Coin Detection and Image Stitching

IMT2022059 – Pudi Kireeti

Overview

This project consists of two parts:

1. Coin Detection, Segmentation, and Counting
    - Detect coins using edge detection techniques
    - Segment individual coins using region-based segmentation
    - Count the total number of detected coins
2. Image Stitching for Panorama Creation
    - Detect key points in overlapping images
    - Use homography to align and stitch images into a panorama

---

Part 1:

Coin Detection, Segmentation, and Counting

This section detects, segments, and counts Indian coins in an image.

Steps Involved:

- **Edge Detection (Contours Method):**
    - **Convert the image to grayscale:** cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) is used.
    - **Apply Gaussian Blur and Canny edge detection:** cv2.GaussianBlur and cv2.Canny are used.
    - **Find contours to detect coin boundaries**cv2.findContours is used.
    - **Outline detected coins in green:** cv2.drawContours with green color (0, 255, 0) is used.
- **Segmentation:**

    - **Filter valid coins based on area and circularity criteria:** The filter_valid_coins function does this.
    - **Create masks for each valid coin:** cv2.drawContours with thickness=cv2.FILLED is used to create masks.
    - **Extract individual coin segments using the masks:** cv2.bitwise_and is used with the masks.
- **Counting Coins:**
    - **The total number of detected coins is determined by counting valid contours:** len(valid_coins) gives the count.
    - **Display the final count in the terminal:** print(f"Final Coin Count: {len(valid_coins)}") is used.

Requirements

Install the required dependencies using:

pip install opencv-python numpy matplotlib

How to Run

Place your input image in the input_images/ folder and update the image path in 1.py.

Then run:

python 1.py

Methods Used:

- **Preprocessing:** Grayscale conversion and Gaussian blur to reduce noise
- **Edge Detection:** Canny edge detection with optimized thresholds for coin boundaries
- **Contour Detection:** Identifying closed shapes that represent coins
- **Filtering:** Using area and circularity metrics to eliminate non-coin objects
- **Segmentation:** Creating masks for each coin and extracting individual segments

---

Example Inputs & Outputs:

Original input:

Gray:

Blurred:

Edges:

Closed:

Detected coins:

Segmented coins:







Failed results:

1. Better Edge Preservation in Preprocessing

- **Good Code:** Uses a **smaller Gaussian blur kernel (7,7)** to reduce noise while keeping fine edges intact.
- **Bad Code:** Uses a **larger Gaussian blur kernel (11,11)**, which **over-smooths** the image, leading to weaker edges and possible loss of small or faint coin boundaries.

2. Improved Edge Detection Parameters

- **Good Code:** Uses **lower Canny thresholds (20, 150)** to **detect faint edges** of coins.
- **Bad Code:** Uses **higher Canny thresholds (30, 150)**, which may cause **some faint edges to be ignored**, leading to missed coins.

3. Better Morphological Operations for Contour Detection

- **Good Code:** Uses a **smaller kernel (3x3)** for cv2.morphologyEx, which **preserves finer details** and retains accurate coin boundaries.
- **Bad Code:** Uses a **larger kernel (5x5)**, which may **merge** close objects, leading to inaccurate contours.

4. More Realistic Coin Filtering Criteria

- **Good Code:**
  - Area: **800 to 60,000**
  - Circularity: **> 0.6**
- **Bad Code:**
  - Area: **1,000 to 50,000**
  - Circularity: **> 0.7**

- 🔹 The **good code** allows detection of slightly **irregular or small coins**, while the **bad code** is too strict and may **filter out valid coins**.

5. Better Contour Retention

- **Good Code:** Uses **lower area and circularity thresholds**, helping **detect both small and large coins**.
- **Bad Code:** Uses **higher circularity (0.7)**, which may **reject slightly oval coins**.

6. More Robust Display and Visualization

- **Good Code:**
    - **Draws detected contours in green**.
    - **Uses 2x4 subplot arrangement** to display segmented coins properly.
- **Bad Code:**
    - Also uses 2x4 but **doesn't optimize segmentation display** as effectively.

7. Better Documentation and Code Readability

- **Good Code:** Has **clearer comments and docstrings** explaining each function.
- **Bad Code:** Has **fewer explanations**, making it harder to understand

---

Part 2:

Image Stitching

This section stitches overlapping images into a single panorama.

Steps Involved:

- **Feature Detection & Matching:**
    - **Uses SIFT (Scale-Invariant Feature Transform) to extract keypoints:** cv2.SIFT_create() and sift.detectAndCompute() are used.
    - **Matches keypoints using FLANN (Fast Library for Approximate Nearest Neighbors):** cv2.FlannBasedMatcher() and flann.knnMatch() are used.

