IMAGE MOSAIC

ALIA ESSAM 7810
AYA RAGAA 7793

Code Analysis:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

- Module imports

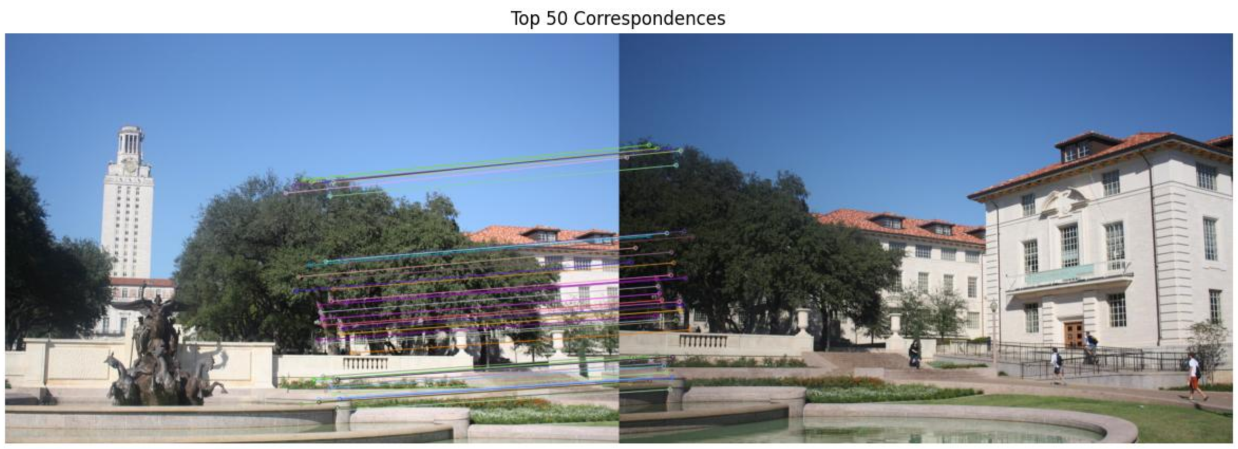
```
image1_path = "pano_image2.jpg"
image2_path = "pano_image1.jpg"
img1 = cv2.imread(image1_path, cv2.IMREAD_COLOR)
img2 = cv2.imread(image2_path, cv2.IMREAD_COLOR)
```

- Reading the images

```
def find_good_matches(img1, img2, ratio_threshold=0.5, top_n_matches=50):

    # Convert to grayscale (required for SIFT)
    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    # Initialize SIFT detector
    sift = cv2.SIFT_create()

    # Detect keypoints and descriptors
    keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

    # Use BFMatcher with k=2
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)
    matches = bf.knnMatch(descriptors1, descriptors2, k=2)

    # Apply ratio test as per Lowe's paper
    good_matches = []
    for m, n in matches:
        if m.distance < ratio_threshold * n.distance:
            good_matches.append(m)
        if len(good_matches) == top_n_matches:
            break

    return keypoints1 , keypoints2 , good_matches
```

- This function implements the main logic of feature detection, description, and matching.
- Converts the input images to grayscale because SIFT works on grayscale images.
- Detects keypoints (distinctive points in an image) and computes descriptors (feature vectors describing local image regions).
- Keypoints and descriptors are returned for both images.
- BFMatcher: Matches descriptors using L2 norm (Euclidean distance).
- crossCheck=False allows matching in one direction only.
- knnMatch(k=2): Finds the two best matches for each descriptor in descriptors1.
- Ratio Test: Retains a match if the closest match (m) is significantly better than the second-closest (n).
- Controlled by ratio_threshold=0.5.
- Limits to top_n_matches=50.
- Returns the keypoints for img1, img2, and the good_matches

```
keypoints1, keypoints2, good_matches = find_good_matches(img1, img2)

# To draw and display the matches:
img_matches = cv2.drawMatches(
    img1, keypoints1, img2, keypoints2, good_matches, None,
    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

plt.figure(figsize=(15, 7))
plt.imshow(cv2.cvtColor(img_matches, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Top 50 Correspondences')
plt.show()
```



- This part draws the matches between the 2 images using the detected keypoints for both images
- Then displays the drawn matches between the keypoints

```python
def compute_homography(points_src, points_dst):

    if(not(len(points_src) == len(points_dst) and len(points_src) >= 4)):
        print('Need at least 4 pair of points')
        return

    n = len(points_src)
    A = []

    for i in range(n):
        x, y = points_src[i]
        x_prime, y_prime = points_dst[i]
        A.append([-x, -y, -1, 0, 0, 0, x * x_prime, y * x_prime, x_prime])
        A.append([0, 0, 0, -x, -y, -1, x * y_prime, y * y_prime, y_prime])

    A = np.array(A)
    U, S, Vt = np.linalg.svd(A)
    h = Vt[-1, :]  # Solution is the last row of Vt
    H = h.reshape(3, 3)

    # Ensure H(3,3) is 1 by normalizing
    H /= H[2, 2]  # Normalize to make H(3,3) = 1

    return H
```

- This function computes the homography matrix (a 3x3 transformation matrix) from a set of point correspondences between the two images. The homography is used to transform points from one image to another, typically in tasks like image stitching or perspective correction.
- Returns the 3x3 homography matrix H that can map points from points_src to points_dst
- The function checks if the number of source points (points_src) is equal to the number of destination points (points_dst).
- It also ensures there are **at least 4 point pairs**. This is necessary because a homography requires at least 4 corresponding points to be computed (otherwise, the system would be underdetermined).

- For each pair of corresponding points ((x, y) in points_src and (x_prime, y_prime) in points_dst):
  Two equations are added to matrix A. These are derived from the homography constraints:
- Each point correspondence produces two equations for the transformation (one for the x coordinates and one for the y coordinates).
- Singular Value Decomposition (SVD) is used to solve the linear system of equations: np.linalg.svd(A) decomposes the matrix A into three matrices: U, S, and Vt.
- This method is preferred because it is numerically stable and helps in finding the least-squares solution to an overdetermined system (when n > 4).
- The solution to the system of equations corresponds to the **last row** of Vt.
- The last row of Vt represents the vector that minimizes the error in the least-squares sense.
- This row is reshaped into a 3x3 matrix H, which is the homography matrix.
- This ensures that the homography matrix is properly normalized by setting the bottom-right element (H[2, 2]) to 1. This is a typical normalization step to ensure the homography is in the correct format for transformations.

```python
def apply_homography(H, points):
    points_homogeneous = np.hstack([points, np.ones((points.shape[0], 1))])
    transformed =  np.dot(H, points_homogeneous.T)
    transformed /= transformed[2, :]  # Normalize by the third coordinate
    return transformed[:2, :].T
```

