CV Lab 2

Youssef Hossam Aboelwafa	20012263
Ahmed Samir Sayed	20010102
Mohamed Raffeek Mohamed	20011574

Part 1: AR

1. `Finding Correspondences`

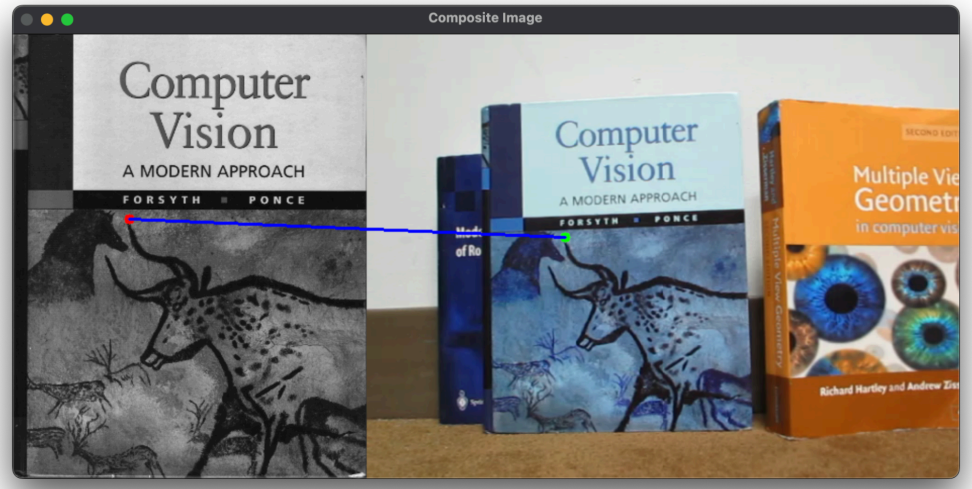
2. `compute_H(correspondences)`

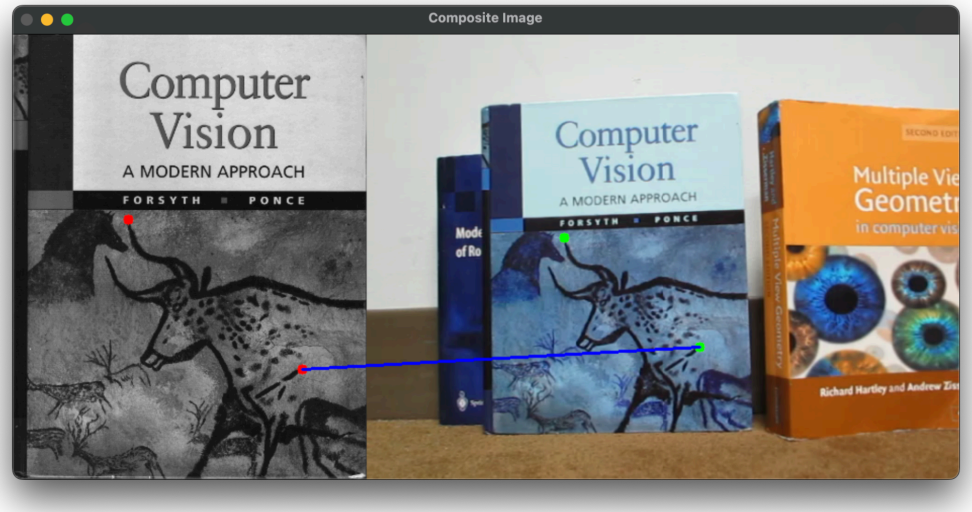
3. `RANSAC(correspondences)`

4. Validating H

Purpose: Make sure that the resulting homography matrix H correctly maps points from one image to another.

- Match the height of the two images by resizing one of them.
- Plot both images in one window.
- Use the homography matrix H to map clicked points on the first image to their locations on the second image.

