Visual Recognition

Image Segmentation and Stitching

Name: Shannon Muthanna I B

Roll no: IMT2022552

Repository Link

https://github.com/Shannon2004/VR_Assignment1_Shannon_IMT2022552

Introduction

The following assignment majorly focuses on two main tasks. Firstly, edge detection and image segmentation. Secondly, panoramic image stitching of a set of images.

Dataset

The images for Task 1 of the assignment consist of coins in different orientations and numbers, while the images for Task 2 of the assignment consist of a set of images that need to be stitched to obtain a final panoramic image.

Task 1

Edge Detection

In the code, multiple edge detection techniques were tested to determine the most effective method for detecting coin boundaries. Initially, the **Sobel operator** was applied using `cv2.Sobel` with different kernel sizes. A **kernel size of 3 (ksize=3)** was used for edge enhancement while keeping noise manageable. The operator was

applied in both X (`dx=1, dy=0`) and Y (`dx=0, dy=1`) directions separately, followed by computing the gradient magnitude. However, the edges detected were not continuous, and noise levels were high, making it unsuitable for coin detection. This also had a lot of false edges.

Next, the **Laplacian operator** was applied using `cv2.Laplacian`, which calculates the second-order derivative of the image. It was tested with different kernel sizes. A kernel size of **3 (ksize=3)** produced sharp edges but also highlighted unwanted textures in the background. Increasing the kernel size to 5 helped smoothen the edges but introduced excessive blurring, leading to loss of finer details. The Laplacian method was found to be ineffective due to its sensitivity to noise.

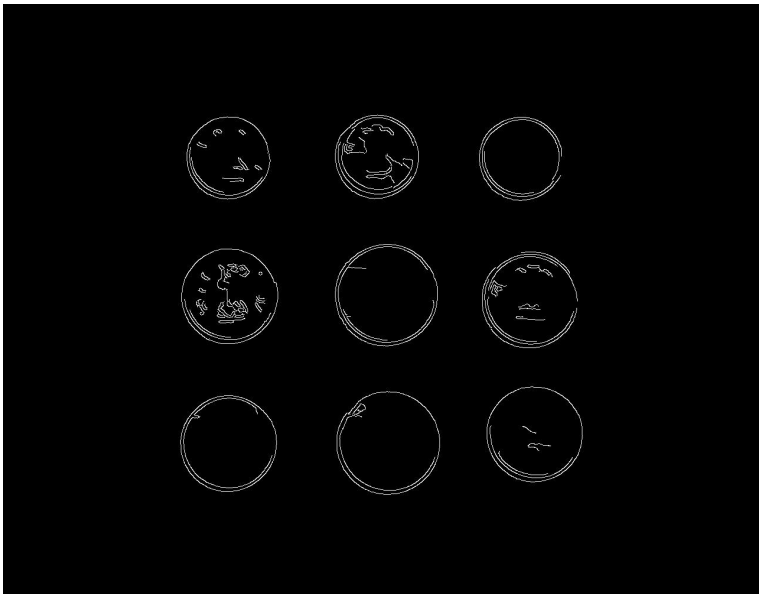
Finally, the **Canny Edge Detector (`cv2.Canny`)** was implemented, which provided the best results. The function was tested with multiple threshold values. After trial and erro, an **optimal lower threshold of 50 and an upper threshold of 150** (`cv2.Canny(image, 50, 150)`) was selected. Before applying Canny, **Gaussian Blurring (`cv2.GaussianBlur`)** with a kernel size of **(5,5)** and a standard deviation of 0 was used to reduce noise and smoothen the image. The Canny algorithm performed better than Sobel and Laplacian due to its **gradient magnitude computation, Non-Maxima Suppression, and Hysteresis Thresholding**, which ensured that only strong edges were retained while weak edges were suppressed. Unlike other methods,

Canny provided well-defined edges with minimal noise, making it the preferred choice for further processing.