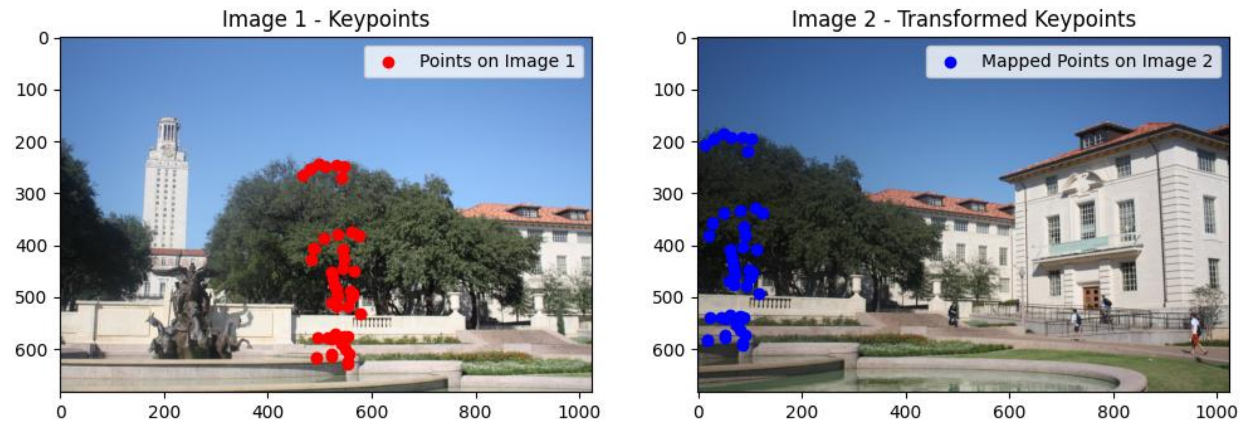
- Converts the 2D points into homogeneous coordinates by appending a 1 to each point.
- The homography matrix H is a 3x3 matrix, and points_homogeneous.T is the transpose of the points in homogeneous coordinates, which will be of shape (3, N).
- The dot product computes the transformed points in homogeneous coordinates.
- After this operation, transformed will have shape (3, N) and represent the transformed points in homogeneous coordinates.
- This step normalizes the transformed points by dividing each coordinate by the third coordinate (the homogeneous coordinate). This ensures that the last coordinate becomes 1 (making it back to standard 2D coordinates).
- After the normalization, the transformed points will be in the form [x', y', 1]

```python
# Extract the matched points
pts1 = np.float32([keypoints1[m.queryIdx].pt for m in good_matches])
pts2 = np.float32([keypoints2[m.trainIdx].pt for m in good_matches])

# Compute the homography matrix using the function
H = compute_homography(pts1, pts2)
print("Computed Homography Matrix:")
print(H)

# Apply the homography to transform points from the first image to the second
image
transformed_pts = apply_homography(H, pts1)

# Visualize the results: plot the points on both images
plt.figure(figsize=(12, 6))

# Show Image 1 with points
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB))
plt.scatter(pts1[:, 0], pts1[:, 1], color='red', label='Points on Image 1')
plt.title('Image 1 - Keypoints')
plt.legend()

# Show Image 2 with the transformed points (from Image 1)
plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB))
plt.scatter(transformed_pts[:, 0], transformed_pts[:, 1], color='blue',
label='Mapped Points on Image 2')
plt.title('Image 2 - Transformed Keypoints')
plt.legend()
plt.show()
```

- keypoints1 and keypoints2 are the keypoints detected in img1 and img2, respectively, typically using SIFT or another feature detector.
- good_matches contains the list of **valid matches** between the keypoints from img1 and img2. Each match is represented by an index pair queryIdx (index in img1) and trainIdx (index in img2).
- The code extracts the 2D coordinates of the matched keypoints (pt) from both keypoints1 and keypoints2 and stores them as floating-point NumPy arrays pts1 and pts2. These are the source and destination points for the homography.
- Then compute the homography matrix by calling the function above.

- For verification, we apply the homography matrix on the source points of 1st image and check if the output corresponds to the locations of the points from source image into destination image.

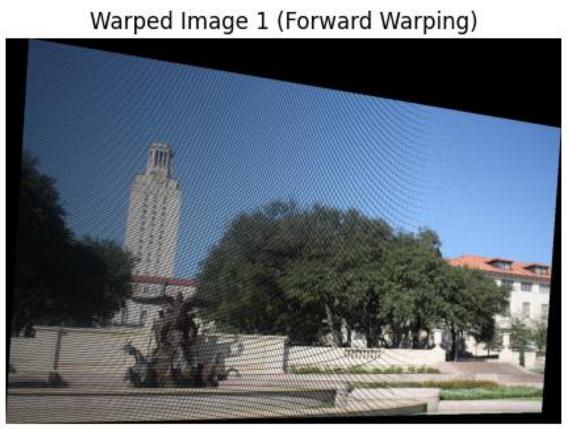
```
Computed Homography Matrix:
[[ 1.35059089e+00 -8.10199718e-02 -5.95823141e+02]
 [ 1.97898033e-01  1.25048024e+00 -1.84549652e+02]
 [ 3.49124112e-04  7.82145742e-06  1.00000000e+00]]
```



```
def forward_warp_image(image, H):

    # Get the dimensions of the input image
    height, width = image.shape[:2]

    # Compute the corners of the input image
    corners = np.array([[0, 0], [width - 1, 0], [width - 1, height - 1], [0, 
height - 1]], dtype=float)
    ones = np.ones((corners.shape[0], 1))
    corners_homogeneous = np.hstack([corners, ones])  # Convert to homogeneous 
coordinates

    # Apply the homography to the corners to find the bounding box of the warped 
image
    transformed_corners = np.dot(H, corners_homogeneous.T).T
    transformed_corners /= transformed_corners[:, 2].reshape(-1, 1)  # Normalize 
by the third coordinate

    # Get the min and max coordinates
    min_x = int(np.floor(min(transformed_corners[:, 0])))
    max_x = int(np.ceil(max(transformed_corners[:, 0])))
    min_y = int(np.floor(min(transformed_corners[:, 1])))
    max_y = int(np.ceil(max(transformed_corners[:, 1])))

    # Compute the dimensions of the new image
    output_width = max_x - min_x
    output_height = max_y - min_y

    # Create an empty output image with a black background
    warped_image = np.zeros((output_height, output_width, 3), dtype=image.dtype)

    # Iterate through each pixel in the input image and map it to the output
image
    for y in range(height):
        for x in range(width):
            # Input pixel in homogeneous coordinates
            input_pixel = np.array([x, y, 1])

            # Map the pixel using the homography matrix
            mapped_pixel = np.dot(H, input_pixel)
            mapped_pixel /= mapped_pixel[2]  # Normalize to get (x, y)
coordinates

            # Compute the coordinates in the output image
            new_x = int(mapped_pixel[0] - min_x)
            new_y = int(mapped_pixel[1] - min_y)

            # Ensure the new coordinates are within bounds
            if 0 <= new_x < output_width and 0 <= new_y < output_height:
                warped_image[new_y, new_x] = image[y, x]

    return warped_image


warped_img1 = forward_warp_image(img1, H)

# Show the original and warped images
plt.figure(figsize=(12, 6))

# Show the original Image 1
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB))
plt.title('Original Image 1')
plt.axis('off')

# Show the warped image with holes
```

```
plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(warped_img1, cv2.COLOR_BGR2RGB))
plt.title('Warped Image 1 (Forward Warping)')
plt.axis('off')

plt.show()
```