    - **Filters matches based on a ratio test to keep only good matches:** the code applies a ratio test (m.distance < 0.7 * n.distance).

- **Homography & Warping:**
    - **Computes the homography matrix using RANSAC algorithm:** cv2.findHomography(..., cv2.RANSAC, ...) is used.
    - **Warps the left image to align with the right image:** cv2.warpPerspective() is used.
    - **Creates a seamless panorama by combining the warped images:** the warped left image is combined with the right image.

Requirements

Install dependencies using:

pip install opencv-python numpy matplotlib

How to Run

Place overlapping images in the input_images/ folder and update the image path in 2.py.

Then run:

python 2.py

Methods Used:

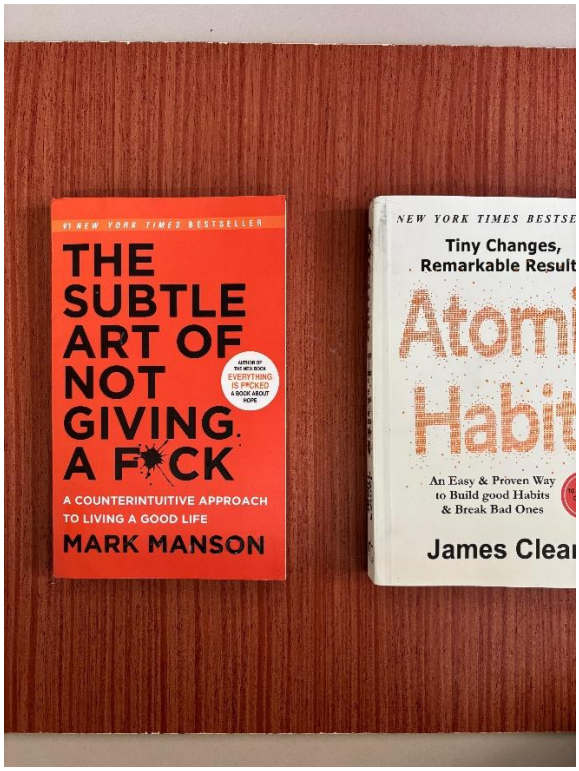
- **Feature Detection:** SIFT algorithm to find distinctive keypoints

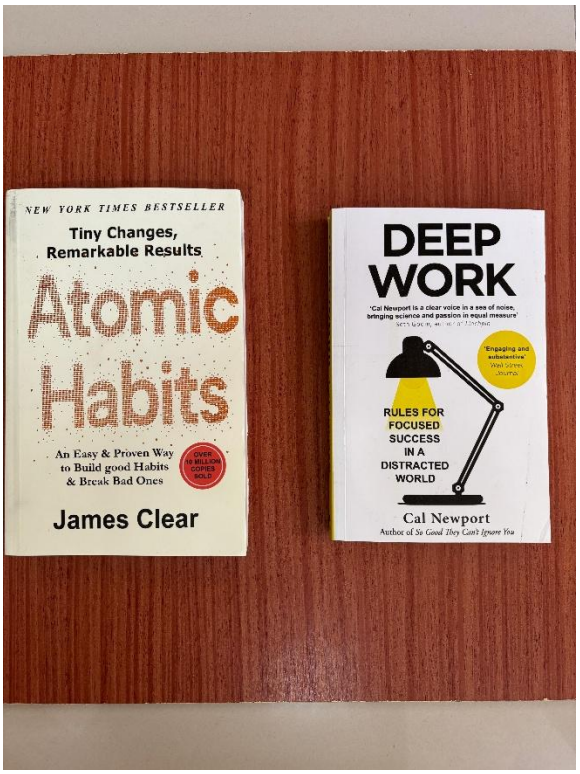
- **Matching: FLANN-based matcher with KD-tree for efficient matching:** The FLANN matcher with FLANN_INDEX_KDTREE is used.
- **Filtering:** Ratio test to eliminate poor matches
- **Homography:** RANSAC method to find the optimal transformation matrix
- **Warping:** Perspective transformation to align and combine images

