keypoint. This pattern is divided into smaller bins, and histograms of gradient orientations are computed for each sub-region. The concatenated histograms form the keypoint descriptor.

Keypoint Matching:

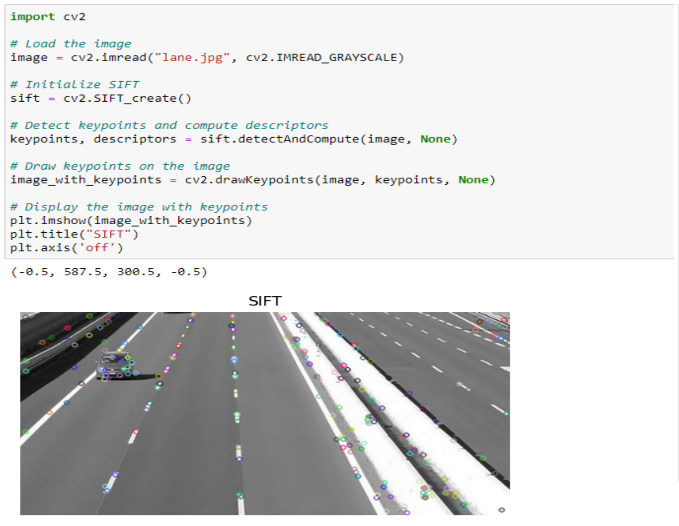
Once the descriptors are computed for keypoints in multiple images, these descriptors can be used to match keypoints between images. A common technique for matching involves calculating the Euclidean distance or other similarity measures between the descriptors of keypoints in different images.

Outlier Rejection:

Keypoint matches may contain some outliers due to variations in lighting, viewpoint changes, or noise. Techniques like RANSAC (Random Sample Consensus) can be employed to robustly estimate transformation models (e.g., affine or perspective transformations) and reject outliers.

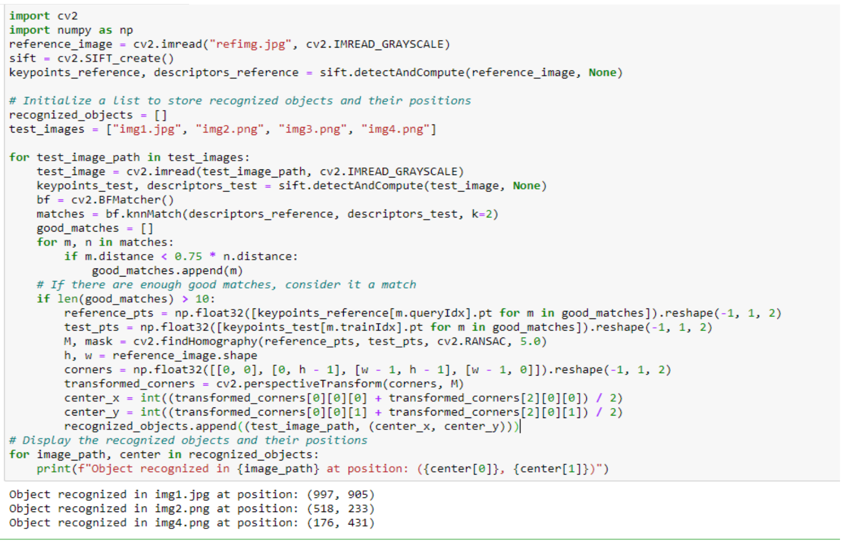
Final Matching and Homography Estimation:

After outlier rejection, the remaining keypoint matches are used to estimate the transformation (homography) between the images. This transformation can be used for image stitching, object recognition, or other applications.



```python
import cv2

# Load the image
image = cv2.imread("lane.jpg", cv2.IMREAD_GRAYSCALE)

# Initialize SIFT
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors
keypoints, descriptors = sift.detectAndCompute(image, None)

# Draw keypoints on the image
image_with_keypoints = cv2.drawKeypoints(image, keypoints, None)

# Display the image with keypoints
plt.imshow(image_with_keypoints)
plt.title("SIFT")
plt.axis('off')
```

```
(-0.5, 587.5, 300.5, -0.5)
```

