AIM 825- Sec-A: Visual Recognition Assignment 1

Part 1

Introduction

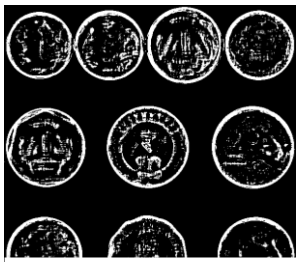
This report details the implementation of a computer vision-based approach for detecting, segmenting, and counting coins in an image using OpenCV. The project leverages image preprocessing techniques, contour detection, and segmentation to accurately isolate coins from a background.

Methodology

Sample input

1. Image Preprocessing

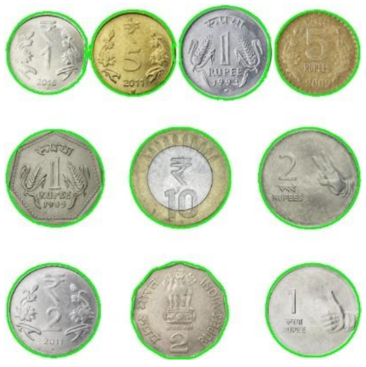
- The input image is loaded and resized to ensure consistent processing.
- The image is converted to grayscale for easier analysis.
- Contrast stretching is applied to enhance the features.
- A median blur filter is used to reduce noise while preserving important edges. (Gaussian Blur failed to remove noise in certain images, which resulted in faulty contour detection)
- Other approaches like image whitening, histogram equalization etc, failed to generalize and gave underwhelming results for some images.
- Adaptive thresholding is applied to create a binary image for contour detection.



Preprocessed Image

2. Contour Detection

- The contours of potential coin regions are detected using the `cv2.findContours` function.
- Contours are filtered based on circularity and area constraints to remove false positives.
- The detected contours are drawn on the original image for visualization.


Coins with contours/edges detected

3. Coin Segmentation

- A mask is created for each detected coin, isolating it from the background.
- The segmented coin images are saved separately for further analysis.
- Each coin is extracted using bounding rectangles and displayed individually.

Some segmented coins

4. Counting Coins

- The total number of detected coins is determined based on the number of valid segmentations found.
- The result is displayed as output.

Results and Observations

- The algorithm successfully detects and segments coins from images with minimal background noise.
- Performance may vary depending on lighting conditions and image quality.
- Further improvements can be made by incorporating more robust filtering techniques to eliminate non-coin objects.
- The process is not robust enough to work with images that have overlapping coins

Conclusion

The project demonstrates an effective method for detecting and segmenting coins using OpenCV. This approach can be further optimized for real-world applications such as automated coin counting systems. Future enhancements could include deep learning-based classification to distinguish different coin denominations.

References

- OpenCV Documentation: https://docs.opencv.org/
- NumPy Documentation: https://numpy.org/doc/

Part 2: Image Stitching

Introduction

Image stitching is a technique used to combine multiple images into a single seamless panoramic image. This report compares two different approaches to image stitching:

1. **`panaroma.py`** – Uses OpenCV's built-in `Stitcher_create()` function.
2. **`panaroma_manual.py`** – Implements a custom stitching pipeline using feature detection, matching, and homography transformation.

Both methods attempt to remove black borders in the final stitched output, but each has its own strengths and limitations.

Methodology

`panaroma.py`

- Uses OpenCV's `Stitcher_create()` for automatic image alignment and stitching.
- Detects ORB keypoints in the input images before stitching.
- Applies a cropping function to remove black borders from the stitched result.
- Saves intermediate images including loaded images, keypoints, raw stitched output, and cropped output.

`panaroma_manual.py`

- Uses SIFT for feature extraction and descriptor matching.
- Matches keypoints between images iteratively.
- Uses homography transformation to align and stitch images.
- Removes black seams after every iteration.
- Crops the stitched result to remove tilted black seams.
- Saves the final stitched image.

Comparison

Feature	`panaroma.py` (OpenCV Stitcher)	`panaroma_manual.py` (Custom Stitching)
Feature Detection	ORB	SIFT
Image Alignment	Automatic (Stitcher API)	Homography-based
Keypoint Matching	Not required (handled internally)	Brute-Force Matcher (BFMatcher)
Seam Handling	Cropping removes black areas (but may lose data)	Black seams removed iteratively (may discard keypoints)
Stitching Quality	High (when images have enough overlap)	Lower accuracy, depends on good matches
Speed	Faster (optimized in OpenCV)	Slower (iterative matching and transformations)
Black Border Removal	May remove excess information	May cause missing keypoints, affecting stitching

Observations

`panaroma.py`

- Produces better stitching results due to OpenCV's optimized stitching pipeline.
- The cropping function effectively removes black borders but sometimes removes too much information.
- Works best with images having sufficient overlap and minimal distortion.