- The first part of the code computes the four corners of the image: top-left (0, 0), top-right (width-1, 0), bottom-right (width-1, height-1), and bottom-left (0, height-1).
- These are the corner coordinates of the original image in the Cartesian coordinate system.
- Then, a column of ones is added to convert these 2D coordinates to homogeneous coordinates (i.e., converting from (x, y) to (x, y, 1)).
- The **homography matrix** H is applied to the corners of the image. This transforms the corners from the original image's coordinate system to the new coordinate system as per the homography.
- The result, transformed_corners, gives the new (x, y) coordinates of the image corners after applying the homography.
- The coordinates are normalized by dividing by the third (homogeneous) coordinate, ensuring the output is in 2D Cartesian coordinates.
- The code computes the **bounding box** of the transformed image by finding the minimum and maximum x and y values from the transformed corners.
- This bounding box will define the size of the output warped image, ensuring that it fully contains the transformed image.
- The width and height of the output image are computed by subtracting the minimum x (left) from the maximum x (right), and similarly for y coordinates.
- These values define the size of the warped output image.
- The code then iterates over each pixel (x, y) in the original input image.
- Each pixel's coordinates are transformed using the homography matrix H to get the corresponding coordinates in the output (warped) image.
- After transforming the pixel coordinates, the result is normalized by dividing by the third coordinate (homogeneous normalization).
- The pixel's new location (new_x, new_y) is calculated by subtracting the minimum x and y values of the transformed corners (this centers the image at (0, 0)).
- The pixel is then mapped from the original image (image[y, x]) to the new warped image (warped_image[new_y, new_x]).
- A check is performed to ensure the new coordinates are within the bounds of the output image.



```python
def bilinear_interpolation(image, x, y):
    # Get the integer coordinates of the four surrounding pixels
    x1, y1 = int(x), int(y)
    x2, y2 = min(x1 + 1, image.shape[1] - 1), min(y1 + 1, image.shape[0] - 1)

    # Calculate the weights for interpolation
    dx = x - x1
    dy = y - y1

    # Get the pixel values for the four neighbors
    I11 = image[y1, x1]
    I12 = image[y1, x2]
    I21 = image[y2, x1]
    I22 = image[y2, x2]

    # Perform the interpolation
    interpolated_value = (1 - dx) * (1 - dy) * I11 + dx * (1 - dy) * I12 + (1 - 
dx) * dy * I21 + dx * dy * I22
    return interpolated_value
```

- The bilinear_interpolation function performs bilinear interpolation on a given image to estimate the color value of a point (x, y) that doesn't necessarily lie on a pixel grid. Bilinear interpolation is a method of interpolating values based on the values of the four nearest neighboring pixels.
- x1, y1 are the integer coordinates (floor values of x and y), representing the top-left pixel in the 2x2 grid surrounding the interpolation point.
- x2, y2 are the next coordinates (bottom-right of the 2x2 grid), calculated by adding 1 to x1, y1.

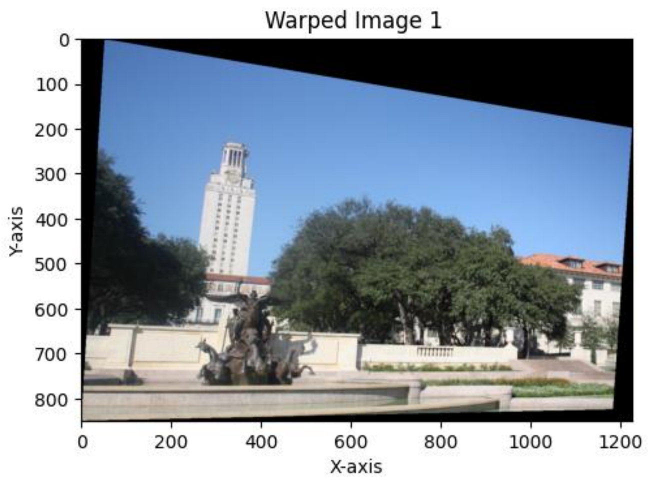
- min(x2, image.shape[1] - 1) ensures that x2 does not exceed the image width, and similarly for y2 (ensuring that it doesn't exceed the image height).
- These four coordinates ((x1, y1), (x2, y2)) represent the four neighboring pixels that will be used for interpolation.
- dx and dy are the **fractional parts** of the coordinates, representing how far the point (x, y) is from the integer coordinates (x1, y1). These values are used as interpolation weights.
- I11, I12, I21, and I22 are the pixel values at the four corners of the interpolation grid:
  I11 is the top-left pixel (x1, y1)
  I12 is the top-right pixel (x2, y1)
  I21 is the bottom-left pixel (x1, y2)
  I22 is the bottom-right pixel (x2, y2)
- These pixel values will be weighted based on how close the point (x, y) is to them.
- Then returns the interpolated pixel value

```
def inverse_warp_image(image, H):

    # Get the dimensions of the input image
    height, width = image.shape[:2]

    # Compute the corners of the image to calculate the required output size
    corners = np.array([[0, 0], [width - 1, 0], [width - 1, height - 1], [0,
height - 1]], dtype=float)
    ones = np.ones((corners.shape[0], 1))
    corners_homogeneous = np.hstack([corners, ones])  # Convert to homogeneous
coordinates

    # Apply the homography to the corners to find the bounding box of the warped
image
    transformed_corners = (np.dot(H, corners_homogeneous.T)).T
    transformed_corners /= transformed_corners[:, 2].reshape(-1, 1)  # Normalize
by the third coordinate

    # Get the min and max coordinates from the transformed corners
    min_x = min(transformed_corners[:, 0])
    max_x = max(transformed_corners[:, 0])
    min_y = min(transformed_corners[:, 1])
    max_y = max(transformed_corners[:, 1])

    # Compute the dimensions of the new image (bounding box)
    output_width = int(np.ceil(max_x - min_x))
    output_height = int(np.ceil(max_y - min_y))
    print(output_width,output_height)

    # Define the translation matrix to move the image so that the coordinates are
non-negative
    translation_matrix = np.array([[1, 0, -min_x], [0, 1, -min_y], [0, 0, 1]])

    # Apply the translation to the homography matrix
    H_translation = np.dot(translation_matrix, H)

    # Create an empty output image (target) with the new size
    warped_image = np.zeros((output_height, output_width, 3), dtype=np.uint8)

    # Inverse warping: For each pixel in the output image, map to the source
image using the inverse homography
    for i in range(output_height):
        for j in range(output_width):
            # Compute the corresponding point in the source image (inverse
homography)
            dst_coords = np.array([j, i, 1])  # Pixel coordinates in the
destination (output) image
            src_coords = np.dot(np.linalg.inv(H_translation) , dst_coords)  #
Apply inverse of homography
            # Normalize by the third coordinate
            src_coords /= src_coords[2]

            x_src, y_src = src_coords[0], src_coords[1]

            # Ensure the source coordinates are within bounds
            if 0 <= x_src < width - 1 and 0 <= y_src < height - 1:
                # Use bilinear interpolation to sample from the source image
                warped_image[i, j] = bilinear_interpolation(image, x_src, y_src)

    return warped_image


warped_img1_inverse = inverse_warp_image(img1, H)

# Show the original and warped images
plt.figure(figsize=(12, 6))

# Show the original Image 1
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB))
plt.title('Original Image 1')
plt.axis('off')
```

```
# Show the warped image
plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(warped_img1_inverse, cv2.COLOR_BGR2RGB))
plt.title('Warped Image 1 ')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

plt.show()
```

