COMPUTER VISION

Assignment 2

Nabeel Mohamed 7547

Mohamed Milad 7479

Eyad Allafy 7839

1. Part 1: Augmented Reality with Planar Homographies

Code Explanation

- Open the video file using OpenCV's `VideoCapture` to allow frame-by-frame access.
- Read the first frame from the video.
- Check if the frame was successfully retrieved.
- Save the extracted frame as an image file frame (`video_frame.jpg`) using OpenCV's `imwrite`.
- Release the video capture object to free system resources.

- **Image Loading**:
  - Load the grayscale versions of the reference image (`cv_cover.jpg`) and the video frame (`video_frame.jpg`) using `cv2.imread` with the `cv2.IMREAD_GRAYSCALE` flag.
- **Keypoint Detection and Descriptor Extraction**:
  - Initialize a SIFT detector using `cv2.SIFT_create`.
  - Compute keypoints and their corresponding feature descriptors for both images using `sift.detectAndCompute`.
    - `keypoints1`, `keypoints2`: Lists of keypoints detected in the images.
    - `descriptors1`, `descriptors2`: Corresponding feature descriptors.
- **Descriptor Matching**:
  - Instantiate a brute-force matcher (`cv2.BFMatcher`).
  - Use KNN matching (`bf.knnMatch`) to find the two closest matches for each descriptor based on Euclidean distance.

- **Lowe's Ratio Test**:

  - Apply Lowe's ratio test to filter matches:
    - Retain a match if the distance of the closest match is less than 0.75 times the distance of the second-closest match, reducing the likelihood of false positives.

- **Top Matches Selection**:

  - Sort the filtered matches by distance in ascending order using `sorted`.
  - Select the top 50 matches for visualization.

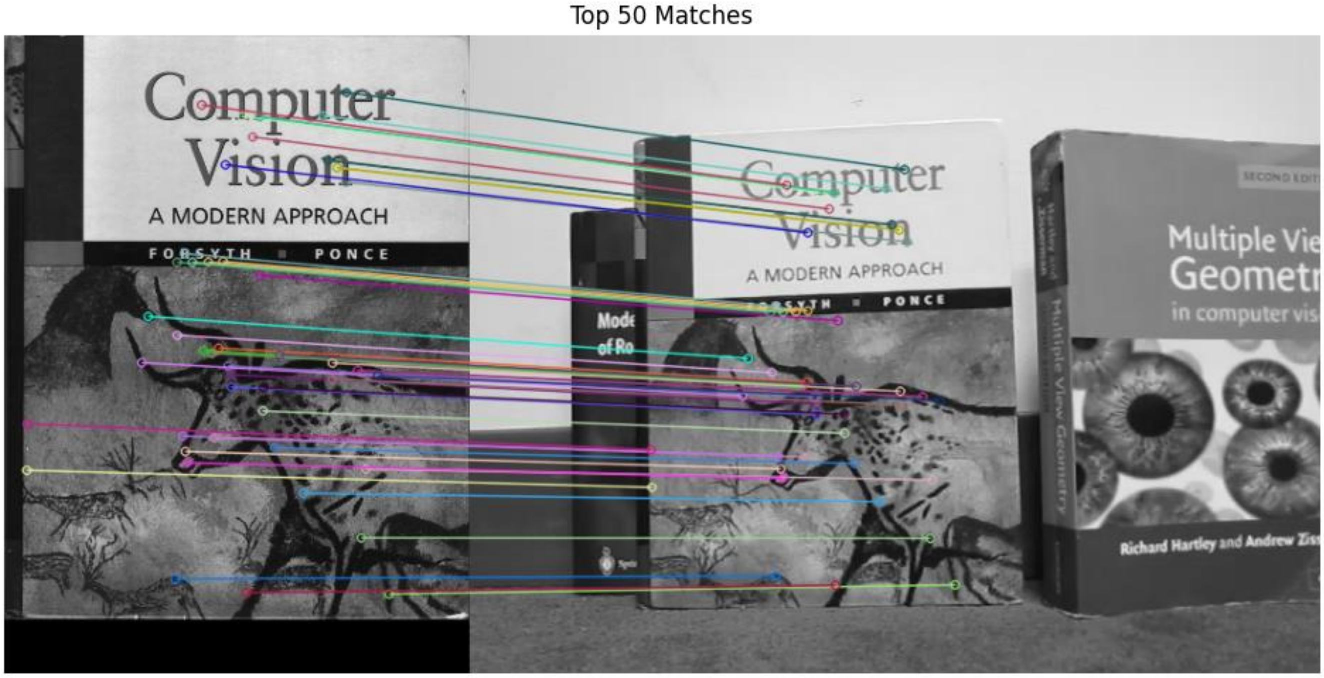
- **Match Visualization**:

  - Use `cv2.drawMatches` to overlay the matched keypoints on a combined image of the two inputs.
  - The `cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS` flag ensures only matched points are visualized.

- **Output Display**:

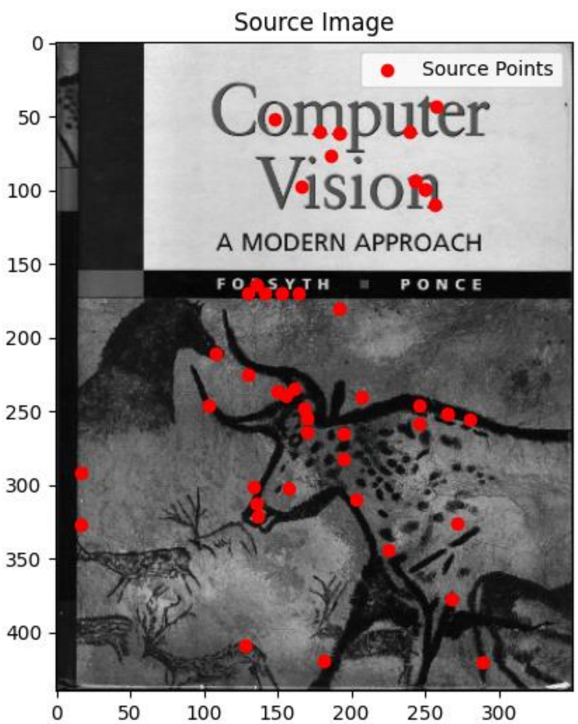
  - Render the visualization using Matplotlib (`plt.imshow`) with a larger figure size for clarity.