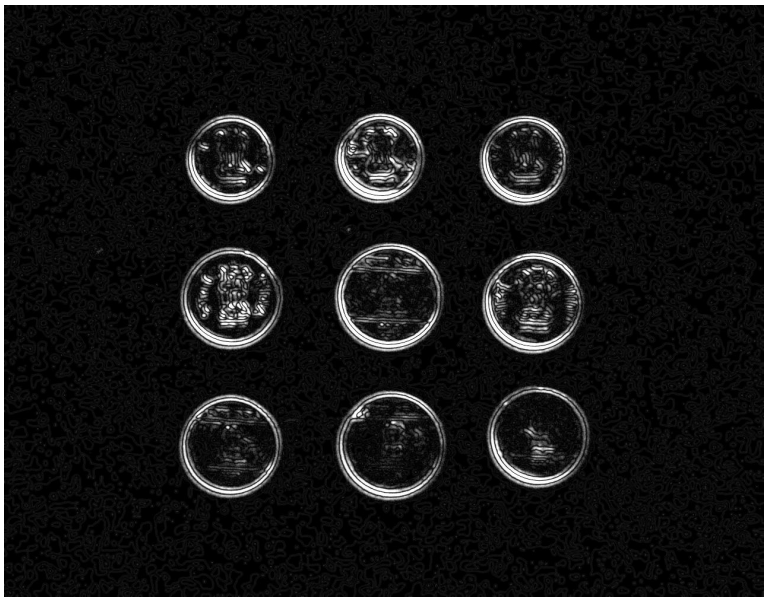
Different edge detectors were passed with different Gaussian LPFs characterized according to the effectiveness of the algorithm. Contours are drawn according to the results of the Canny Detector around the original image.

Original Image

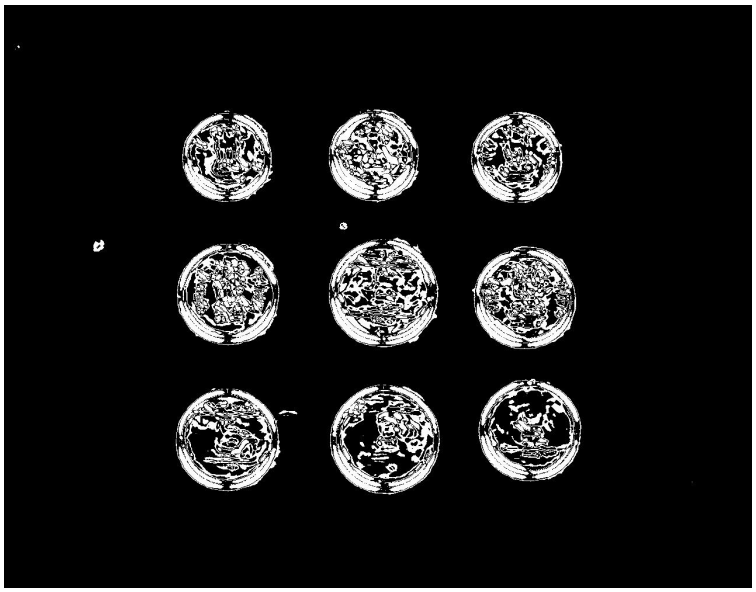
Canny Edge Detector



Laplacian Edge Detector



Sobel Edge Detector



Outlined Image

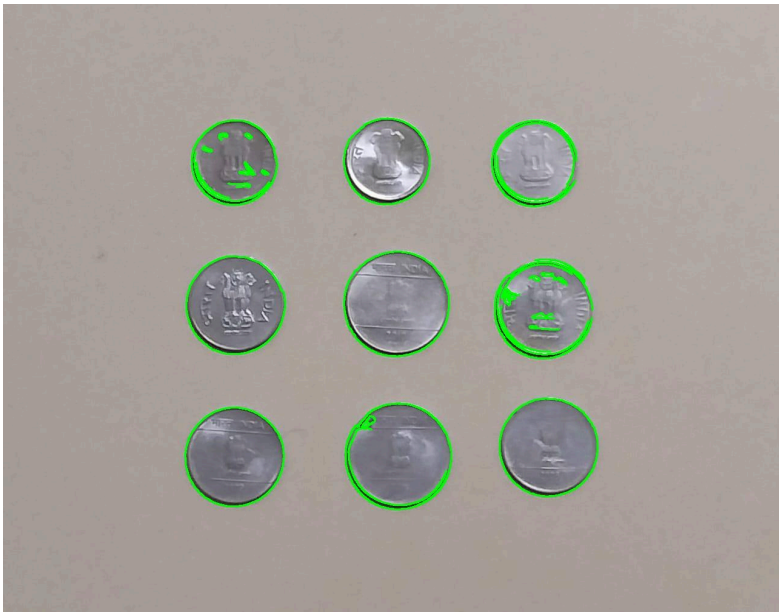


Image Segmentation

The segmentation process in the code was designed to accurately separate coins from the background. First, the input image was converted to **grayscale (cv2.cvtColor)** to remove color variations. Initially, **Otsu's thresholding** was attempted, but due to uneven lighting and reflections on the coins, it failed to consistently separate all coins from the background. To overcome this, **Adaptive Thresholding (cv2.adaptiveThreshold)** was implemented, which calculates the threshold dynamically for different parts of the image. This significantly improved segmentation, making the coin regions more distinct.