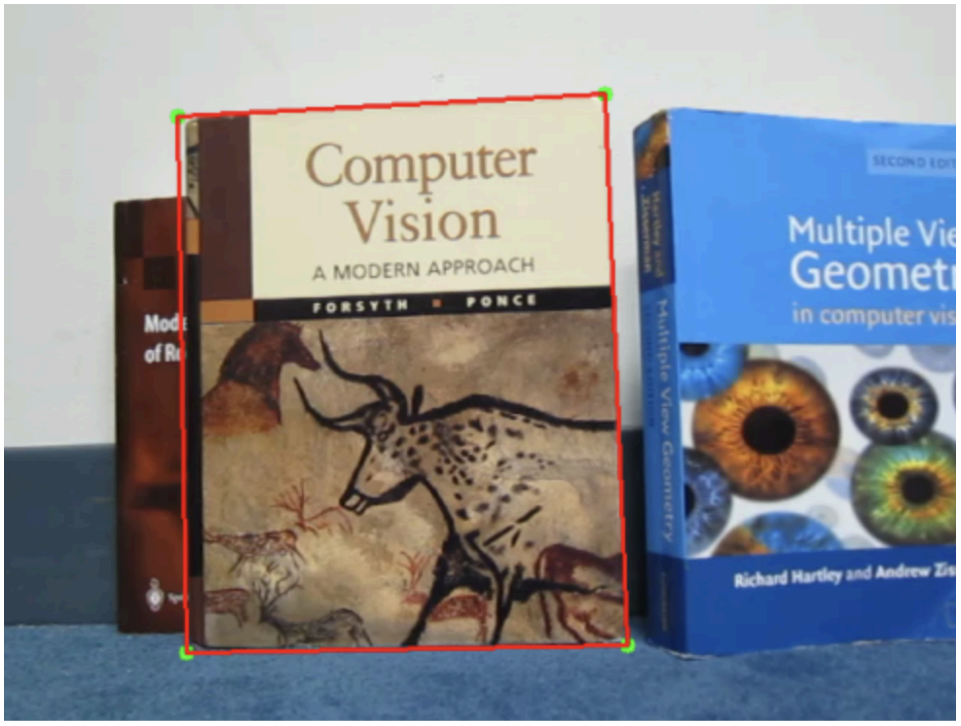




5. `get_video_book_corners(H, book_img)`

Purpose: Get the corners of the book in the video frame using the homography matrix.

- Define the corners of the book cover in the first image.
- Transform corners using the homography matrix to find their locations on the video frame.




```python
def get_video_book_corners(H, book_img):
    """
    Get the corners of the book in the video frame using the homography matrix.

    :param H: 3x3 Homography matrix.
    :param img1: First image (source).
    :param img2: Second image (destination).
    :return: List of points representing the corners of the book in the video frame.

    """ # Define the corners of the book cover in the first image
    h, w = book_img.shape[:2]
    corners = np.array([[0, 0], [w, 0], [w, h], [0, h]], dtype=np.float32).reshape(-1, 1, 2)
    transformed_corners = np.zeros((4, 1, 2), dtype=np.float32)

    # Transform corners using the homography matrix
    for i in range(4):
        corner = corners[i][0]
        # Map the corner point using the homography matrix
        transformed_corner = map_point(H, corner)
        transformed_corners[i][0] = transformed_corner

    return transformed_corners
```

6. `crop_video_frame(book_img, video_frame)`

Purpose: Crop the source video frame to fit onto the book cover in the book video frame. The frame is cropped such that only its central region is used in the overlaid frame.

- Define the book dimensions.
- Crop a sector from the center of the source video frame with the same dimensions as the book cover.

7. `overlay_frames(H, book_frame, cropped_frame, transformed_corners)`

Purpose: Overlay the cropped source video frame onto the book cover in the book video frame.

