Assignment 1

- Sasi Snigdha Yadavalli IMT2022571

Github Repo Link:
https://github.com/Snigdha2005/VR_Assignment_1_Snigdha_IMT2022571

Coin edge detection, segmentation and counting

Edge Based detection using contours

1. Preprocessing and Edge Detection
    - Grayscale Conversion: The input image is first converted to grayscale using cv2.cvtColor(image, cv2.COLOR_BGR2GRAY). This step simplifies the image by removing color information, making it easier to process.
    - Gaussian Blur for Noise Reduction: A Gaussian blur is applied with cv2.GaussianBlur(gray, (11, 11), 0). The blurring smoothens variations in pixel intensities, reducing noise and making edge detection more reliable. Larger kernel sizes (like (11, 11)) provide stronger smoothing, but excessive blurring may remove weak edges.
    - Canny Edge Detection: Edges are detected using cv2.Canny(blurred, 50, 234), which applies gradient-based edge detection. The first threshold (50) filters weak edges, and the second threshold (234) filters out insignificant edges. This helps in detecting the circular boundaries of the coins while ignoring unwanted noise.
2. Contour Detection and Extraction
    - Finding Contours: cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE) extracts the outermost contours from the detected edges. Contours represent the boundaries of objects (coins) in the image.
    - Drawing Contours on the Image: cv2.drawContours(output, contours, -1, (0, 255, 0), 3) overlays the detected contours on the original image in green. This visualization confirms whether the contours correctly enclose the coin boundaries.
    - Extracting Individual Coins: Each detected coin is cropped using cv2.boundingRect(contour), and saved as a separate image. This segmentation isolates each coin for further processing or analysis.

Region based segmentation using thresholding and morphological operations

- Otsu's Thresholding for Foreground Extraction: cv2.threshold(blurred, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU) automatically finds an optimal threshold to separate coins from the background. This converts the image into a binary format (white foreground, black background), making segmentation easier.
- Morphological Operations for Noise Removal: cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel, iterations=2) applies an opening operation (erosion followed by dilation). This removes small noise patches while preserving the shape of coins.
- Connected Component Labeling: measure.label(cleaned, connectivity=2) labels each distinct connected region in the binary image. This step helps identify individual coins and separate them from the background.
- Removing Small Objects: morphology.remove_small_objects(labels, min_size=2000) eliminates small regions that are unlikely to be coins. This ensures that only sufficiently large objects are considered as valid coin detections.
- Colorizing the Segmented Coins: color.label2rgb(labels, bg_label=0) assigns unique colors to different detected regions. This helps in visually distinguishing each segmented coin.
- Counting the Total Number of Coins: len(np.unique(labels)) - 1 counts the total segmented regions while excluding the background. This provides an automatic way to count the number of coins in the image.

Dependencies and Installations:

OpenCV, NumPy, OS, Shutil, Matplotlib, Scikit-Image

pip install opencv-python numpy matplotlib scikit-image

Ensure the input images are available in the input_images directory. Run the following command to execute the code:

python3 Q1.py

Output files generated will be available in the output_images directory.

Original Image:

Output Images


2012


भारत INDI
2010


INDIA
2010


2008


5
FIVE RUPEES

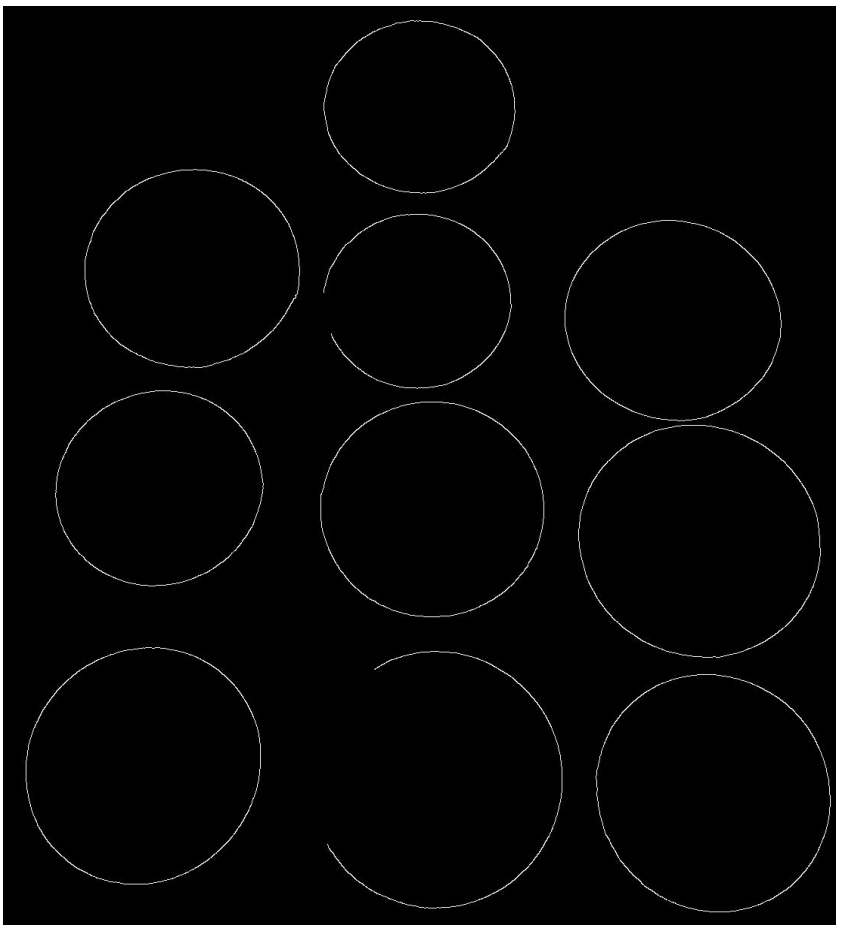

भारत INDIA
2010

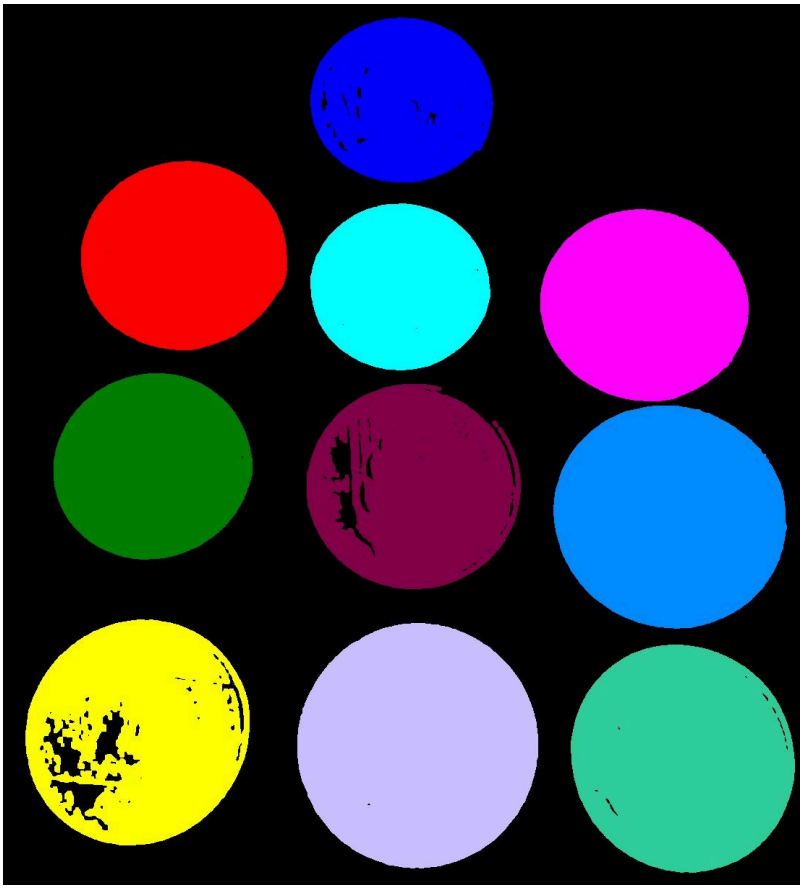

1
2018


RUPEE
2009


2
TWO RUPEES


2
TWO RUPEES
1
RUPEE
2009
1
2018
भारत INDIA
2010
5
2015
5
FIVE RUPEES
2008
भारत INDIA
2010
भारत INDIA
2010
2012





Edge Detection + Contour Total number of coins detected: 10
Segmentation Total Coins Detected: 10