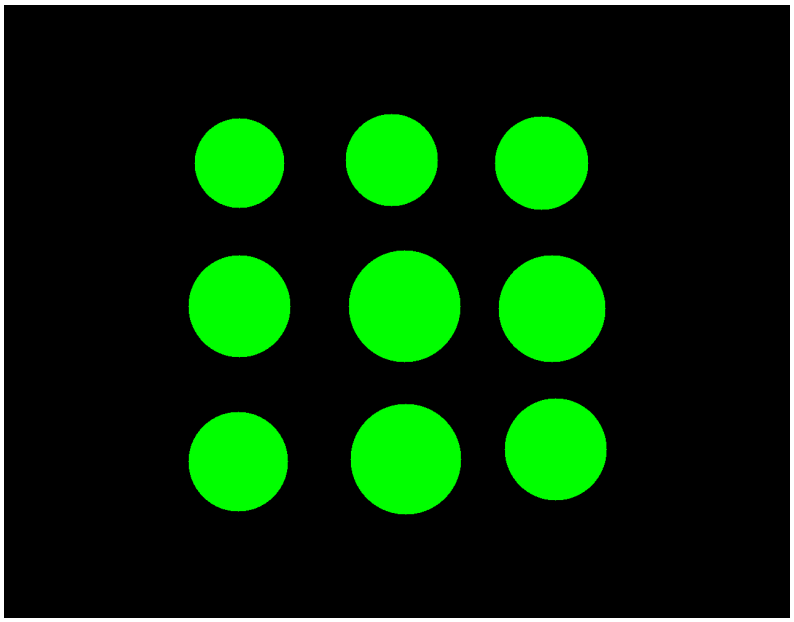
After thresholding, **Morphological Transformations** were applied to enhance the segmented output. **Morphological Closing (cv2.morphologyEx with cv2.MORPH_CLOSE)** was used with a **5×5 structuring element**, effectively filling small gaps within the coin contours. Without this step, some coins had incomplete boundaries due to reflections or faint edges. To further refine the segmentation, **Morphological Opening (cv2.morphologyEx with cv2.MORPH_OPEN)** was also tested to remove small noise, but it was later discarded as it sometimes eroded coin edges.

The segmented regions were then passed through **Contour Detection (cv2.findContours)** to extract individual coin boundaries. In the code, contours were filtered based on **area constraints** to remove unwanted small objects that were falsely detected due to noise. Additionally, a **circularity check** was implemented using the formula **$(4\pi \times \text{Area}) / (\text{Perimeter}^2)$** to ensure that only round objects (coins) were retained.

This approach eliminated false positives such as reflections or non-coin objects.

Various structuring element sizes were tested for morphological operations, and it was found that a **(5,5) kernel** provided the best balance between noise removal and edge preservation. Similarly, different adaptive thresholding block sizes and constant values were experimented with to optimize coin visibility. Through multiple trials, the final segmentation method effectively extracted and isolated each coin with minimal background interference.

Segmented Coins



One of the Segmented Coin

Counting the number of coins

Once segmentation was complete, counting was performed using **Contour Detection**. The final segmented image was passed to **cv2.findContours**, and the total number of valid contours was counted using `len(contours)`. To ensure accuracy, contours were filtered based on area, eliminating extremely small or large objects that could be noise or merged coins. This provided an accurate count of individual coins in the image.

```
(vr_assignment_1) shannon@shannon-Inspiron-14-5430:~/Sem_6/VR/Assignment1$ python3 coins1.py
Segmented coins saved in 'Output_Q1_p2/Coins'
Masked segmented image saved at 'Output_Q1_p2/segmented_coins.png'
Detected 9 coins.
Number of coins detected: 9
(vr_assignment_1) shannon@shannon-Inspiron-14-5430:~/Sem_6/VR/Assignment1$
```

Task 2

Key points detection

Key points detection is a crucial step in image stitching as it helps identify distinct and repeatable features across overlapping images. In this implementation, the **Scale-Invariant Feature Transform (SIFT)** algorithm is used to extract key points from the input images. SIFT is a widely used feature detection method that identifies important points in an image that remain consistent despite variations in scale, rotation, and illumination.

The SIFT detector first converts the image to grayscale and then applies a **Difference of Gaussians (DoG)** operation to detect potential key points. These key points are then refined by eliminating weak features using a contrast threshold. Each detected key point is assigned an orientation based on local image gradients, ensuring rotation invariance. Finally, a **128-dimensional descriptor** is created for each key point, representing its local texture information, making it robust for matching across images.