- Apply the homography matrix on the cropped frame to find the new coordinates in the new book video frame. (warping)
- Create two masks:
    - Foreground: the warped frame.
    - Background: the original frame.
- Combine the warped frame with the book frame using the masks.

```python
def crop_video_frame(book_img, video_frame):
    """
    Crop the video frame to fit onto the book cover in the book video
    frame. The frame is cropped such that only its central region is
    used in the overlayed frame.
    :param book_img: Book cover image.
    :param video_frame: Video frame.
    :return: Cropped video frame.

    """ # Get the dimensions of the book cover
    book_h, book_w = book_img.shape[:2]

    # Resize the video frame to 640x480
    video_frame = cv.resize(video_frame, (640, 480))

    # Get the dimensions of the video frame
    video_h, video_w = video_frame.shape[:2]

    # Calculate the cropping margins to center-crop the video frame
    top = max(0, (video_h - book_h) // 2)
    bottom = top + book_h
    left = max(0, (video_w - book_w) // 2)
    right = left + book_w

    # Crop the video frame
    cropped_video_frame = video_frame[top:bottom, left:right]

    return cropped_video_frame
```

```python
def overlay_frames(H, book_frame, cropped_frame, transformed_corners):
    """
    Overlay the cropped video frame onto the book cover in the book video frame.

    :param H: Homography matrix.
    :param book_frame: Book video frame (destination frame).
    :param cropped_frame: Cropped video frame (source fram
e). :param transformed_corners: Transformed corners of the book in the video frame.
    :return: Overlayed frame.

    """ # Convert transformed corners to a NumPy array
    transformed_corners = np.array(transformed_corners, dtype=np.float32)

    # Warp the cropped frame to fit the transformed corners
    warped_frame = cv.warpPerspective(cropped_frame, H, (book_frame.shape[1], book_frame.shape[0]))

    # Create a mask from the warped frame to blend it with the book frame
    mask = np.zeros_like(book_frame, dtype=np.uint8)
    cv.fillConvexPoly(mask, transformed_corners.astype(np.int32), (255, 255, 255))

    # Invert the mask to keep the original book frame where the warped frame is not present
    mask_inv = cv.bitwise_not(mask)

    # Combine the warped frame with the book frame
    book_frame_bg = cv.bitwise_and(book_frame, mask_inv)
    warped_frame_fg = cv.bitwise_and(warped_frame, mask)
    overlayed_frame = cv.add(book_frame_bg, warped_frame_fg)

    return overlayed_frame
```

7. Create the AR Video

- Read the book video.
- Compute the SIFT keypoints and descriptors for each frame.
- Find correspondences between the book cover and each book video frame using the descriptors.
- Compute the homography matrix for each book video frame given the correspondences.
- For each book video frame:
    - Crop the corresponding source video frame to fit the book cover
    - Overlay the cropped frame onto the book video frame
    - Append the resulting frame to a list
- Save the resulting frames as a video.

```python
# Each video frame descriptor is computed using the SIFT algorithm
_book_video = read_video('data/book.mov')
_video_kps = []
_video_des = []
for _frame in _book_video:
    _kp_frame, _des_frame = compute_SIFT_kps_and_descriptors(_frame)
    _video_kps.append(_kp_frame)
    _video_des.append(_des_frame)
```

```python
# Find correspondences between the book cover and each video frame
_matches = []
_book_cover_image = read_image('data/cv_cover.jpg')
_book_cover_gray = RGB2GRAY(_book_cover_image)
_book_cover_sift = cv.SIFT_create()
_book_cover_kps, _book_cover_des = _book_cover_sift.detectAndCompute(_book_cover_gray, None)
for i in range(len(_video_des)):
    _good = get_top_correspondences(_book_cover_des, _video_des[i], 50)
    _matches.append(_good)

print(len(_matches))
print(len(_book_video))
```

```python
# Compute the homography matrix for each video frame
_homographies = []
for i in range(len(_matches)):
    _correspondences =
        [_book_cover_kps[match[0].queryIdx].pt, _video_kps[i][match[0].trainIdx].pt]
        for match in _matches[i]
    ]
    _H = RANSAC(_correspondences)
    if i < 5:
        print(_H)
    _homographies.append(_H)

print(len(_homographies))