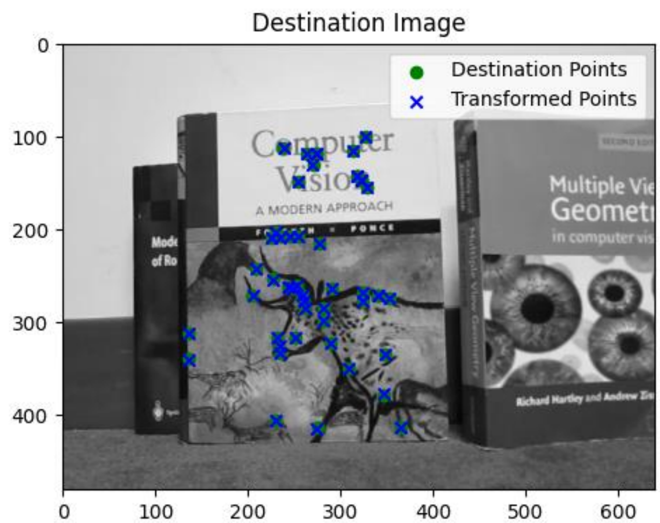
Top 50 Matches



- **Homography Computation** (`compute_homography`):
  - **Purpose**: Computes the homography matrix HHH that relates two sets of corresponding points between two images. This matrix is used for tasks such as image alignment or perspective transformation.
  - **Steps**:
    - Validate that at least 4 pairs of corresponding points are provided.
    - Build a matrix A based on the point correspondences.
    - Solve for H using Singular Value Decomposition (SVD) of matrix A.
    - Normalize H so that H[2,2]=1H[2, 2] = 1H[2,2]=1.
- **Homography Verification** (`verify_homography`):
  - **Purpose**: Verifies the computed homography by applying it to the source points and checking if they map correctly to the destination points.
  - **Steps**:
    - Convert source points to homogeneous coordinates.
    - Apply the homography matrix to transform the source points.
    - Normalize the result to non-homogeneous coordinates.
- **Point Extraction from Matches** (`get_points_from_matches`):
  - **Purpose**: Extracts the corresponding points from the keypoints of the matched descriptors.
  - **Steps**:
    - For each good match, retrieve the source and destination points from `keypoints1` and `keypoints2`.
- **Homography Matrix Calculation**:
  - After extracting the best 50 matches using Lowe's ratio test, the source and destination points are passed to the `compute_homography` function to compute the homography matrix HHH.
- **Homography Verification**:
  - The transformed points, obtained by applying H to the source points, are compared to the original destination points to verify the accuracy of the homography matrix.

- **Visualization** (`visualize_homography`):
  - **Purpose**: Displays the source image with the original points and the destination image with the corresponding points, along with the transformed points.
  - **Steps**:
    - Use `matplotlib` to create a side-by-side visualization.
    - Display the source points in red, destination points in green, and the transformed points in blue (with 'x' markers).





- **Mapping Book Corners** (**`map_book_corners`**):
  - **Purpose**: Maps the four corners of the book image to the video frame using the computed homography matrix H.
  - **Steps**:
    - Retrieve the dimensions (`h`, `w`) of the book image.
    - Define the four corners of the book image in Cartesian coordinates (`[0, 0]`, `[w, 0]`, `[w, h]`, `[0, h]`).
    - Convert these corners to homogeneous coordinates by adding a third coordinate with a value of 1 (i.e., `x, y, 1`).

    - Apply the homography matrix H to the corners, resulting in the mapped corners in homogeneous coordinates.
    - Convert the mapped corners back to Cartesian coordinates by dividing by the third coordinate to normalize them.
  - **Output**: Returns the mapped corner points in the video frame.

- **Visualizing Mapped Corners (`visualize_book_corners`)**:

  - **Purpose**: Displays the mapped corners on the video frame to visualize the alignment of the book image in the video frame.
  - **Steps**:
    - Create a copy of the video frame to avoid modifying the original.
    - For each mapped corner, draw a small circle (`cv2.circle`) at the corresponding coordinates.
    - Use Matplotlib to display the frame with the corners overlaid on the video.

Book Corners in Video Frame

- `calculate_book_dimensions` **Function**:

  - **Purpose**: Calculates the width and height of the book in the video frame using the mapped corners of the book.
  - **Steps**:
    - The width is calculated as the Euclidean distance between the top-left corner and the top-right corner.
    - The height is calculated as the distance between the top-left corner and the bottom-left corner.
    - The dimensions are returned as integers (book width and height).

- `crop_ar_frame_to_fit_book` **Function**:

  - **Purpose**: Crops and resizes the AR video frame to fit the book's aspect ratio, removing any unnecessary padding.
  - **Steps**:
    - Converts the AR frame to grayscale and creates a binary mask to detect non-black regions.
    - Identifies the rows with non-zero pixels (i.e., the content area).
    - Crops the frame by selecting the region that contains non-black pixels.
    - Resizes the cropped frame to match the aspect ratio of the book image, either cropping the left/right or top/bottom depending on the aspect ratio comparison.

- `warp_frame_to_book` **Function**:

  - **Purpose**: Warps the cropped AR frame to align with the book's area in the video frame using a perspective transformation.
  - **Steps**:
    - Defines the source points (corners of the cropped AR frame) and destination points (mapped corners of the book).
    - Computes the perspective transformation matrix using `cv2.getPerspectiveTransform`.
    - Applies the transformation to warp the AR frame to the book's position in the video.
    - Creates a mask for the book area and blends the warped AR frame with the original video frame, ensuring the AR content is only visible within the book's region.

- `process_ar_video` **Function**:

  - **Purpose**: Main function to process the AR video and overlay the AR content onto the book area in the input video.
  - **Steps**:
    - Loads both the AR video and the regular video frames.
    - Calculates the dimensions of the book in the video.
    - Initializes a video writer to output the processed video.
    - Uses SIFT to detect and compute keypoints and descriptors for the book image and the video frame.
    - Matches the descriptors between the book and the video frame using the BFMatcher with Lowe’s ratio test.
    - Computes the homography matrix to map the book’s corners in the video frame.
    - Crops and resizes the AR frame to fit the book’s dimensions and aspect ratio.
    - Warps the AR frame onto the book area in the video using the computed homography.
    - Writes the final result to the output video.
    - Releases the video and AR video resources after processing is complete.