Image Stitching - Panorama Generation

This script stitches multiple images into a seamless panorama using feature detection, homography estimation, and warping techniques.

Steps Involved

1. Preprocessing and Loading Images
    - Reads all images from the input_images directory that contain _panorama in their filename.
    - Resizes images to a fixed width (800 pixels) while maintaining the aspect ratio.
2. Feature Detection and Matching
    - Detects keypoints and computes descriptors using SIFT (Scale-Invariant Feature Transform).
    - Uses the Brute-Force Matcher (BFMatcher) with L2 norm to find feature correspondences between consecutive images.
    - Saves keypoint visualizations for reference.
3. Homography Estimation and Warping
    - Computes homography using RANSAC to estimate the transformation between images.
    - Warps images to align them and blends them into a single panoramic image.
4. Trimming Black Borders
    - Identifies and removes unnecessary black areas in the final stitched panorama.
5. Saving and Displaying Output
    - The final stitched image is saved in the output_images directory as stitched_panorama.jpg.
    - The result is displayed using Matplotlib.

Dependencies and Installations

OpenCV, NumPy, Matplotlib, and Glob

pip install opencv-python numpy matplotlib

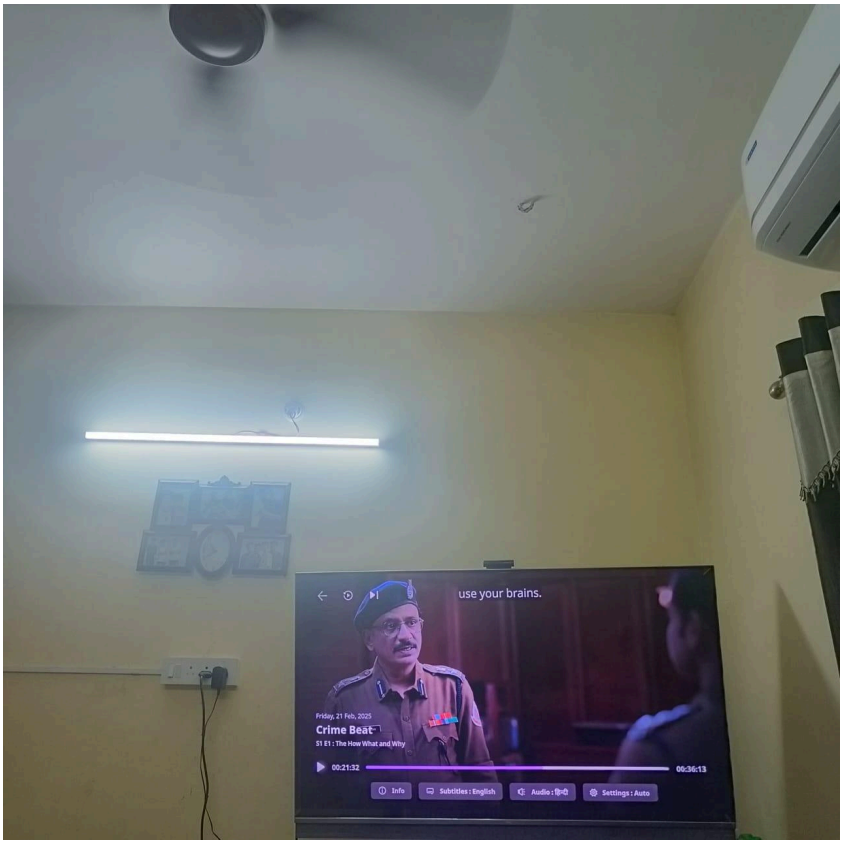
Ensure the input images are available in the input_images directory. Run the following command to execute the code:

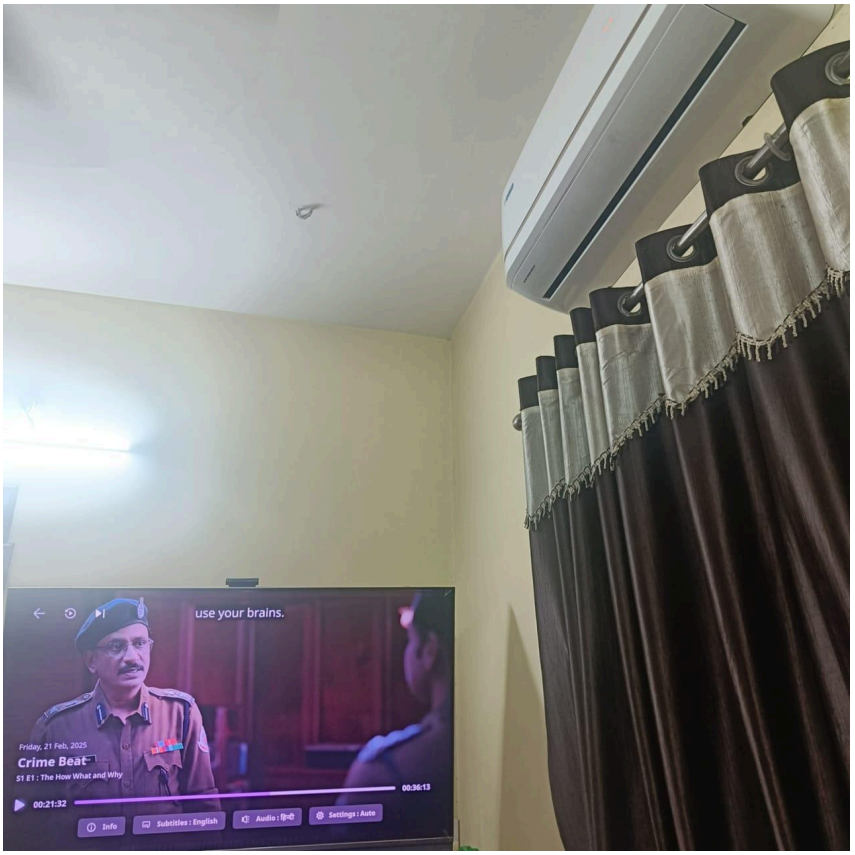
python3 Q2_1.py

Q2.py uses Stitcher API.

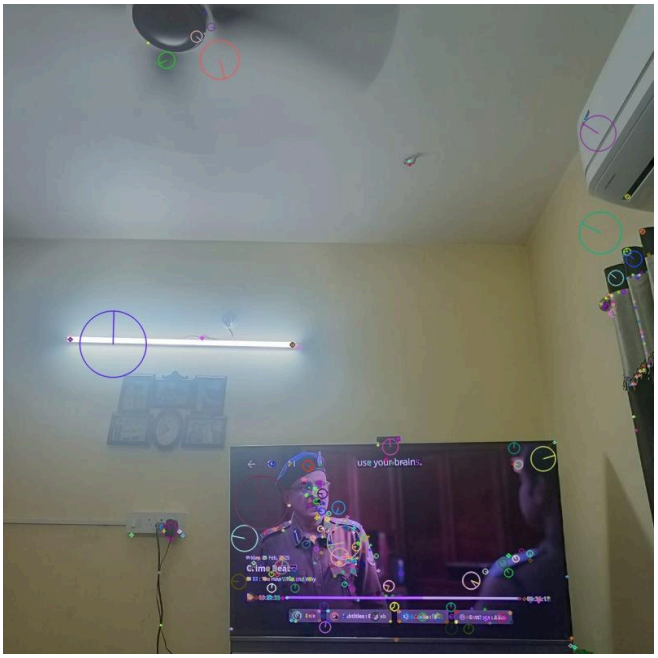
Output files generated will be available in the output_images directory.