In this implementation, the following parameters are used while creating the SIFT object:

- **nfeatures=2000**: Limits the number of key points to 2000 per image to balance accuracy and computational efficiency.
- **contrastThreshold=0.04**: Sets the minimum contrast required for a key point to be considered, filtering out less significant features.

After extracting key points from each image, they are drawn and stored for reference. This helps visualize the detected features before proceeding with stitching.

Images with Keypoints

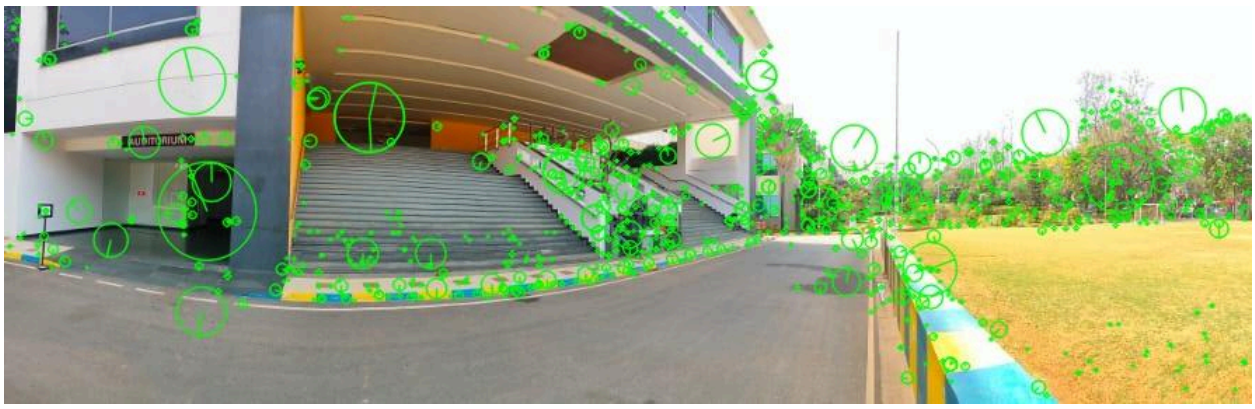


Image Stiching

Image stitching is performed using a **feature-based approach**, where the detected key points are matched between overlapping images to determine their alignment. The process is as follows:

- **Feature Matching:**

  Once the key points and descriptors are extracted, the **Brute Force Matcher (BFMatcher)** is used to compare descriptors between consecutive images. The **L2 norm** is used as the distance metric to find the best-matching features. To ensure robustness, **KNN matching** is applied with **k=2**, meaning for each key point, the two best matches are found. A **ratio test** is then performed, where only matches where the best match is significantly better than the second-best match are considered valid. This helps filter out incorrect matches.
- **Homography Estimation:**

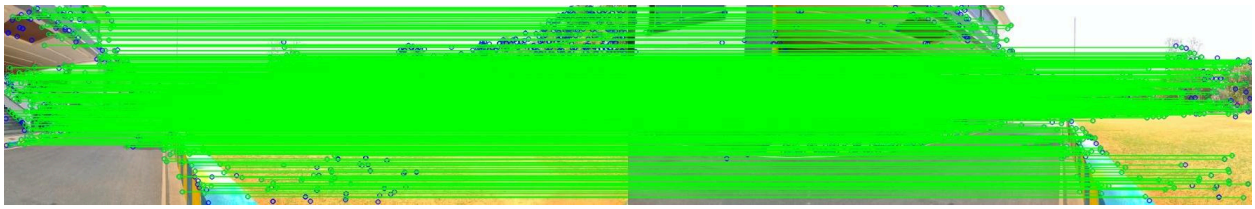
  After identifying matching key points, their spatial relationships are used to estimate a transformation matrix known as the **homography matrix (H)**. This matrix allows one image to be warped so that it aligns with the next image in the sequence. The homography is calculated using **RANSAC (Random Sample Consensus)**, which ensures that outliers (incorrect matches) do not affect the final alignment.
- **Perspective Warping and Stitching:**

  The computed homography matrix is used to warp the second image onto the first. To handle misalignments and avoid harsh transitions, an **image blending technique** is applied. This includes:
    - Creating **masks** for the overlapping regions.

    - Using **distance transform-based weight maps** to achieve smooth blending between overlapping areas.
- **Handling Black Borders:**

  After stitching multiple images, unwanted black borders may appear due to warping. To remove these, a **contour detection method** is applied to detect the region of interest, and the final stitched image is cropped accordingly.

Matched Image



Final Panaromic Output