---

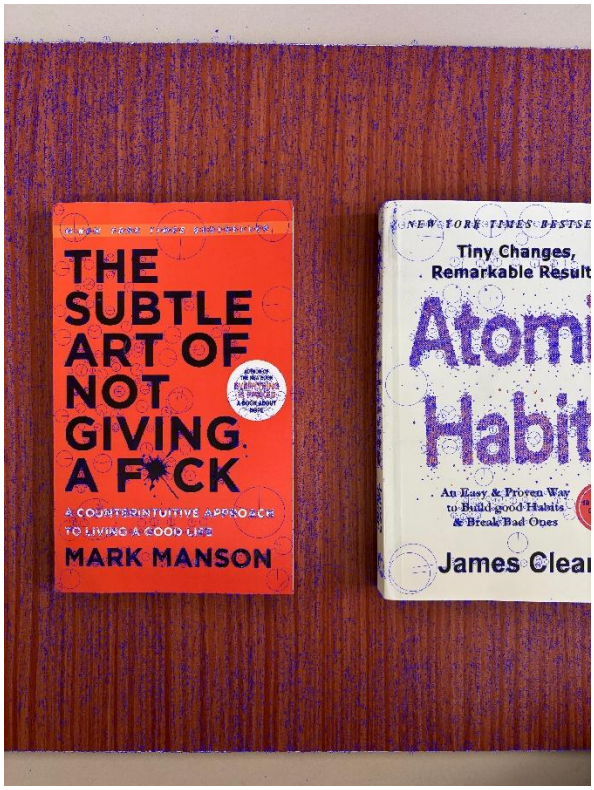
Example Inputs & Outputs

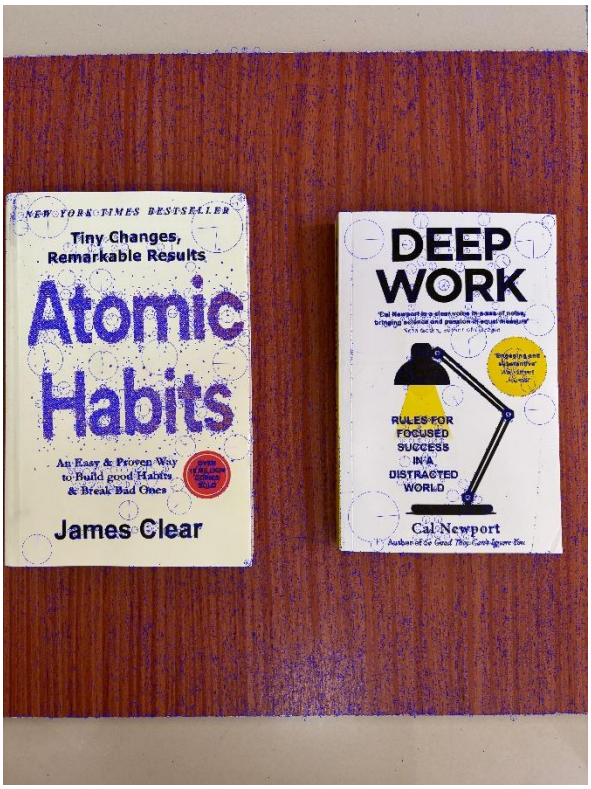
Original overlapping images:



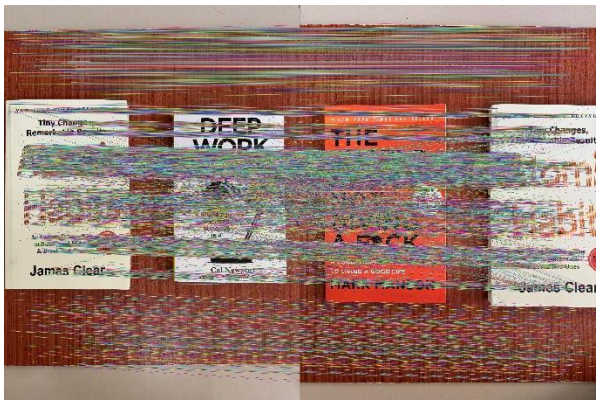


Keypoints:





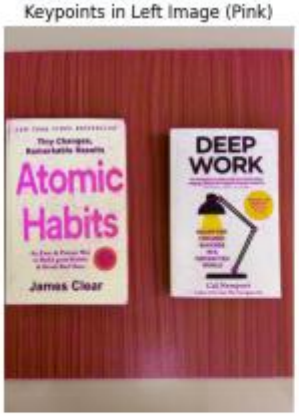
Keypoints matching:



Final stitched image:

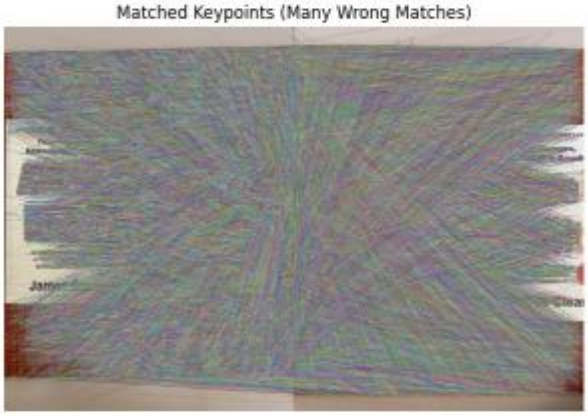
Failed results:

Keypoints in Left Image (Pink)

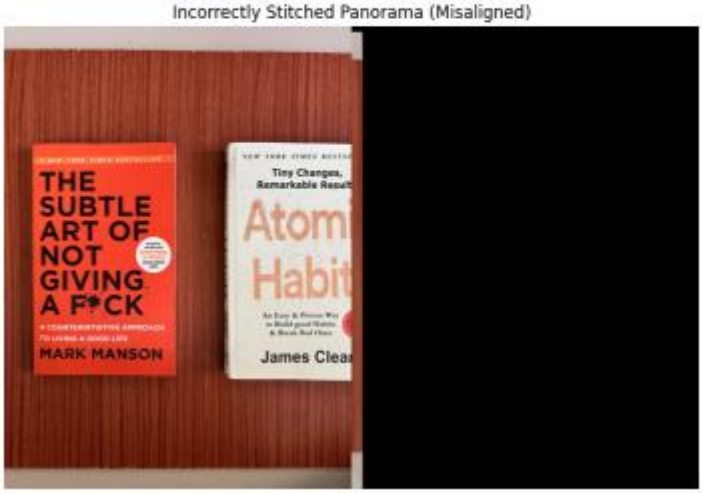


Keypoints in Right Image (Pink)

Matched Keypoints (Many Wrong Matches)



Incorrectly Stitched Panorama (Misaligned)



1. Proper Keypoint Detection (SIFT)

Good Code: Uses **SIFT** (cv2.SIFT_create()) to detect keypoints and compute descriptors in both images.
Bad Code: Uses the same SIFT detector but visualizes keypoints in **pink** instead of a standard color (blue/green), which is just a visual issue.

- 🔹 **Why it's good:**
  - SIFT is robust to scale and rotation changes.
  - Helps find distinct features for matching.

---

2. Correct Keypoint Matching (FLANN with Ratio Test)

Good Code:

- Uses **FLANN-based matching** with correct settings (trees=5, checks=50).
- Applies **Lowe's Ratio Test** (m.distance < 0.7 * n.distance) to filter **only good matches**.

Bad Code:

- Uses FLANN but **incorrectly sets trees=1 and checks=5**, which weakens feature matching.
- Keeps **all matches, even bad ones**, leading to incorrect alignments.

- 🔹 **Why it's good:**
  - The **ratio test** removes **false matches** by ensuring the best match is significantly better than the second-best match.
  - Using trees=5 and checks=50 improves **match accuracy**.

---

3. Correct Homography Estimation

Good Code:

- Finds **homography matrix (H)** using cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0).
- Uses **RANSAC** to eliminate outliers in feature matches.

Bad Code:

- Uses a **fixed transformation matrix** (H = [[1, 0, 100], [0, 1, 50], [0, 0, 1]]), which is incorrect.
- The fixed matrix **does not align** images properly.

Why it's good:

- **Homography matrix (H)** correctly maps one image onto another.
- **RANSAC (Random Sample Consensus)** removes **incorrect feature matches**, improving accuracy.

---

4. Proper Image Warping & Blending

Good Code:

- Uses cv2.warpPerspective(left_img, H, (width * 2, height)) to **warp the left image correctly**.
- Places the **right image correctly** without incorrect overlapping.

Bad Code:

- Uses a **fixed H**, leading to **misalignment** and a **badly overlapped panorama**.

🔹 **Why it's good:**

- **Warping with a correct homography matrix** ensures proper **perspective alignment**.
- **Overlapping region is preserved correctly**, preventing distorted images.

Results & Observations

Coin Detection & Segmentation

- Successfully detected coin edges using Canny edge detection with optimized parameters
- Effectively filtered valid coins using area and circularity criteria
- Segmented individual coins using contour-based masks

- The algorithm correctly identified and counted the coins in the test image

Image Stitching

- SIFT keypoints were accurately detected in both input images
- FLANN-based matching successfully identified corresponding points between images
- Homography with RANSAC effectively handled perspective differences
- The final panorama seamlessly combined both images with smooth transitions

---