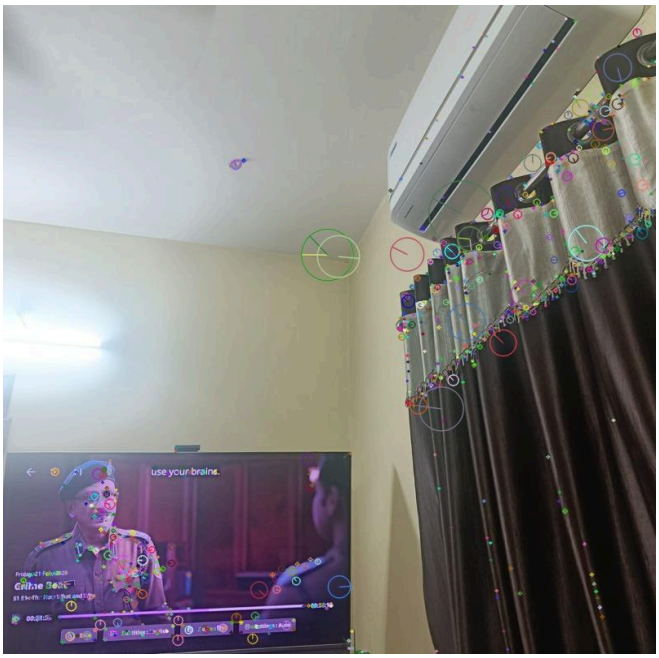
Original Images

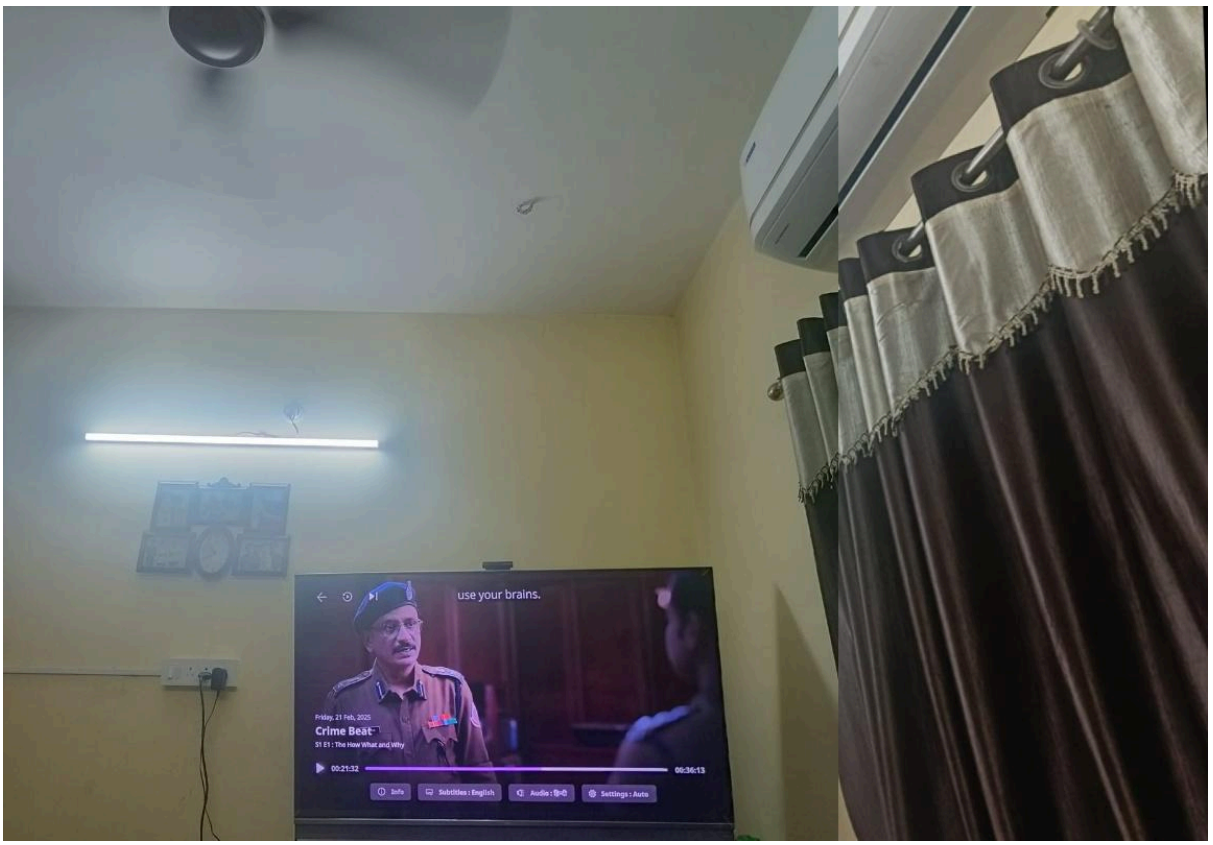




Output Images







Folder Structure

```
├── Q1.py
├── Q2.py
├── Q2_1.py
├── README.md
├── input_images
│   ├── A_panorama.jpg
│   ├── B_panorama.jpg
│   └── scatter.jpg
└── output_images
    ├── 1_keypoints.jpg
    ├── 2_keypoints.jpg
    ├── coin_1.jpg
    ├── coin_10.jpg
    ├── coin_2.jpg
    ├── coin_3.jpg
    ├── coin_4.jpg
    ├── coin_5.jpg
    ├── coin_6.jpg
    ├── coin_7.jpg
    ├── coin_8.jpg
```

├── coin_9.jpg
├── detected_coins.jpg
├── edges.jpg
├── region_based_segmentation.jpg
└── stitched_panorama.jpg

Results and Observations

Coin Edge Detection, Segmentation, and Counting

1. Edge-Based Detection (Contours Method)
- Successfully detected coin edges using the Canny Edge Detector.
- Contours were drawn accurately around the coin boundaries.
2. Region-Based Segmentation (Thresholding and Morphological Operations)
- Otsu's thresholding effectively separated the foreground (coins) from the background.
- Morphological operations helped in noise reduction and filling gaps in segmented coins.
- The connected component analysis counted the number of coins correctly in most cases.
- Some overlapping coins may affect the segmentation accuracy.

Final Output

- The total number of coins was correctly counted.
- Segmented coin images were stored successfully in the output_images folder.

Panorama Stitching

1. Keypoint Detection and Matching
    a. SIFT detected keypoints efficiently in overlapping image regions.
    b. Brute-force matching worked well to establish feature correspondences.
    c. Keypoints were saved as images for visualization.