```

```python
# For each book video frame:
# 1. Crop the corresponding video frame to fit the book cover
# 2. Overlay the cropped frame onto the book video frame
# 3. Append the resulting frame to a list
_source_video = read_video('data/ar_source.mov')
_ar_frames = []
for i in range(len(_book_video)):
    # Crop the video frame
    _source_frame = _source_video[i%len(_source_video)]
    _cropped_frame = crop_video_frame(_book_cover_image, _source_frame)

    # Overlay the cropped frame onto the book video frame
    _transformed_corners = get_video_book_corners(_homographies[i], _book_cover_image)
    if i < 3:
        print(_transformed_corners)
    _overlayed_frame = overlay_frames(_homographies[i], _book_video[i], _cropped_frame, _transformed_corners)

    # Append the overlayed frame to the list
    _ar_frames.append(_overlayed_frame)
print(len(_ar_frames))

```

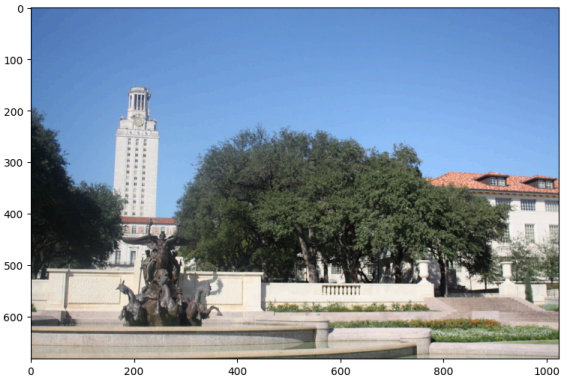
```python
# Save the augmented reality frames as a video with mov extension
fourcc = cv.VideoWriter_fourcc(*'mp4v')
out = cv.VideoWriter('data/ar_output.mov', fourcc, 30.0, (640, 480))
for frame in _ar_frames:
    rgb_frame = BGR2RGB(frame)
    out.write(rgb_frame)
out.release()
```

Part 2: Image Mosaics

1. `getMatches(img1, img2)`

Purpose: Finds matching keypoints between the two images using SIFT (Scale-Invariant Feature Transform).

- Converts both images to grayscale.
- Detects keypoints + computes descriptors with SIFT.
- Uses KNN matching with BFMatcher (brute-force) and ratio test to filter out bad matches.
    - **Number of matches : 4251**
    - **Number of good matches: 1063**
- Returns a list of point correspondences and an image showing the matches.



```python
def getMatches(img1, img2):

    img1_grey = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2_grey = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    sift = cv2.SIFT_create()

    keypoints_1, descriptors_1 = sift.detectAndCompute(img1_grey, None)
    keypoints_2, descriptors_2 = sift.detectAndCompute(img2_grey, None)

    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)
    matches = bf.knnMatch(descriptors_1, descriptors_2, k=2)
    # print(f"Number of matches: {len(matches)}")

    good_matches = [m1 for m1, m2 in matches if m1.distance < 0.75 * m2.distance]
    # print(f"Number of good matches: {len(good_matches)}")

    good_matches = sorted(good_matches, key=lambda x: x.distance)[:50]

    matched_img = cv2.drawMatches(
        img1_grey,
        keypoints_1,
        img2_grey,
        keypoints_2,
        good_matches,
        None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS,
    )

    correspondences = [
        [keypoints_1[match.queryIdx].pt, keypoints_2[match.trainIdx].pt]
        for match in good_matches
    ]

    return correspondences, matched_img
```

2. `compute_H(correspondences)`

Purpose: Computes the **homography matrix** HHH using **Direct Linear Transform (DLT)** from the provided point correspondences.

- Builds a system of equations `A`.
- Solves it using **SVD** (Singular Value Decomposition).
- Returns the 3x3 matrix `H`.

```python
def compute_H(correspondences):

    A = []
    for p, p_dash in correspondences:
        A.append(
            [-p[0], -p[1], -1, 0, 0, 0, p[0] * p_dash[0], p[1] * p_dash[0], p_dash[0]]
        )
        A.append(
            [0, 0, 0, -p[0], -p[1], -1, p[0] * p_dash[1], p[1] * p_dash[1], p_dash[1]]
        )

    A = np.array(A)
    U, D, V_transpose = svd(A)
    H = np.reshape(V_transpose[8], (3, 3))
    H /= H[2, 2]

    return H
```

3. `RANSAC(correspondences)`

Purpose: Makes the homography estimation **robust to outliers**.