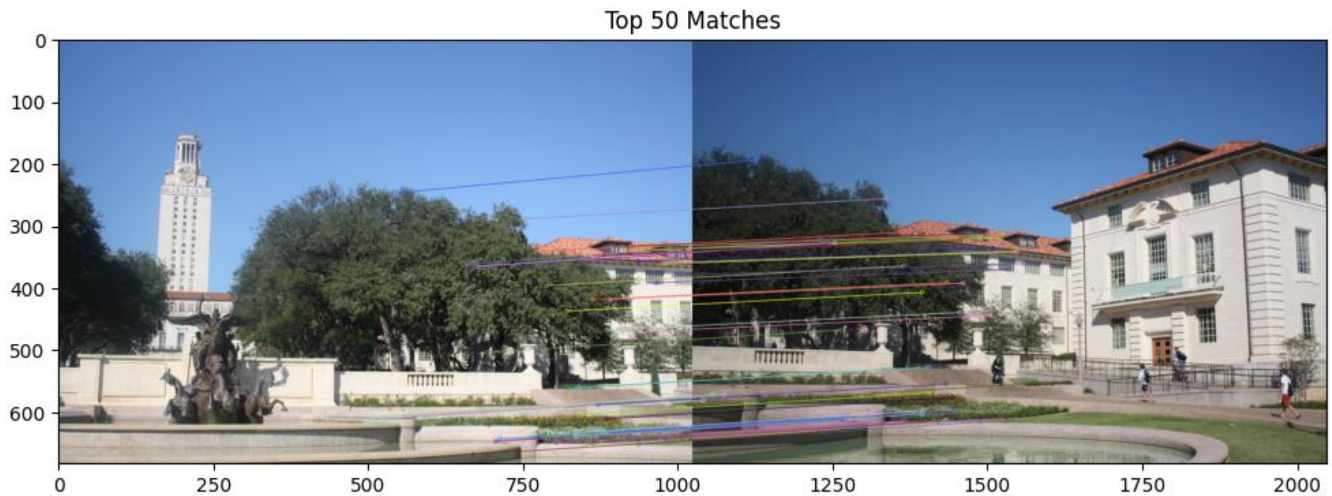
2. Part 2: Image Mosaics

2.1 Code Explanation

- **SIFT Feature Detection**:
  - `sift = cv2.SIFT_create()` initializes a SIFT detector.
  - `keypoints1, descriptors1 = sift.detectAndCompute(image1, None)` detects keypoints and computes descriptors for the first image (`image1`).
  - Similarly, `keypoints2, descriptors2 = sift.detectAndCompute(image2, None)` does the same for the second image (`image2`).
- **Brute-Force Matching with KNN**:
  - `bf = cv2.BFMatcher()` initializes a brute-force matcher.
  - `matches = bf.knnMatch(descriptors1, descriptors2, k=2)` finds the best 2 matches for each descriptor in the first image from the second image descriptors using K-nearest neighbor (KNN).
- **Lowe's Ratio Test**:
  - Lowe's ratio test is applied to filter out weak matches. The test ensures that the closest match is significantly better than the second closest match by checking if `m.distance < 0.75 * n.distance`.
- **Top 50 Matches**:
  - After applying Lowe’s ratio test, the matches are sorted by distance (`good_matches = sorted(good_matches, key=lambda x: x.distance)[:50]`), and the top 50 matches are selected.
- **Correspondences**:
  - The corresponding points between the two images are extracted by accessing the points from the keypoints using `match.queryIdx` and `match.trainIdx` for the first and second images, respectively.
  - These correspondences are stored in a list of tuples.

- **Visualization**:

  - The `cv2.drawMatches()` function is used to draw the good matches between the two images.
  - `plt.imshow(cv2.cvtColor(matched_image, cv2.COLOR_BGR2RGB))` displays the matched image using `matplotlib`, converting it from BGR to RGB color format for proper display.



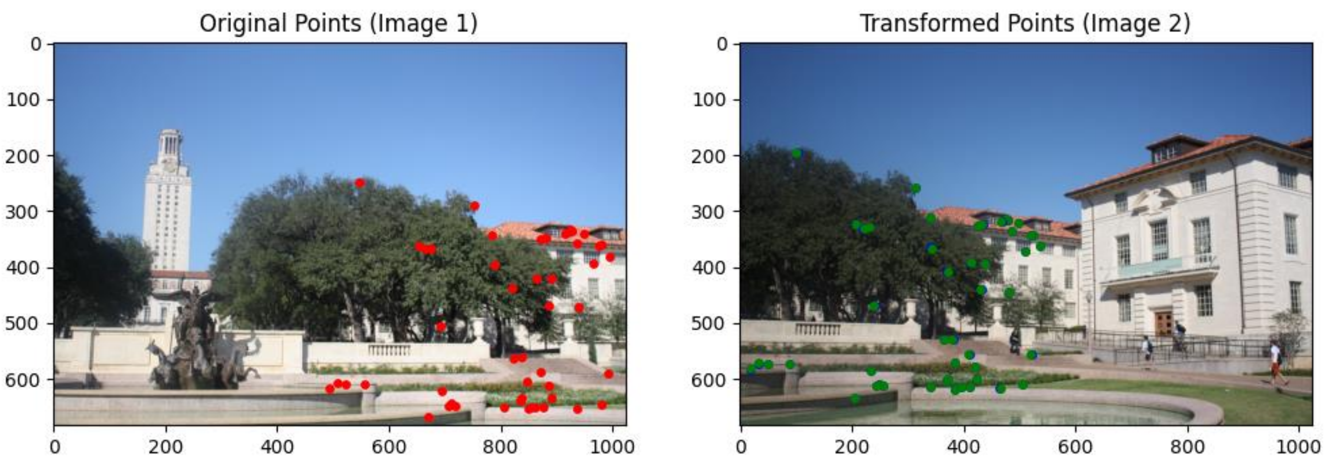
- **`compute_homography(correspondences)`**

  - **Function:** Computes the 3x3 homography matrix `H` from point correspondences.
    - Constructs the linear system `A` for the Direct Linear Transform (DLT) algorithm.
    - Solves for `H` using Singular Value Decomposition (SVD).
    - Normalizes `H` to ensure the last element is 1.

- **`apply_homography(H, points)`**

- **Function:** Applies a homography matrix to transform a set of points.
  - Converts points to homogeneous coordinates.
  - Multiplies each point by `H`.
  - Normalizes the results back to Cartesian coordinates.

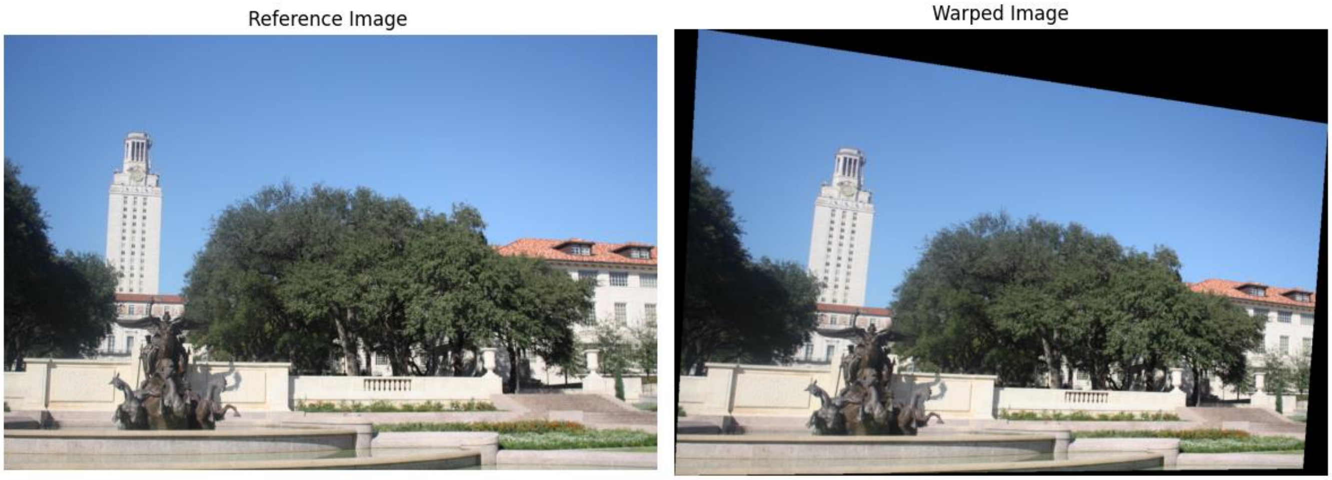
- `verify_homography(image1, image2, correspondences, H)`
  - **Function:** Visualizes the accuracy of the computed homography matrix.
    - Extracts matched points from both images.
    - Transforms points from the first image using `H`.
    - Displays original and transformed points on the respective images for comparison.