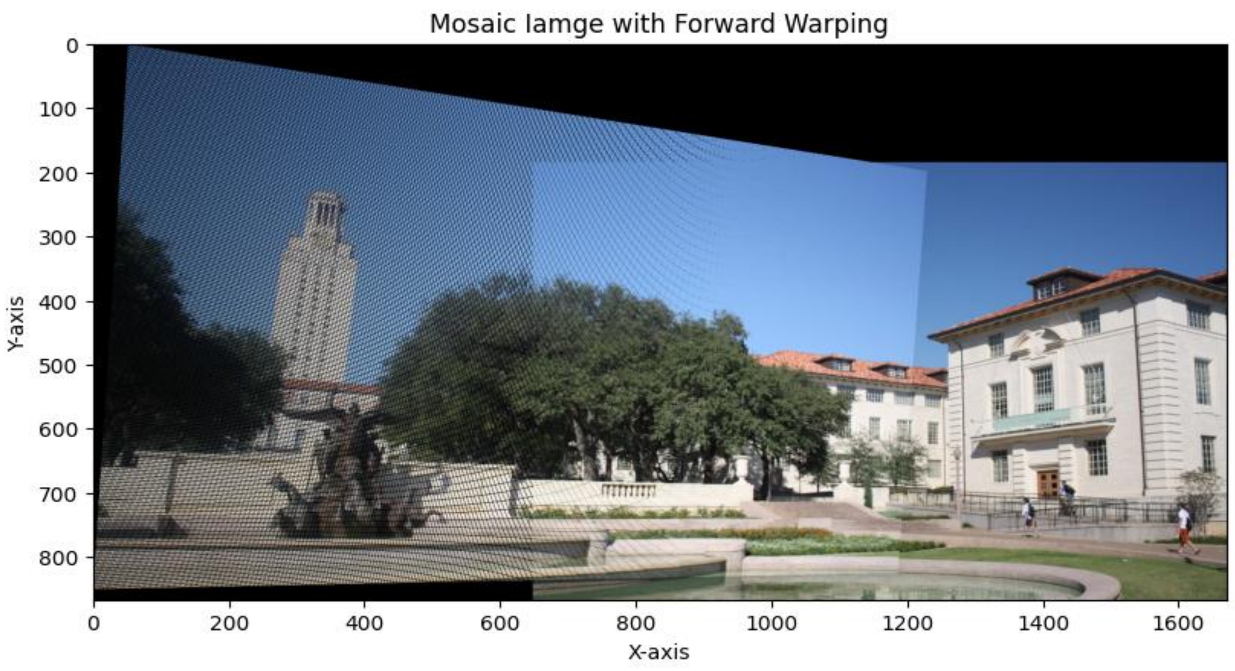
- height and width store the dimensions of the input image.
- The corners array stores the 4 corners of the image: top-left (0, 0), top-right (width-1, 0), bottom-right (width-1, height-1), and bottom-left (0, height-1).
- The ones array is appended to the corner points to convert them into homogeneous coordinates. This is necessary because homographies operate on homogeneous coordinates.
- The homography matrix H is applied to the 4 corner points of the image. This transformation maps the corners to new positions in the output space.
- The result is normalized by dividing the coordinates by the third component (to convert back from homogeneous to Cartesian coordinates).
- The bounding box around the transformed corners is calculated. This gives us the dimensions of the output image after the inverse warp.
- min_x, max_x, min_y, and max_y are the minimum and maximum x and y coordinates of the transformed corners.
- The output_width and output_height represent the size of the warped (output) image.
- A translation matrix is created to shift the output image such that all the transformed pixels fall within the non-negative range. This avoids any negative pixel coordinates in the output image.
- The translation matrix is multiplied by the homography H to get a new homography H_translation that includes the translation.
- • Inverse warping involves mapping each pixel in the output (destination) image to a corresponding pixel in the source image using the inverse of the homography matrix.
- • For each pixel (i, j) in the output image:
- The pixel coordinates in the output image (dst_coords) are represented in homogeneous coordinates.
- The inverse of the homography (np.linalg.inv(H_translation)) is applied to map the pixel coordinates from the output image back to the source image coordinates.
- The result is normalized by dividing by the third component to convert it back to Cartesian coordinates (x_src, y_src).

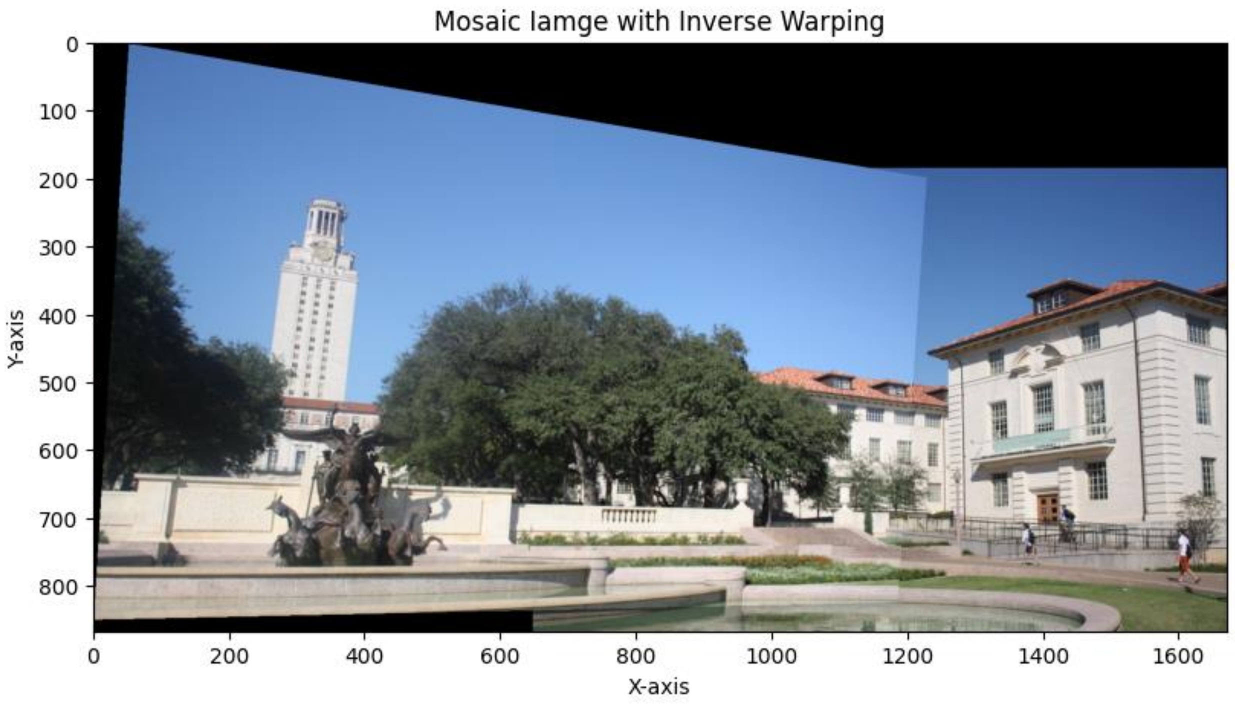
- Bilinear interpolation is used to sample the pixel value from the source image at (x_src, y_src) and assign it to the corresponding pixel in the output image.



```python
def merge_images(img1, img2, H):
    # Get the dimensions of img1 and img2
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    # Calculate the bounding box of the transformed image (img1) using the
homography
    corners_img1 = np.array([[0, 0, 1], [w1, 0, 1], [0, h1, 1], [w1, h1, 1]])
    transformed_corners = H @ corners_img1.T
    transformed_corners /= transformed_corners[2, :]  # Normalize homogeneous
coordinates
    min_x1, min_y1 = np.min(transformed_corners[:2], axis=1)
    max_x1, max_y1 = np.max(transformed_corners[:2], axis=1)

    # Calculate the new canvas size
    min_x = min(0, min_x1)
    min_y = min(0, min_y1)
    max_x = max(w2, max_x1)
    max_y = max(h2, max_y1)

    # Initialize a blank canvas large enough to contain both images
    canvas_width = int(max_x - min_x)
    canvas_height = int(max_y - min_y)
    merged_img1 = np.zeros((canvas_height, canvas_width, 3), dtype=np.uint8)
    merged_img2 = np.zeros((canvas_height, canvas_width, 3), dtype=np.uint8)

    # Offset for the img2 (to fit into the new canvas)
    offset_x2 = int(-min_x)
    offset_y2 = int(-min_y)

    merged_img1[offset_y2:offset_y2 + h2, offset_x2:offset_x2 + w2] = img2
    merged_img2[offset_y2:offset_y2 + h2, offset_x2:offset_x2 + w2] = img2

    # Forward warp img1 into the new canvas
    warped_img1 = forward_warp_image( img1,H)
    warped_img2 = inverse_warp_image( img1,H)

    for y in range(warped_img1.shape[0]):
        for x in range(warped_img1.shape[1]):
            # Ensure coordinates are within canvas bounds
            if 0 <= y < canvas_height and 0 <= x < canvas_width:
                if np.any(warped_img1[y, x]):  # If the pixel is not black
                    merged_img1[y, x] = warped_img1[y, x]

    # Overlay the inverse warped image onto merged_img2
    for y in range(warped_img2.shape[0]):
        for x in range(warped_img2.shape[1]):
            if 0 <= y < canvas_height and 0 <= x < canvas_width:
                if np.any(warped_img2[y, x]):  # If the pixel is not black
                    merged_img2[y, x] = warped_img2[y, x]

    return merged_img1, merged_img2

merged_img1,merged_img2 = merge_images(img1, img2, H)

# Display the merged result
plt.figure(figsize=(10, 5))
plt.imshow(cv2.cvtColor(merged_img1, cv2.COLOR_BGR2RGB))
plt.title('Mosaic Iamge with Forward Warping')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()

plt.figure(figsize=(10, 5))
plt.imshow(cv2.cvtColor(merged_img2, cv2.COLOR_BGR2RGB))
plt.title('Mosaic Iamge with Inverse Warping')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()
```