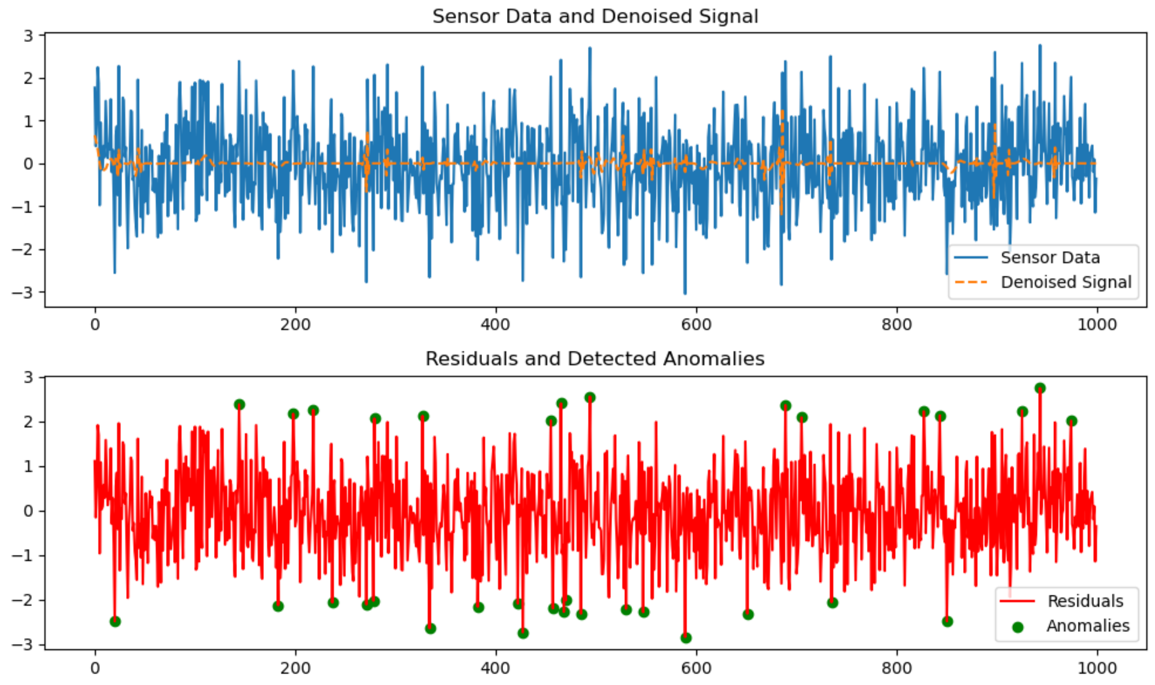


```python
import cv2
import numpy as np
reference_image = cv2.imread("refimg.jpg", cv2.IMREAD_GRAYSCALE)
sift = cv2.SIFT_create()
keypoints_reference, descriptors_reference = sift.detectAndCompute(reference_image, None)

# Initialize a list to store recognized objects and their positions
recognized_objects = []
test_images = ["img1.jpg", "img2.png", "img3.png", "img4.png"]

for test_image_path in test_images:
    test_image = cv2.imread(test_image_path, cv2.IMREAD_GRAYSCALE)
    keypoints_test, descriptors_test = sift.detectAndCompute(test_image, None)
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(descriptors_reference, descriptors_test, k=2)
    good_matches = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good_matches.append(m)
    # If there are enough good matches, consider it a match
    if len(good_matches) > 10:
        reference_pts = np.float32([keypoints_reference[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
        test_pts = np.float32([keypoints_test[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)
        M, mask = cv2.findHomography(reference_pts, test_pts, cv2.RANSAC, 5.0)
        h, w = reference_image.shape
        corners = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2)
        transformed_corners = cv2.perspectiveTransform(corners, M)
        center_x = int((transformed_corners[0][0][0] + transformed_corners[2][0][0]) / 2)
        center_y = int((transformed_corners[0][0][1] + transformed_corners[2][0][1]) / 2)
        recognized_objects.append((test_image_path, (center_x, center_y)))
# Display the recognized objects and their positions
for image_path, center in recognized_objects:
    print(f"Object recognized in {image_path} at position: ({center[0]}, {center[1]})")
```

```
Object recognized in img1.jpg at position: (997, 905)
Object recognized in img2.png at position: (518, 233)
Object recognized in img4.png at position: (176, 431)
```

Lab Tasks (Lab Timing)

1. Complete the above codes where required.
2. **Computer Screen Detection in a Computer Lab**
   You are working on a computer vision project for monitoring computer lab usage and ensuring the availability of computer screens. Your goal is to implement screen detection using the Hough Line Transformation. The lab contains rows of computers, and you need to identify the boundaries of computer screens to monitor their status (on or off) and detect any anomalies, such as missing screens.
3. **Asset Tracking in a Computer Lab Using SIFT**
   You are responsible for managing computer assets in a busy computing lab. One of the challenges you face is keeping track of the various computers and their components within the lab. To streamline this process, you want to implement a computer vision solution using the Scale-Invariant Feature Transform (SIFT). Your goal is to automatically recognize and identify individual computer systems and their unique components, such as monitors and keyboards, to maintain an up-to-date inventory
4. **Anomaly Detection in Sensor Data Using Wavelet Transformation**
   You are tasked with developing a system to monitor sensor data from industrial machines and identify anomalies in real-time. Anomalies may indicate potential machine malfunctions or issues. Your approach is to use the wavelet transformation to analyze the sensor data and detect abnormal patterns. Your goal is to create a simple anomaly detection system using a sample sensor dataset and wavelet analysis.
   Output Expected:



Lab Tasks

1. **Object Recognition (Using Video):**

You are working on an image processing project, and your task is to implement object recognition using the Scale-Invariant Feature Transform (SIFT) algorithm. You have a reference image of the object you want to recognize and a set of test images. Your goal is to identify and locate the object in each test image, even if it appears at different scales, orientations, or under partial occlusion. You will use SIFT features for object recognition and draw bounding boxes around the recognized objects in the test images.

2. **Your Panoramic Image**

You are working on a project where you need to create a panoramic image by stitching multiple overlapping images together. The task involves taking a set of images captured from a camera while panning, and then using the SIFT algorithm to find key points and match them between adjacent images. Your goal is to automatically align and stitch the images to create a single panoramic image.

3. **Lane detection using the Hough Line Transformation**

You are working on an autonomous vehicle project, and one of the critical tasks is to detect lane markings on the road to ensure safe and accurate navigation. Your goal is to implement lane detection using the Hough Line Transformation. You have a set of images captured from the vehicle's camera. Your task is to identify and draw the detected lane lines on these images to assist in lane-keeping and control algorithms.

4. **Coins Detection and Counting**

You are working on a computer vision project that involves the identification and counting of coins in images. Your task is to implement coin detection using the Hough Circle Transformation. You have a set of images containing coins of various sizes and positions. Your goal is to automatically identify and count the coins in these images.

5. **Smart Security System**

You are working on a security system for an organization, and one of the key tasks is to identify unauthorized objects that are placed near sensitive areas. To achieve this, you need to implement boundary detection using computer vision. Your goal is to detect the boundaries of objects that are within a predefined security zone in a real-time video stream. When an unauthorized object enters this zone, an alarm should be triggered.