- `warp_image_combined(image, H, output_size)`
  - **Purpose:** Transforms an input image using forward and inverse homography warping and crops the final result to fit the warped region.
    - **Forward Warping:**
      - For each pixel in the source image:
        - Maps its position using the homography matrix H.
        - Places the pixel value at the computed location in the destination image.
      - Ensures valid destination coordinates fall within the output image bounds.
    - **Inverse Warping:**
      - For each pixel in the destination (output) image:
        - Maps it back to the source image using the inverse homography H^(-1).
        - Checks if the transformed coordinates lie within the bounds of the source image.
        - Applies bilinear interpolation to compute pixel values for smoother output.

- **Combining Results:**
  - Merges forward-warped and inverse-warped images into one.
  - Ensures non-zero values from inverse warping take precedence.
- **Cropping the Warped Region:**
  - Identifies non-zero pixels in the combined image.
  - Computes a bounding box around these pixels to crop the final result.
- **Visualization:**
  - Displays the input and cropped output images side by side for comparison.

Reference Image Warped Image

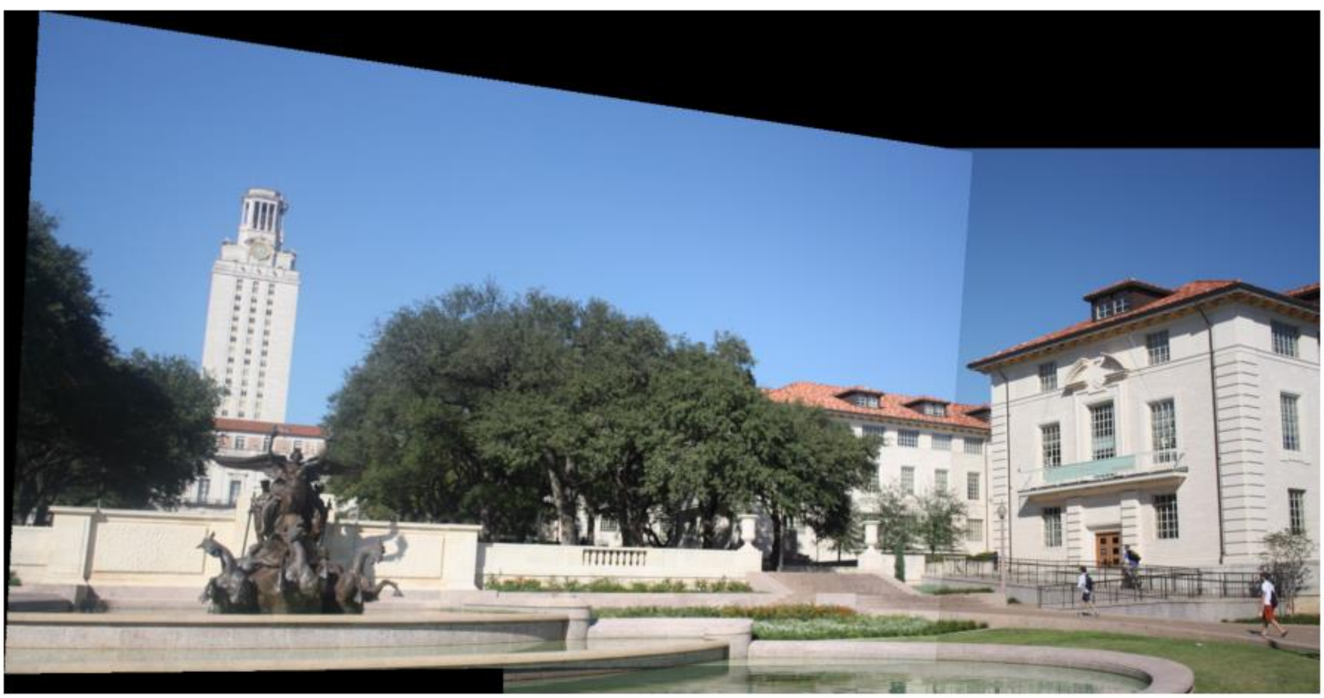


- `stitch_images(image2, image1, H)`

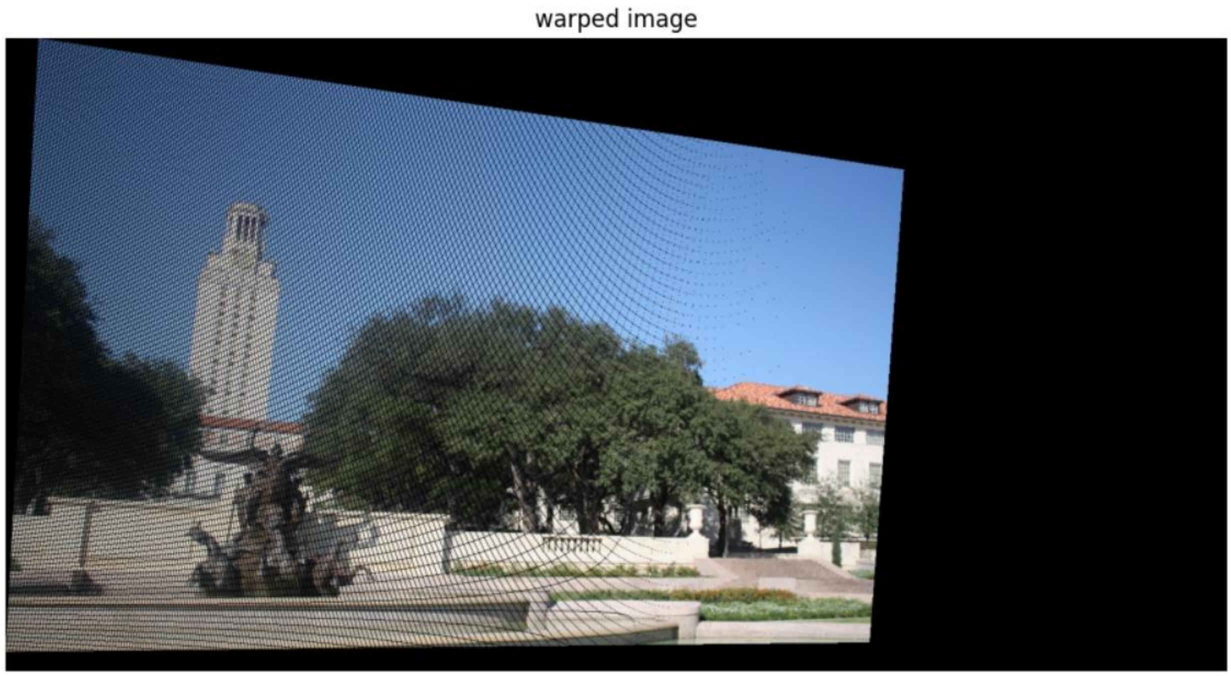
- **Purpose:** Combines two images into a seamless panorama using a given homography matrix.
  - **Defining Corners:**
    - Extracts corners of both images in their respective coordinate systems.
    - Computes the transformed coordinates of the second image's corners using H.
    - Normalizes the transformed corners back to Cartesian coordinates.
  - **Bounding Box Calculation:**
    - Combines the corners of both images to determine the minimum and maximum coordinates in the panorama.
    - Computes the panorama dimensions based on these bounds.