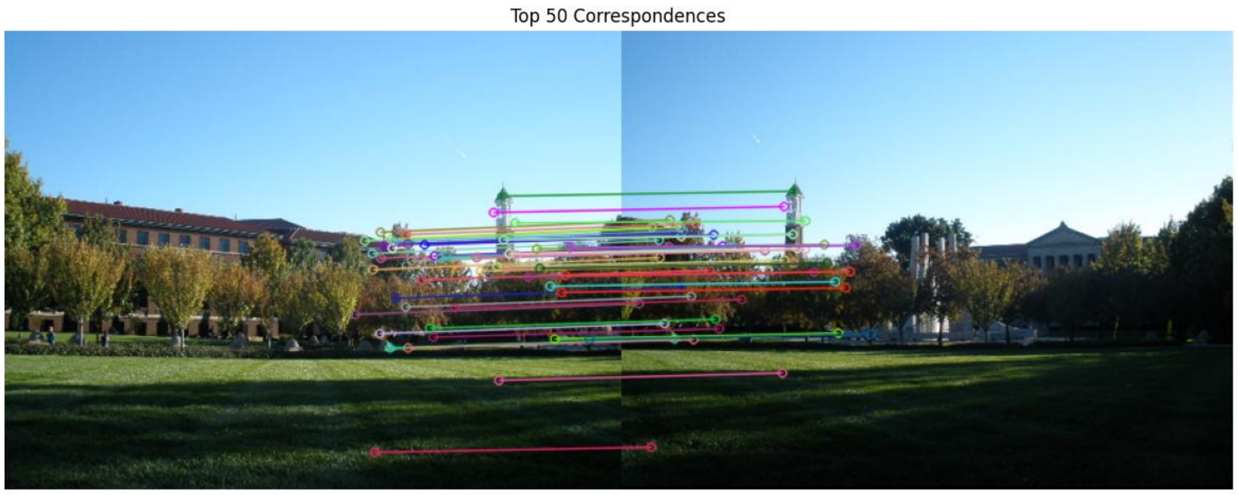
- The heights (h1, h2) and widths (w1, w2) of img1 and img2 are extracted. These are used later to calculate the dimensions of the merged canvas.
- The corners of img1 are represented in homogeneous coordinates.
- The homography matrix H is applied to these corners to get the transformed positions in the output space.
- The result is normalized by dividing by the third (homogeneous) coordinate.
- The minimum and maximum x and y coordinates of the transformed corners are computed, which defines the bounding box of the warped img1.
- The final canvas size is calculated to accommodate both images (img2 and the warped img1).
- The min_x, min_y, max_x, and max_y values represent the smallest and largest coordinates in both images, ensuring that the canvas will be large enough to contain both images after warping.
- The img2 image is offset by (min_x, min_y) to ensure it fits within the bounds of the canvas.
- img2 is placed in both merged_img1 and merged_img2. These will hold the final result using forward and inverse warping, respectively.
- Applied both forward and inverse warping
- For each pixel in the forward-warped image (warped_img1), if the pixel is not black (i.e., it contains valid information), it is placed into merged_img1 at the corresponding coordinates.
- Similarly, for each pixel in the inverse-warped image (warped_img2), if the pixel is valid, it is placed into merged_img2.

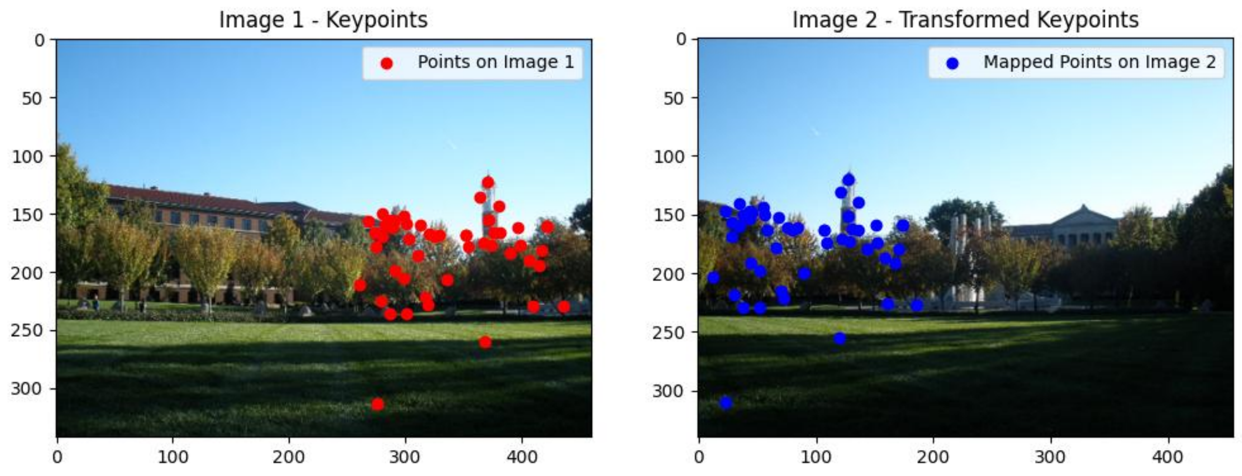

Mosaic Iamge with Forward Warping
Y-axis
X-axis
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000
1200
1400
1600