- Repeats 500 times:
    - Randomly selects 4 correspondences.
    - Computes H from them.
    - Applies H to all points and measures error.
    - Keeps the transformation with the most inliers.
- Returns the final H from the best set of inliers.

```python
def RANSAC(correspondences):
    max_inliers = []

    for _ in range(500):

        random_indices = random.sample(range(0, len(correspondences)), 4)
        random_correspondences = [correspondences[i] for i in random_indices]

        H = compute_H(random_correspondences)

        curr_inliers = []
        for corr in correspondences:
            P = corr[0]
            mapped_P = tuple(
                map(
                    int,
                    (
                        np.dot(H, np.transpose([P[0], P[1], 1]))
                        / (np.dot(H, np.transpose([P[0], P[1], 1]))[2])
                    ).astype(int)[:2],
                )
            )
            e = np.linalg.norm(np.asarray(corr[1]) - np.asarray(mapped_P))
            if e < 5:
                curr_inliers.append(corr)

        if len(curr_inliers) > len(max_inliers):
            max_inliers = curr_inliers

    return compute_H(max_inliers)
```

4. `interpolation(x, y, image)`

Purpose: Performs **bilinear interpolation** to estimate a pixel value at non-integer coordinates `(x, y)`. **[Eliminate holes]**

- Weights pixel values of 4 neighboring pixels.
- Makes the warping smoother.

```python
def interpolation(x, y, image):
    x1 = math.floor(x)
    x2 = math.ceil(x)
    y1 = math.floor(y)
    y2 = math.ceil(y)
    w1 = (abs(y - y2)) * (abs(x - x2)) * image[y1][x1]
    w2 = (abs(y - y1)) * (abs(x - x1)) * image[y2][x2]
    w3 = (abs(y - y1)) * (abs(x - x2)) * image[y1][x2]
    w4 = (abs(y - y2)) * (abs(x - x1)) * image[y2][x1]
    return w1 + w2 + w3 + w4
```

5. `compute_p_dash(h, p, float=False)`

What it does:
Applies a homography transformation to a single 2D point.

How it works:

- Converts the point to homogeneous coordinates.
- Multiply it by the homography matrix.
- Converts it back to 2D by dividing by the third (homogeneous) coordinate.
- If `float=True`, returns fractional coordinates; otherwise, returns rounded integers.

```python
def compute_p_dash(h, p, float=False):
    p = np.transpose([p[0], p[1], 1])
    p_dash = np.dot(h, p)
    p_dash /= p_dash[2]
    if float == True:
        return (p_dash[0], p_dash[1])
    return (int(p_dash[0]), int(p_dash[1]))
```