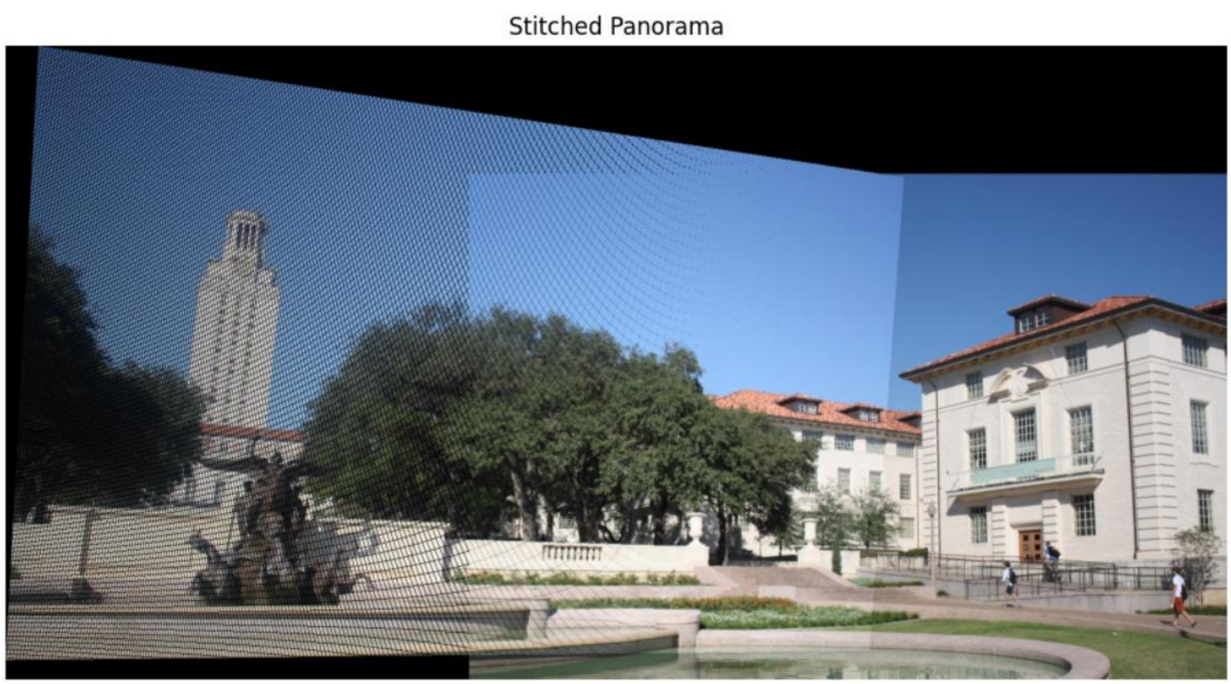
- **Translation Matrix:**
  - Creates a translation matrix to shift all coordinates into non-negative space.
- **Warping the Second Image:**
  - Calls `warp_image_combined` to warp the second image into the panorama space using T·H, where T is the translation matrix.
- **Canvas Initialization:**
  - Creates a blank canvas with the dimensions of the panorama.
  - Places the first image directly onto the canvas.
- **Overlaying Images:**
  - Identifies non-zero pixels in the warped second image.
  - Overlays these pixels onto the canvas, ensuring smooth blending in overlapping regions.
- **Output:**
  - Returns the combined panorama containing both images.

Stitched Panorama

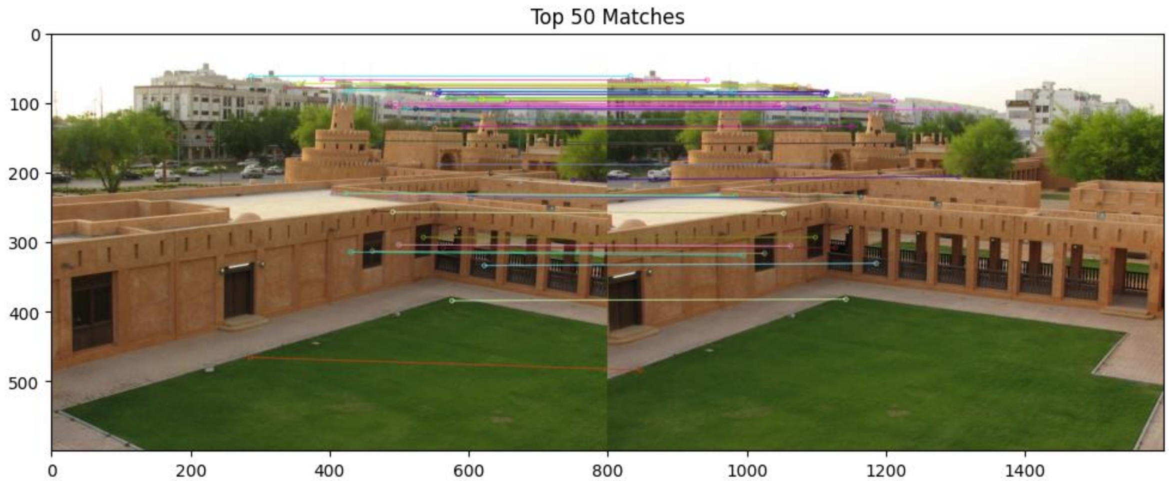


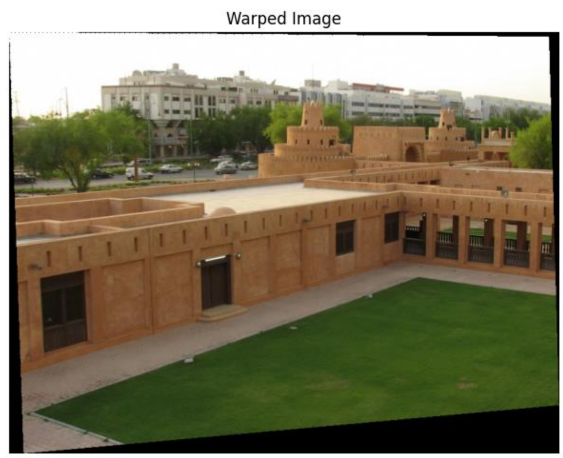
- **Limitation of Forward Warping Alone:**
  - If only forward warping were used:
    - Gaps (holes) would appear in the output due to uneven pixel mapping or rounding errors.
    - Some destination pixels might remain unassigned, creating artifacts in the warped image.