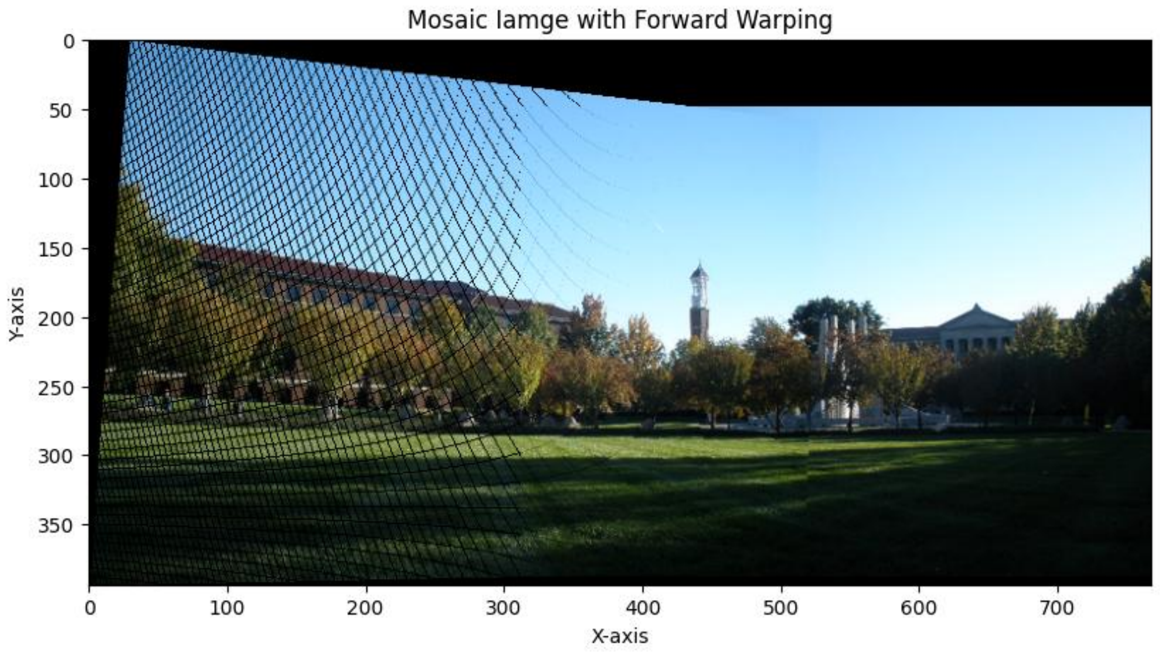

Mosaic Iamge with Inverse Warping
Y-axis
X-axis
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000
1200
1400
1600

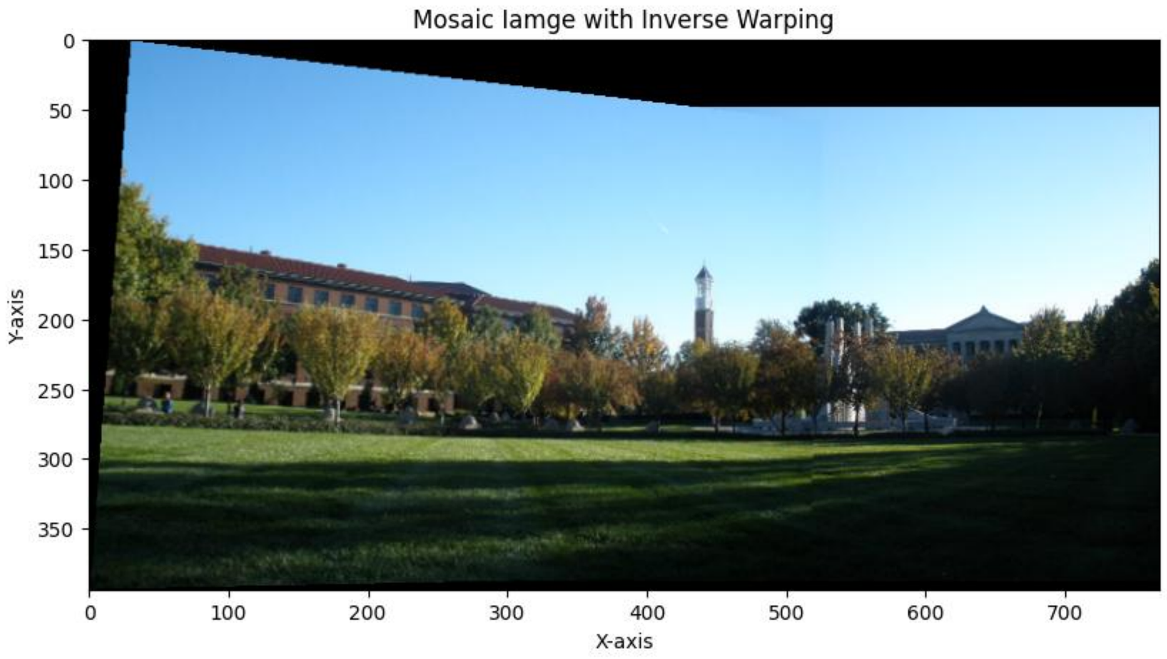
Another Example

- Calling the functions that are used above.





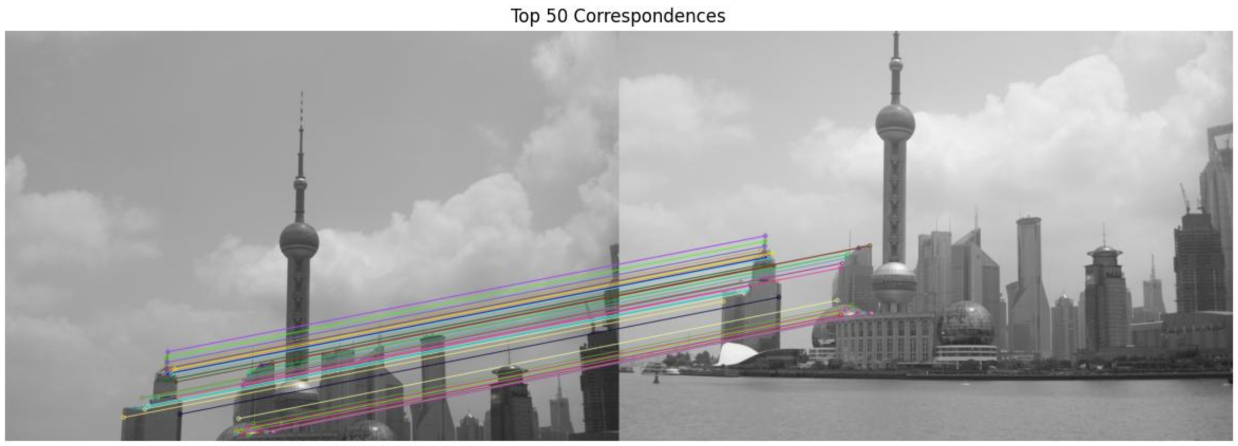

Mosaic Iamge with Forward Warping
0
50
100
150
200
250
300
350
Y-axis
0
100
200
300
400
500
600
700
X-axis

