VR Assignment 1

Akash Chaudhari
MT2024012

1 Introduction

This document presents the implementation of image processing techniques including edge detection, image segmentation, coin counting, and panorama stitching. The techniques are implemented using Python and OpenCV.

2 Processing Steps

2.1 Edge Detection

1. **Load the Image**

Figure 1: Original Image

2. **Convert to Grayscale**
    - Edge detection algorithms work more efficiently on grayscale images.
3. **Apply Gaussian Blur**

- Gaussian Blur smoothens the image, reducing noise and unnecessary details.
- It prevents false edge detection by removing small variations and minor artifacts.
- The kernel size (e.g., (13,13)) is chosen based on the level of noise present.

4. **Normalize the Image**
    - Normalization scales pixel intensities to a consistent range (0-255).
    - This enhances contrast, making edges more distinct.
5. **Perform Sobel Edge Detection**
    - The Sobel operator detects gradients in the image in both horizontal and vertical directions.
    - It highlights areas where intensity changes rapidly, indicating the presence of edges.

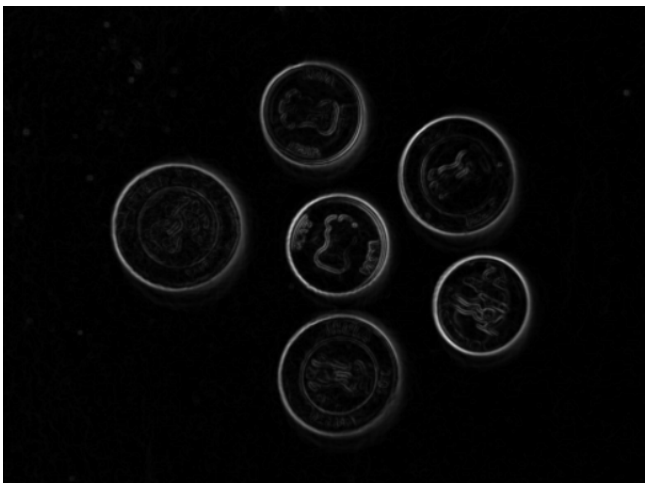

Figure 2: Edge detection result using Sobel

6. **Apply Thresholding**
    - Thresholding converts the grayscale image into a binary image.
    - It removes weak edges and enhances strong edges by setting a fixed intensity limit.
7. **Apply Canny Edge Detection**

- Canny Edge detection is applied on the threshold image to get edges.

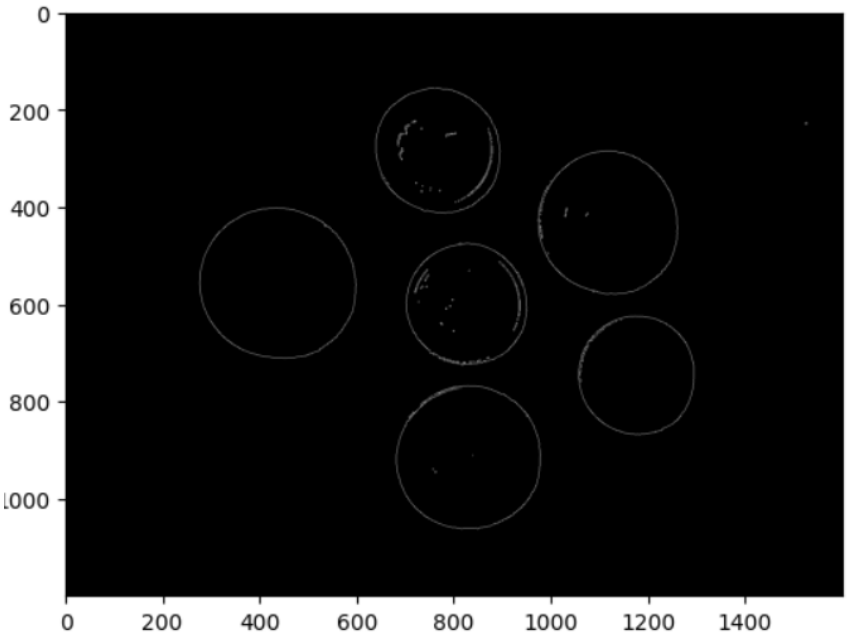

Figure 3: Edge detection result using Canny

8. **Blend Grayscale and Edge Image**

   - Combining the original grayscale image with the detected edges improves visualization.