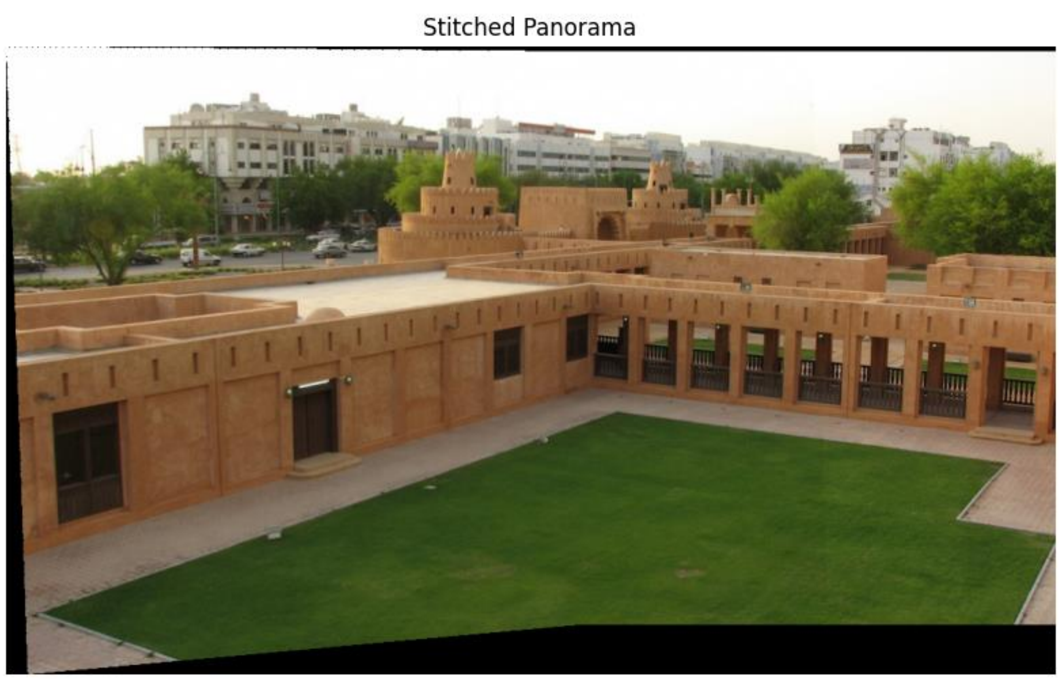




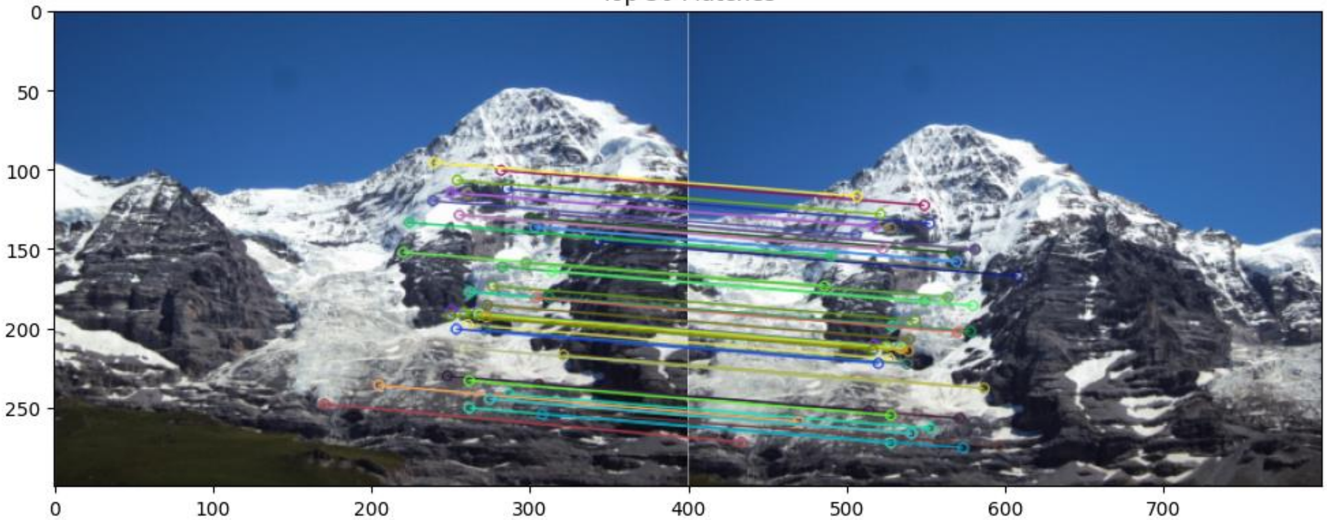
2.2 Other Examples





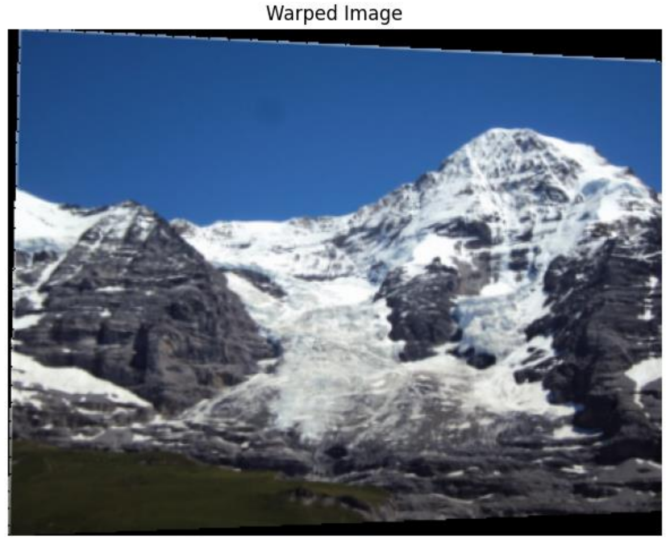


Top 50 Matches



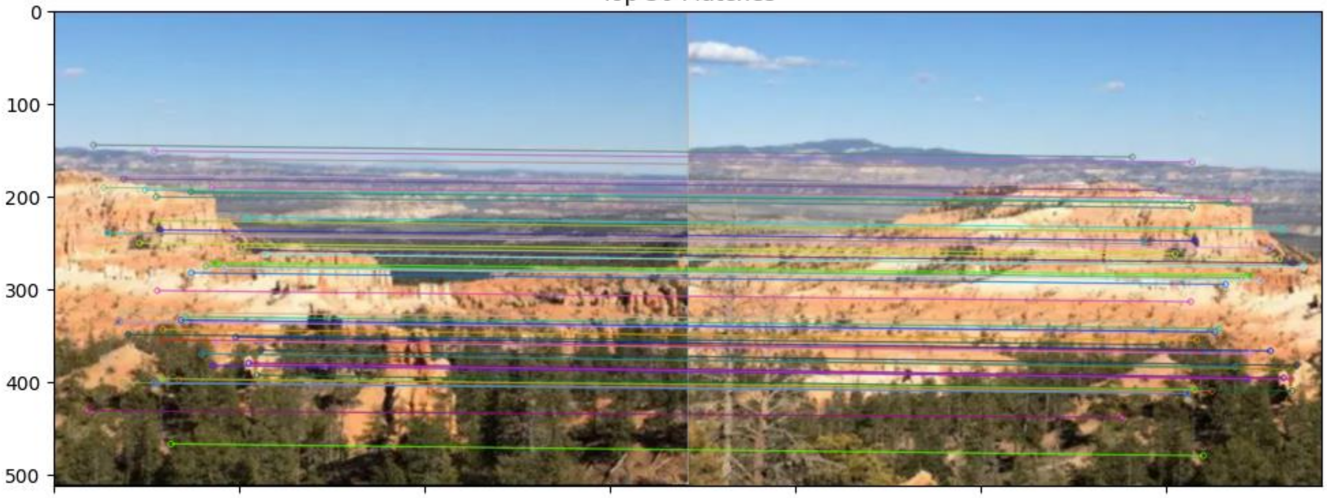
Reference Image

Warped Image

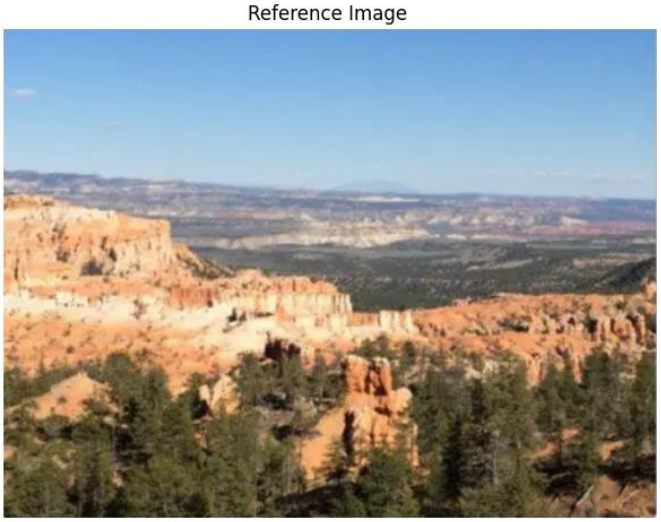


Stitched Panorama

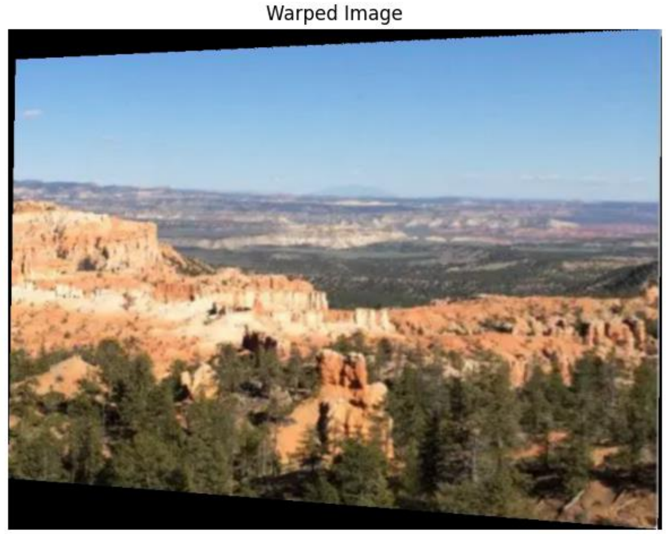
Top 50 Matches



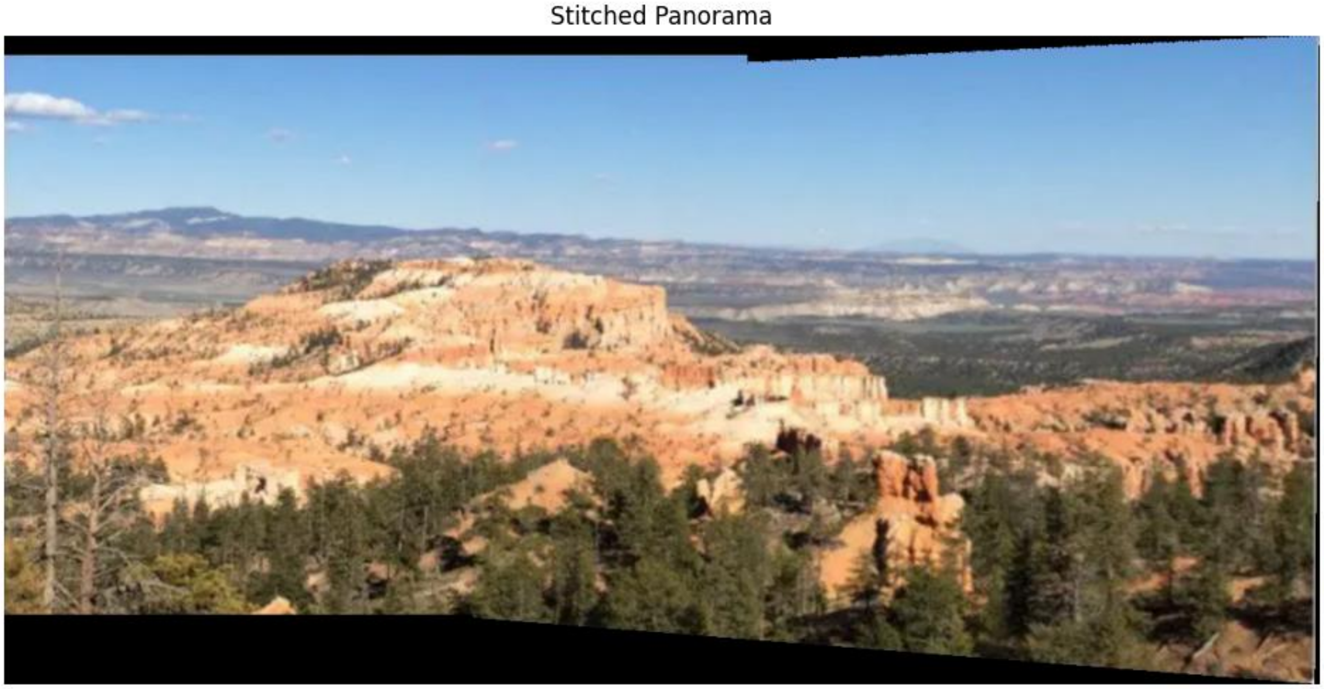
Reference Image



Warped Image



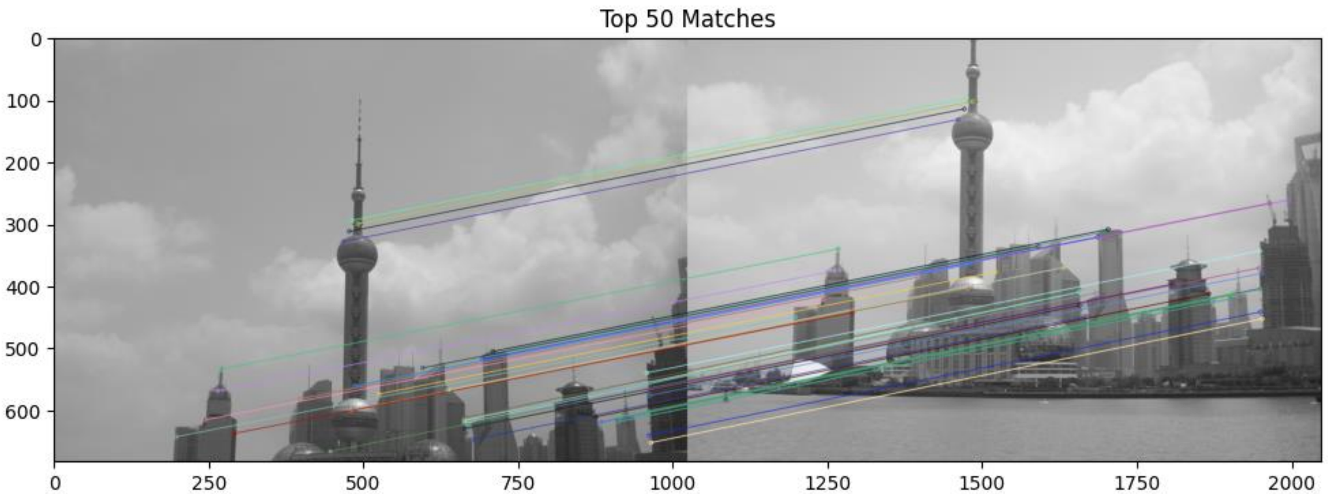
Stitched Panorama



3. Bonus: Stitching 3 images

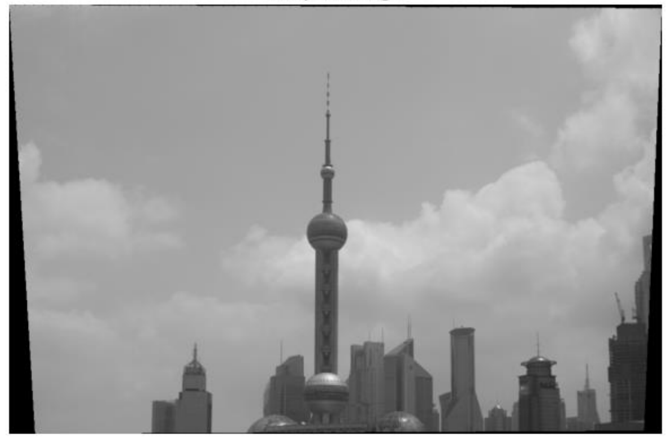
Code Explanation

- **Load Images**