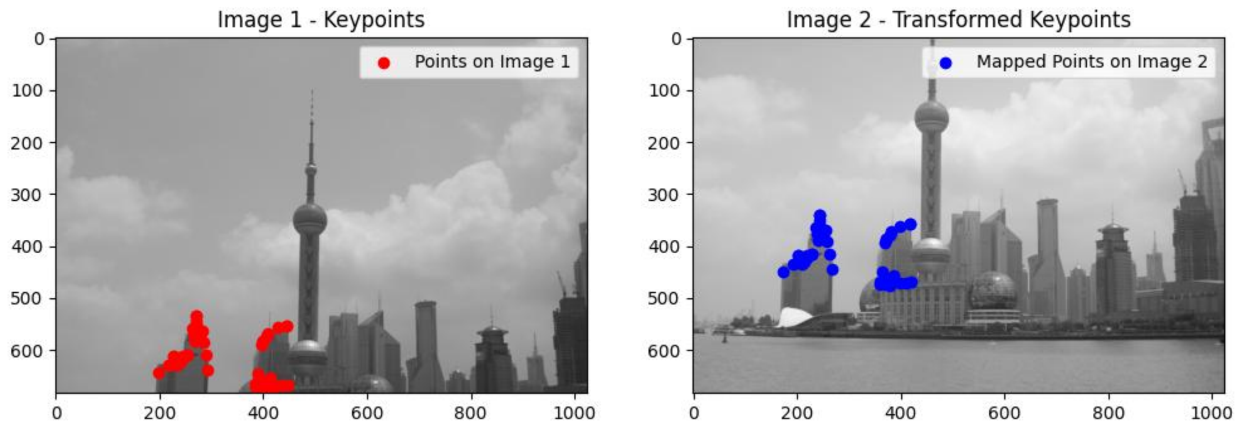

Mosaic Iamge with Inverse Warping
0
50
100
150
200
250
300
350
Y-axis
0
100
200
300
400
500
600
700
X-axis

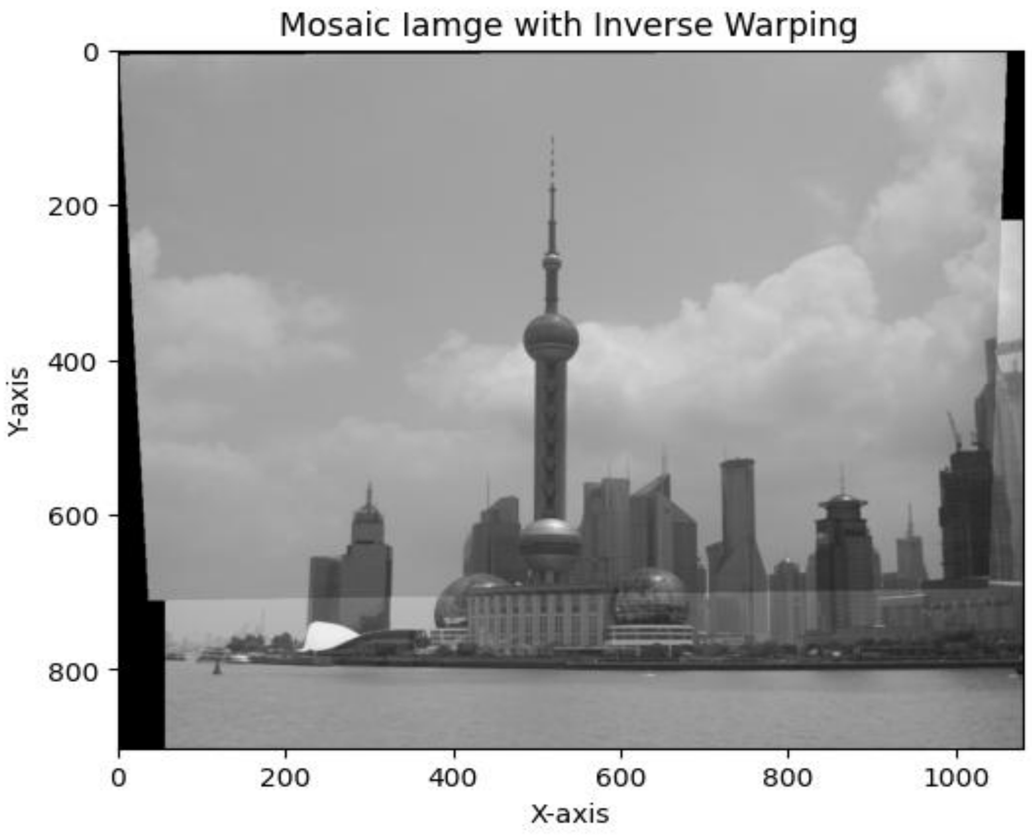
Bonus

- Calling the functions above
- We found the keypoints between the first 2 images and calculated the homography matrix for the source and destination points. We applied warping using the calculated homography matrix and got the first stitched images between the first 2 images.
- We then calculated the keypoints between the the stitched image above and the third image. We calculated the homography matrix and applied the warping. Finally, we stitched the previous stitched image with the third image.

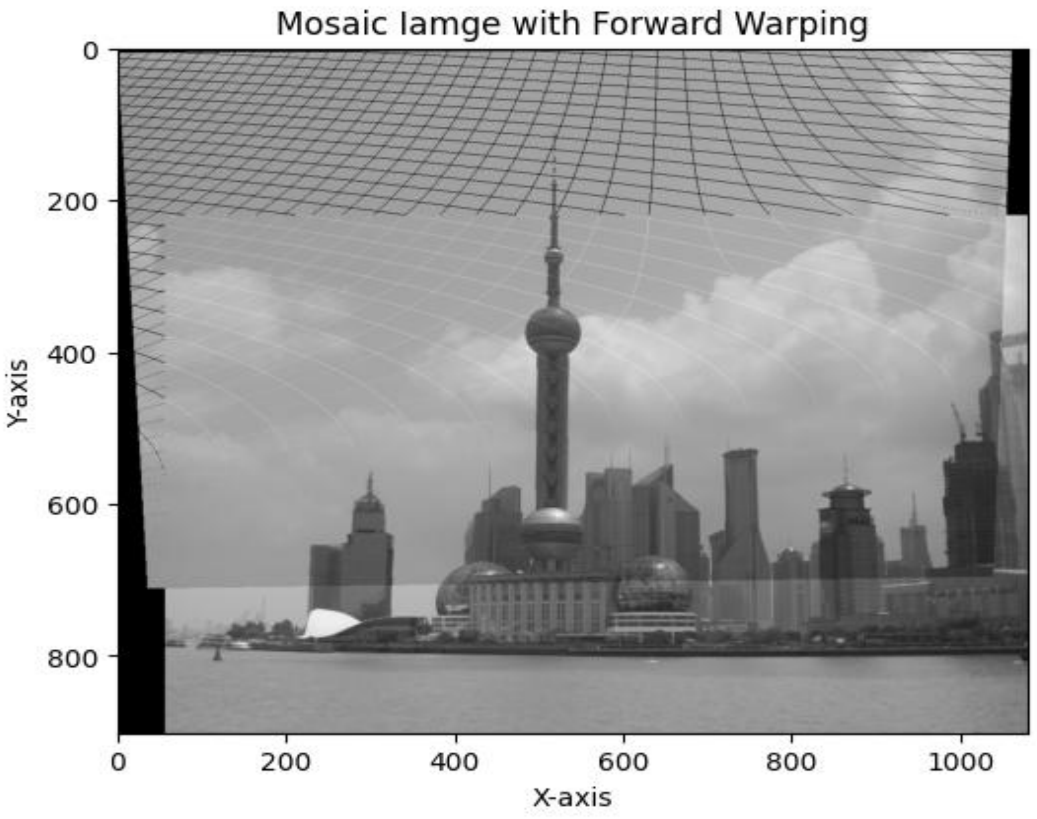
First 2 images:





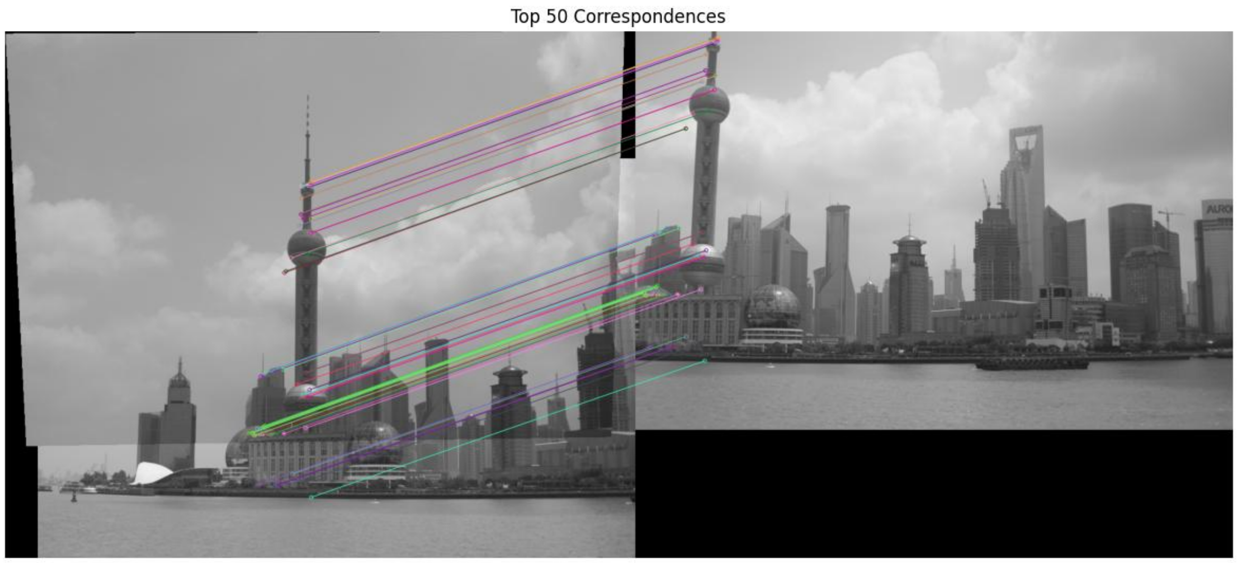


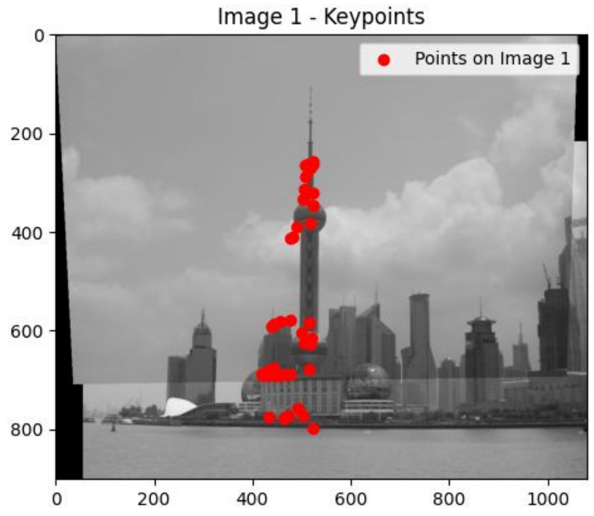
Mosaic Iamge with Inverse Warping
Y-axis
X-axis
0
200
400
600
800
1000

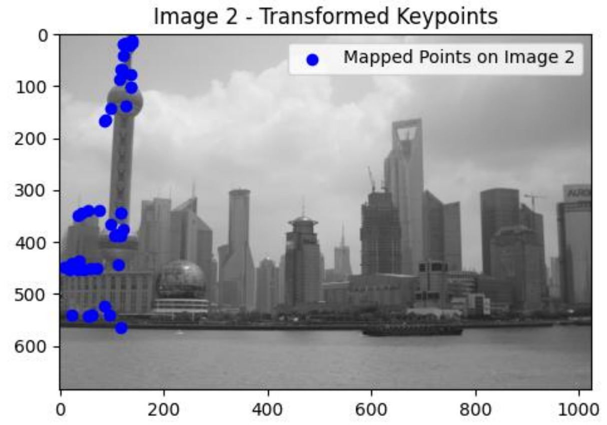


Mosaic Iamge with Forward Warping
Y-axis
X-axis
0
200
400
600
800
1000

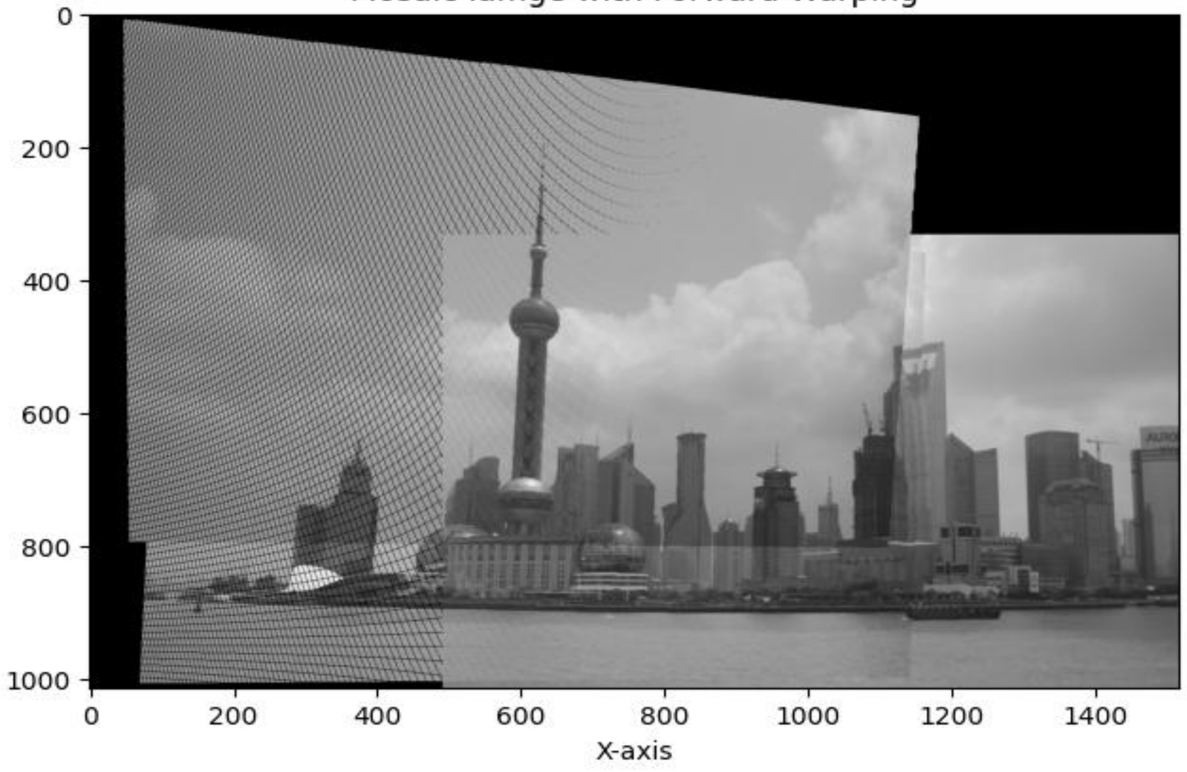
Stitched image with the 3rd image







Mosaic Iamge with Forward Warping



Mosaic Iamge with Inverse Warping