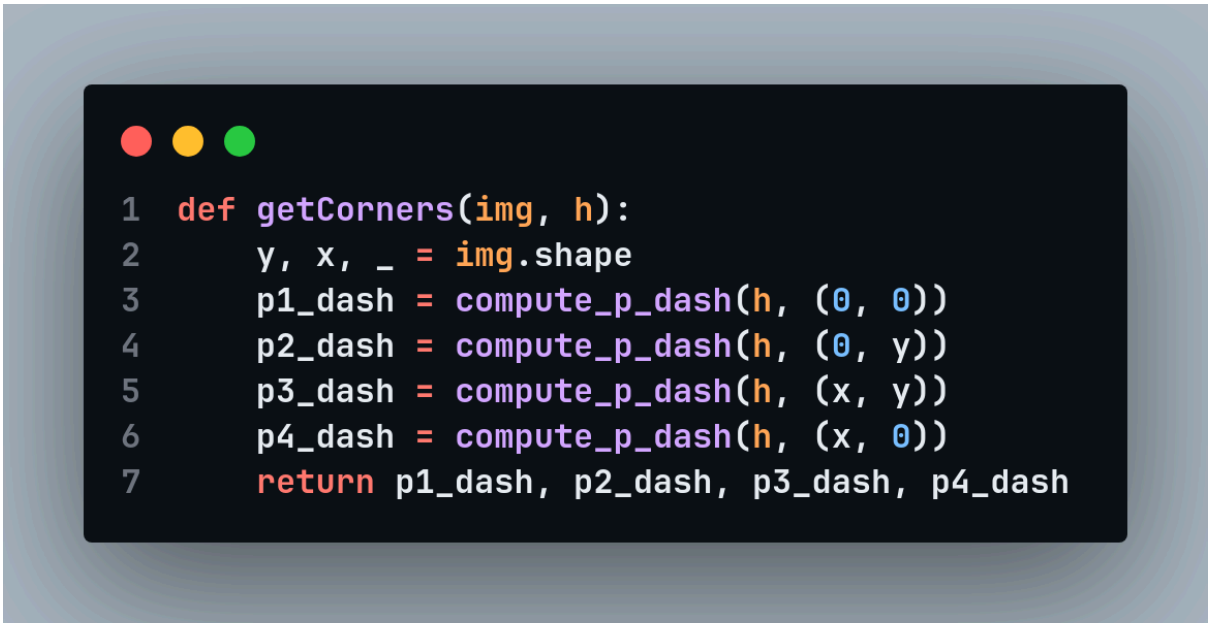
6. `getCorners(img, h)`

What it does:
Calculates where the four corners of an image land after applying a homography.

How it works:

- Applies `compute_p_dash` to all four corners of the input image (top-left, bottom-left, bottom-right, top-right).

Why it's important:
Helps determine the area that the warped image will occupy and where to place it on the canvas.



```python
def getCorners(img, h):
    y, x, _ = img.shape
    p1_dash = compute_p_dash(h, (0, 0))
    p2_dash = compute_p_dash(h, (0, y))
    p3_dash = compute_p_dash(h, (x, y))
    p4_dash = compute_p_dash(h, (x, 0))
    return p1_dash, p2_dash, p3_dash, p4_dash
```

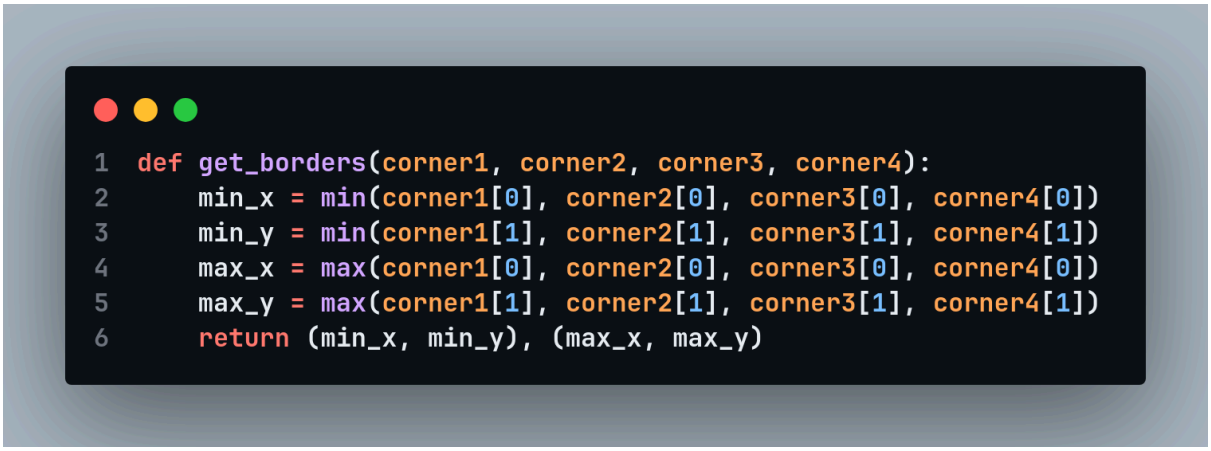
7. `get_borders(corner1, corner2, corner3, corner4)`

What it does:
Calculates the bounding box (min and max x and y) that encloses four given points.

