VISUAL RECOGNITION
AIM-825: Sec A

ASSIGNMENT-1

Name	Roll no	Email
Aryan Mishra	IMT2022502	Aryan.Mishra@iiitb.ac.in

PART-A(coins):

a. Detect all coins in the image:Use edge detection, to detect all coins in the image. Visualize the detected coins by outlining them in the image.

b. Segmentation of Each Coin: Apply region-based segmentation techniques to isolate individual coins from the image. Provide segmented outputs for each detected coin.

c. Count the Total Number of Coins: Write a function to count the total number of coins detected in the image. Display the final count as an output.

(a) Edge Detection

1.Library Imports and Setup

→ Imported OpenCV (cv2), NumPy, and Matplotlib for image processing and visualization.

→ Loaded the coin image (coin_image2.png) and converted it to grayscale.

→ Resized the image for convenience in display and processing.

2. We used Canny edge detector as it has several advantages over Mark Hildreth edge detector as it minimises error avoiding false edges, ensures edge thickness and maintains continuity

→ A Gaussian blur (cv2.GaussianBlur()) was applied beforehand to reduce image noise. This was particularly important for accurately detecting the edges of 10-rupee coins as it has a circle inside as well.

3.Edge Detection Workflow

→ Blur the image with a kernel size of (9, 9). This choice was critical to reduce noise without eliminating the finer edges.

→ We got an idea of the thresholds using the median of the blurred image and then used cv2.Canny(). This helps adapt the detection thresholds to the specific image brightness and contrast.

4.Contour Detection

→ Once edges were identified, they were morphologically filled and then, cv2.findContours() was used to locate the outlines (contours) of each coin.

→ The discovered contours were then drawn on a copy of the original image for a visual check (to confirm if the edges properly circled the coins).

B. Segmentation

After detecting edges and contours, we explored three main segmentation approaches for isolating each coin:

Three methods were explored: Bounding rectangle, Masked segmentation and minimum enclosing circle. Methods (a) and (c) gave the best outputs as observed in the python notebook and README file.

C. Counting the Coins

→ Finally, a simple count_coins() function was defined, which uses len(contours) to count how many distinct contours (i.e., coins) are in the image.

→ The number is displayed directly on the image via cv2.putText().

<u>Initial problems and their solutions:</u>

→ A lot of false edges were being detected especially in the 10 rupee coin where the inner circle was also detected as an edge. We solved this by increasing the Gaussian blurring by using a bigger kernel. But we had to be careful there as well, as too big of a kernel implied less true edges to be detected and too less of a kernel implied more false edges to be detected.

→ The thresholds obtained through the median gave good results but not certainly the best. So, we tried tweaking it by referring to what was obtained using median and obtained very good threshold values empirically.

→ We can observe in the input image that in the first row the 3 coins are placed very close to each other. So, with the kernel used for morphological closing being of size (5,5) initially, the entire first row was detected as 1 coin. We reduced the size of the kernel to (2,2) to solve this, so that the edges were thinner.

To conclude, after making the necessary modifications, we got near perfect outputs as can be seen in the README file.

PART-2(Panorama stitching):

a. Extract Key Points: Detect key points in overlapping images.

b. Image Stitching: Use the extracted key points to align and stitch the images into a single panorama. Provide the final panorama image as output.

(a) Keypoint Extraction

→ Read multiple overlapping images(img_left, img_centre, img_right).

→ Detect keypoints using SIFT.

→ Compute descriptors for each keypoint.

(b) Stitching

→ Warp one image onto the plane of the other using the estimated homography.

→ Blend the overlapped region to reduce visible seams.

→ The Stitcher from OpenCV essentially matches descriptors across overlapping images, filters matches(e.g., Lowe's ratio test) and finally estimates a homography transformation that aligns the overlapping areas.

→ Cylindrical warper function was used.

→ The black borders were removed from the stitched image generating the final stitched image.

Initial problems and solutions:

→ There were a lot of black areas in the resultant stitched image, so we removed them by using the bounding rectangle approach.

→ We initially tried tuning the parameters like Lowe's ratio test and also RANSAC parameters a lot but couldn't achieve good results so, the above is done using Stitcher from the OpenCV module.

→ We used a planar warper function initially(the component that projects each input image onto a specified geometric surface—such as a plane, cylinder, or sphere). This caused the resultant image to have tilted edges. As our input images, together are taken over a wide range, hence we see these distortions were caused. As the cylindrical warper function works well over a wide range, it was used to fix this issue.

Hence, after applying the above solutions, the resultant image was improved a lot as can be seen in the README file. But, it still contains some seam due to misalignment(parallax), exposure differences, poor blending, or lens artifacts like vignetting.