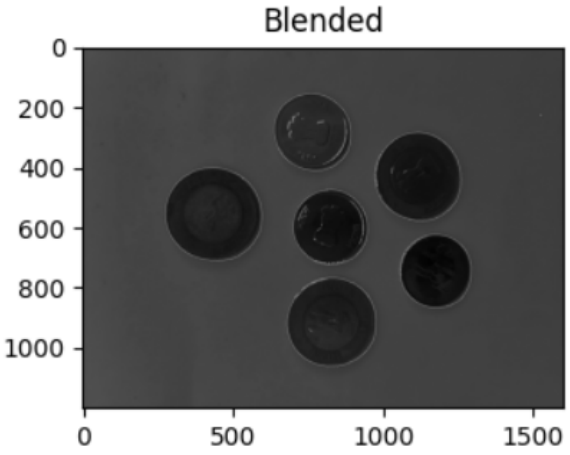

Figure 4: Combined Image

2.2 Image Segmentation

1. **Load the Image**
2. **Convert to Grayscale**
   - Converts the image to a single-channel grayscale representation.
3. **Normalize the Image**
   - Scales pixel values to a standard range (0-255).
   - Improves contrast and ensures better segmentation results.
4. **Apply Gaussian Blur**
   - Smoothens the image to reduce noise.
   - Prevents false segmentations by removing small intensity variations.
5. **Apply Thresholding**
   - Converts the grayscale image into a binary format.
   - Segments the image by setting pixel intensities to either black or white.

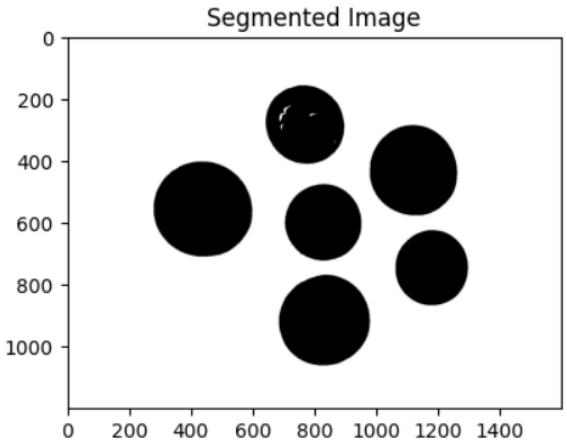

Figure 5: Segmented Image

2.3 Coin Counting

1. **Load the Image and Resize it**
2. **Convert to Grayscale**
   - Since color is not required for edge and contour detection, converting to grayscale simplifies processing.
3. **Normalize Pixel Intensity**
   - Normalization adjusts pixel values to a specific range (e.g., 0-255), enhancing contrast.
   - It helps to reduce variations caused by lighting conditions, making features more distinguishable.
4. **Apply Gaussian Blur**
   - Blurring helps to smooth the image and reduce noise, making coin edges more distinct.
   - It ensures small irrelevant details do not interfere with contour detection.
5. **Apply Thresholding and Invert It**
   - Thresholding converts the grayscale image into a binary image, where pixels are either black (0) or white (255).
   - This simplifies contour detection by clearly distinguishing objects (coins) from the background.

- Inverting the image ensures that the coins appear as white objects against a black background, which is required for contour detection.

6. **Detect Contours**
   - Contours are the outlines of objects, and detecting them allows us to identify the coins in the image.
   - The `cv2.findContours()` function is used to extract the shapes in the thresholded image.
7. **Filter Contours Based on Size**
   - Not all detected contours belong to coins; some may be noise or small unwanted objects.
   - Filtering ensures that only contours with a reasonable area (size) are considered as valid coins.
8. **Count and Highlight Coins**
   - The remaining filtered contours represent the actual coins in the image.
   - Each detected coin is counted, and a bounding box is drawn around it to highlight it.