How it works:

- Takes four points (usually image corners after transformation).

Finds the smallest and largest x and y values.



```python
def get_borders(corner1, corner2, corner3, corner4):
    min_x = min(corner1[0], corner2[0], corner3[0], corner4[0])
    min_y = min(corner1[1], corner2[1], corner3[1], corner4[1])
    max_x = max(corner1[0], corner2[0], corner3[0], corner4[0])
    max_y = max(corner1[1], corner2[1], corner3[1], corner4[1])
    return (min_x, min_y), (max_x, max_y)
```

8. `warp_image(img1, img2, h)`

What it does:
Warps `img1` using a homography and overlays it on top of `img2` to create a stitched mosaic.

How it works:

- Computes the projected corners of the warped image and the bounding box for the final stitched image.
- Initializes a canvas large enough to hold both images.
- Paste `img2` onto the canvas.
- Iterates through the region where `img1` will go, maps each canvas pixel back to the source (`img1`) using the inverse homography.
- For valid coordinates, it uses interpolation to get the color from `img1` and sets it in the canvas.

Why it's important:
This is the core of image stitching. It geometrically transforms one image and blends it with the other to form a panoramic view.

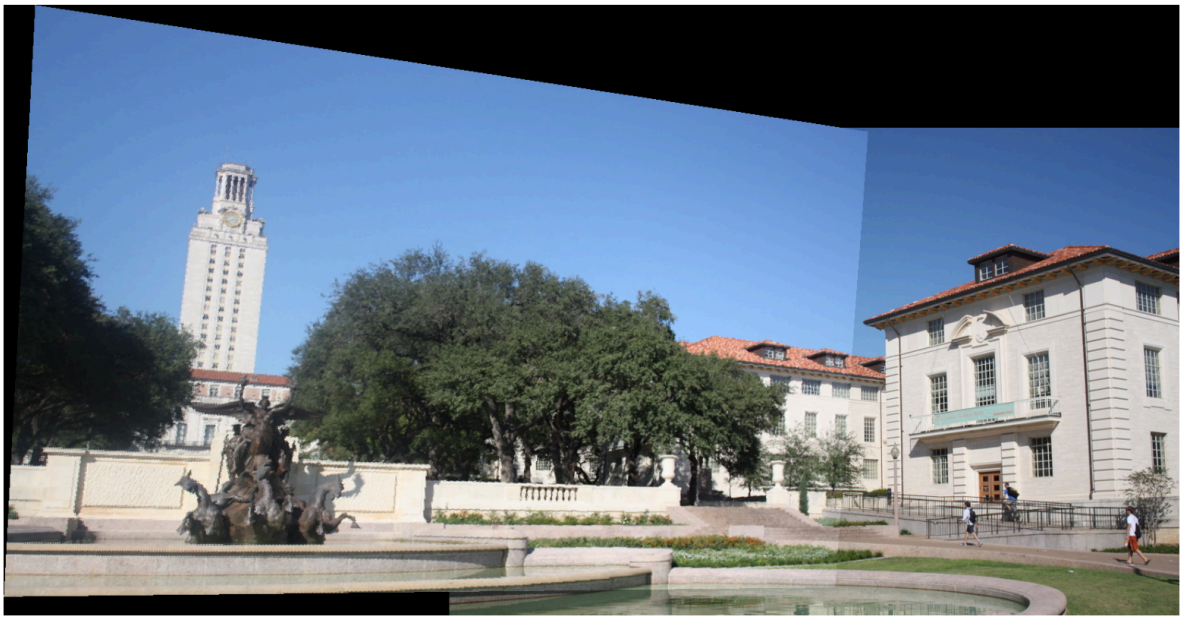
```python
def warp_image(warped_img, original_img, h):
    warped_y, warped_x, _ = warped_img.shape
    original_y, original_x, _ = original_img.shape

    warped_corner1, warped_corner2, warped_corner3, warped_corner4 = getCorners(
        warped_img, h
    )

    border_min_point, _ = get_borders(
        warped_corner1, warped_corner2, (0, 0), (0, original_y)
    )

    _, border_max_point = get_borders(
        warped_corner3,
        warped_corner4,
        (original_x, 0),
        (original_x, original_y),
    )

    height = border_max_point[1] - border_min_point[1]
    width = border_max_point[0] - border_min_point[0]
    final_img = np.zeros((height, width, 3))

    warped_min_border, warped_max_border = get_borders(
        warped_corner1, warped_corner2, warped_corner3, warped_corner4
    )

    corners = [
        (0, 0),
        (0, original_y),
        (original_x, original_y),
        (original_x, 0),
    ]

    if border_min_point[0] == 0 and border_min_point[1] == 0:
        final_img[0:original_img.shape[0], 0:original_img.shape[1]] = original_img
    else:
        shifted_corners = [
            (point[0] - warped_min_border[0], point[1] - warped_min_border[1])
            for point in corners
        ]
        final_img[
            shifted_corners[0][1]:shifted_corners[2][1],
            shifted_corners[0][0]:shifted_corners[2][0],
        ] = original_img

    h_inverse = np.linalg.inv(h)

    # Warp each channel separately
    for channel in range(3):
        for i in range(warped_min_border[0], warped_max_border[0]):
            for j in range(warped_min_border[1], warped_max_border[1]):
                p_dash = compute_p_dash(h_inverse, (i, j), True)
                if (
                    p_dash[0] < 0
                    or p_dash[0] > warped_x - 1
                    or p_dash[1] < 0
                    or p_dash[1] > warped_y - 1
                ):
                    continue
                new_x = abs(border_min_point[0] - i)
                new_y = abs(border_min_point[1] - j)

                final_img[new_y][new_x][channel] = interpolation(
                    p_dash[0], p_dash[1], warped_img[:, :, channel]
                )

    return final_img, warped_min_border
```