    - **Image1**, **Image2**, and **Image3** are loaded using `cv2.imread`. These will be stitched sequentially.
- **Stitch `Image1` and `Image2`**
    - **Get Correspondences:** Use `get_correspondences(image2, image1)` to extract matching keypoints between `Image1` and `Image2`.
    - **Compute Homography:** Compute the transformation matrix H1→2 using `compute_homography`.
    - **Stitch:** Use `stitch_images(image2, image1, H_1_2)` to create the first panorama (`panorama_1_2`) by aligning `Image1` and `Image2` using the homography.
    - **Display Intermediate Result:** Visualize the stitched result for these two images.



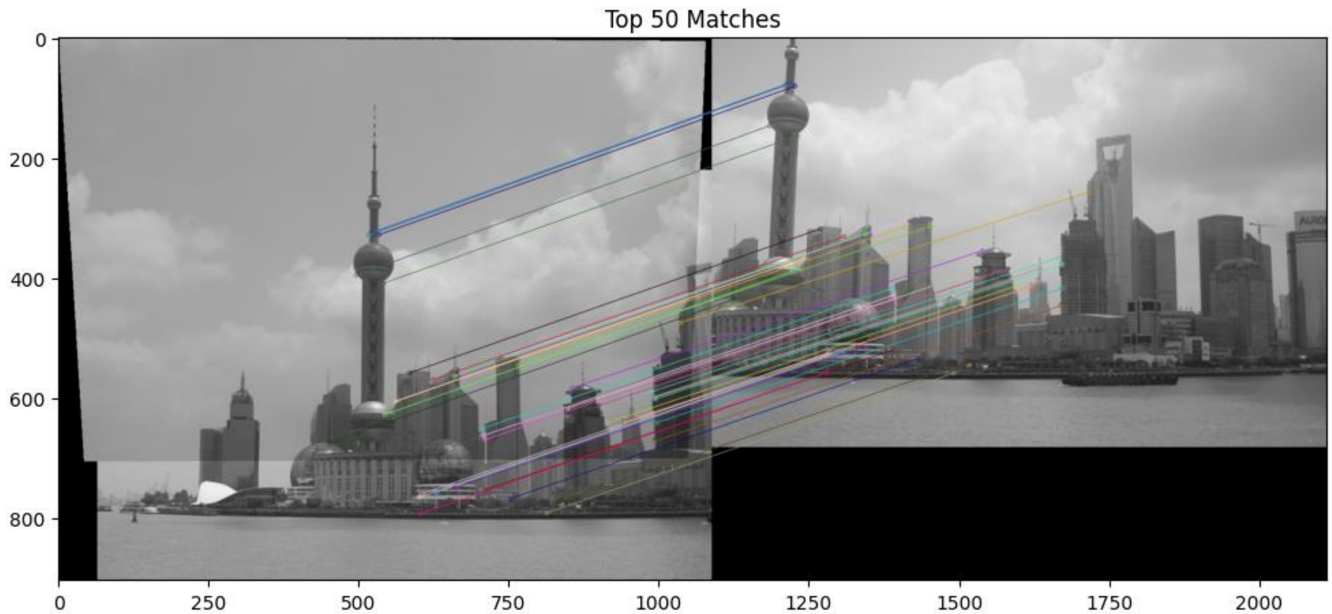
Reference Image

Warped Image



Stitched Panorama with first 2 Images

- **Stitch `panorama_1_2` and `Image3`**
    - **Get Correspondences:** Use `get_correspondences(panorama_1_2, image3)` to extract keypoints between the first panorama and `Image3`.
    - **Compute Homography:** Compute the transformation matrix Hpanorama→3.
    - **Stitch:** Use `stitch_images(panorama_1_2, image3, H_2_3)` to combine the first panorama and `Image3` into the final panorama (`panorama_with_image3`).
    - **Display Final Result:** Show the stitched result for all three images.



Final Stitched Panorama with 3 Images