    d. In addition to SIFT, ORB (Oriented FAST and Rotated BRIEF) was also tested for keypoint detection and descriptor extraction. ORB is a more computationally efficient alternative that performs well in real-time applications.
2. Image Warping and Homography
- The computed homography matrix effectively aligned images.

- Warping successfully transformed images onto a common perspective.
- Some distortions were observed at the edges due to perspective transformation.
- Instead of manually computing homography and warping, OpenCV's built-in `cv2.Stitcher.create()` API was also tested. This method provided an automated stitching solution, reducing manual homography computations.
- The `cv2.Stitcher_SCANS` mode was used to optimize stitching for images captured in a scanning sequence.

Final Panorama

- Trimming black borders improved the aesthetics of the panorama.
- Misalignment issues arose in cases of extreme perspective differences or low feature matches.
- ORB-based keypoint detection was faster than SIFT but resulted in slightly fewer matching keypoints, affecting alignment in some cases.
- The automated stitching pipeline using `cv2.Stitcher.create()` provided a seamless result for well-overlapping images but struggled with images having limited common features.

Challenges

- Stitching failed or produced artifacts when input images had poor overlap.
- Variations in brightness or exposure caused visible seams in some panoramas.
- ORB, though computationally efficient, sometimes missed key feature points compared to SIFT, leading to stitching errors.
- The OpenCV Stitcher API was tested as an alternative approach, but it had limitations when images lacked sufficient distinct keypoints.