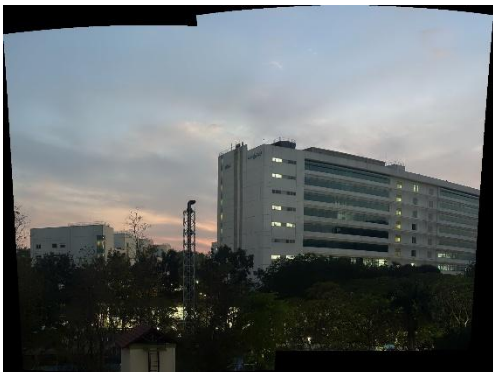
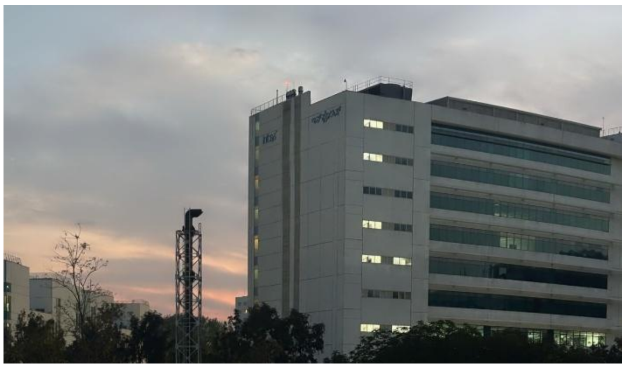


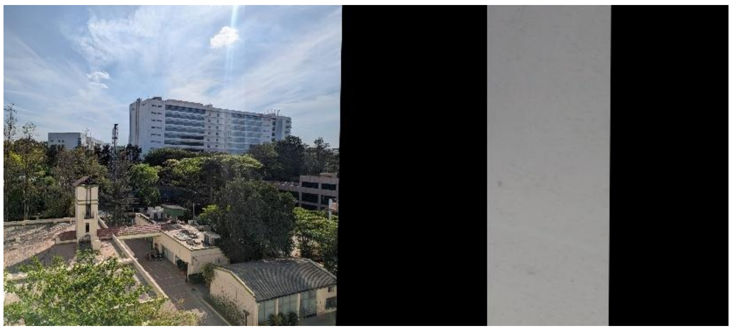
Image stitched using Stitcher_create()

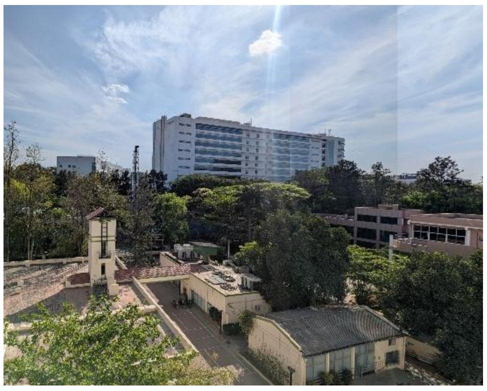

After cropping black seams we see that some useful parts are also lost

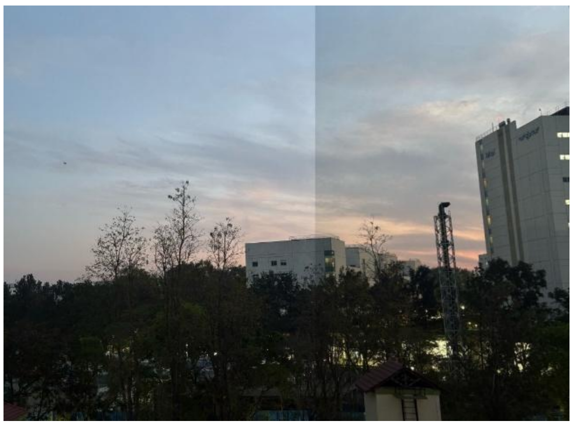
`panaroma_manual.py`

- Provides more control over the stitching process but requires careful tuning.

- Black seam removal may result in missing keypoints, leading to failed stitching in subsequent images.
- More computationally expensive and prone to errors when keypoints are not well-matched.
- Some images may not get stitched due to missing keypoints after seam removal.


Stitching without black seam removal after every step


Stitching with black seam removal after every iteration


The right side of the image failed to get stitched after black seam removal in the intermediate step.

Conclusion

- If the priority is **accuracy and speed**, `panaroma.py` is the better choice since OpenCV's `Stitcher_create()` produces high-quality results with minimal effort.
- If **custom control and iterative improvements** are needed, `panaroma_manual.py` provides flexibility but requires careful tuning of feature detection, matching, and seam removal.
- Future improvements could include using adaptive cropping techniques to minimize data loss and refining keypoint filtering in `panaroma_manual.py` to prevent missing features during iterative stitching.

Recommendations

1. **For most users:** Use `panaroma.py` for quick and effective stitching.
2. **For advanced users:** Improve `panaroma_manual.py` by implementing better black seam detection and ensuring keypoints are retained across images.
3. **Further improvements:** Consider blending techniques like multi-band blending to reduce seams and enhance image transitions.