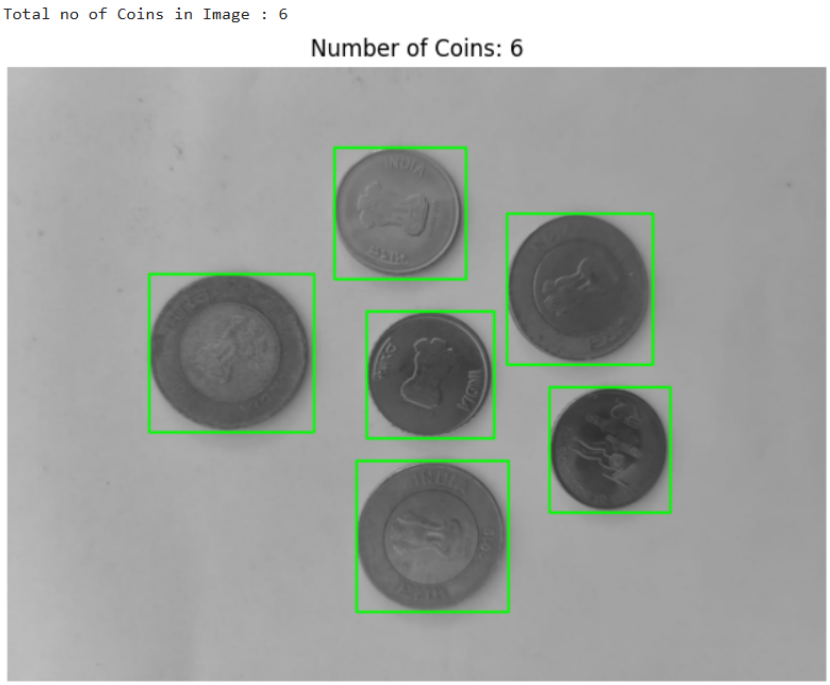

Figure 6: Coin Counting

2.4 Key Point Detection and Matching using SIFT

1. **Convert images to grayscale**
2. **Initialize SIFT detector:** The *Scale-Invariant Feature Transform (SIFT)* algorithm is used to detect distinctive key points in the image that remain stable across different scales, rotations, and lighting conditions.
3. **Compute key points and descriptors:** SIFT detects key points in the image and generates corresponding descriptors.
4. **Initialize BFMatcher:** A *Brute-Force Matcher (BFMatcher)* is used to compare feature descriptors between two images and find the closest matches.
5. **Find matches using KNN Matching:** The K-Nearest Neighbors (KNN) algorithm is applied to find the best matches for each key point descriptor by selecting the two nearest matches from the second image.
6. **Apply Lowe's ratio test:** To filter out poor matches, Lowe's ratio test is used. It retains matches where the closest match is significantly better than the second-closest match, reducing false correspondences.

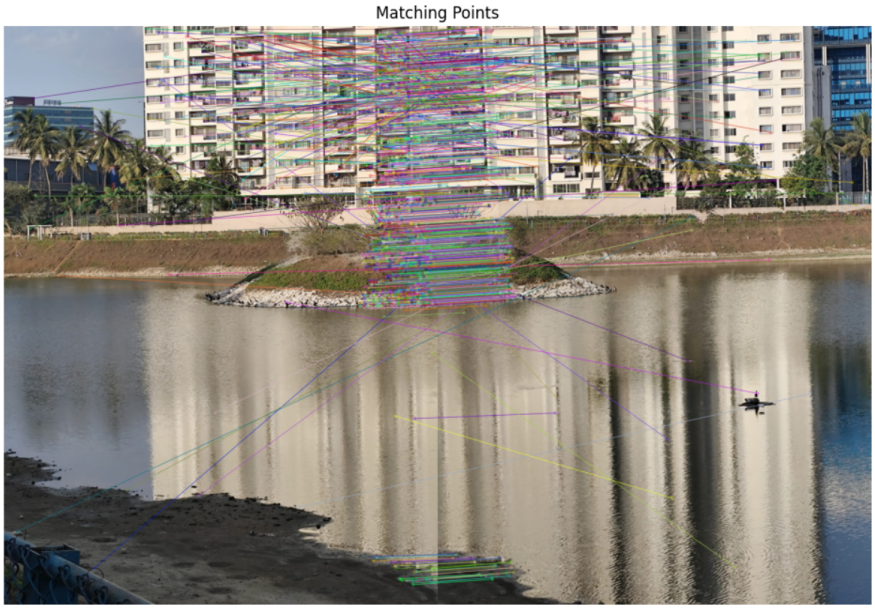

Figure 7: Matching Points using SIFT

2.5 Image Stitching

After Keypoint Detection Follow the following steps

1. **Find homography and warp the image:** Using the good matches obtained, a transformation matrix (*homography*) is computed using the *RANSAC* (Random Sample Consensus) algorithm. This transformation aligns the two images by mapping key points from one image onto the corresponding key points in the second image.

2. **Merge and blend images:** The first image is warped onto the second image's perspective using the computed homography matrix. The images are then merged, and blending techniques are applied to smooth the transition between overlapping regions.

Figure 8: Panorama Image

3. **Another Approach** Use 'cv2.Stitcher_create()' to create panorama Images.