All Steps combined in a function

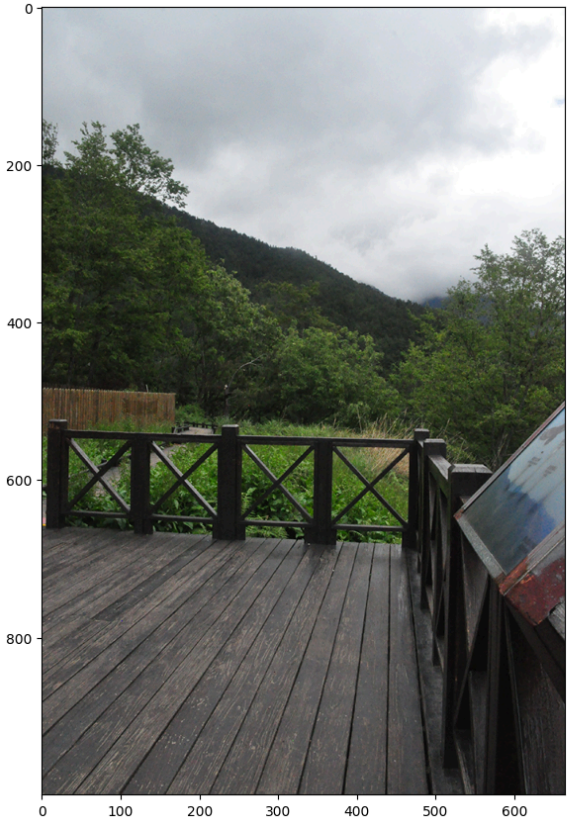
```python
def stitch_images(img1, img2):
    correspondences, _ = getMatches(img1, img2)
    h = RANSAC(correspondences)
    stitched_img, _ = warp_image(img1, img2, h)
    stitched_img = stitched_img.astype(np.uint8)
    return stitched_img
```

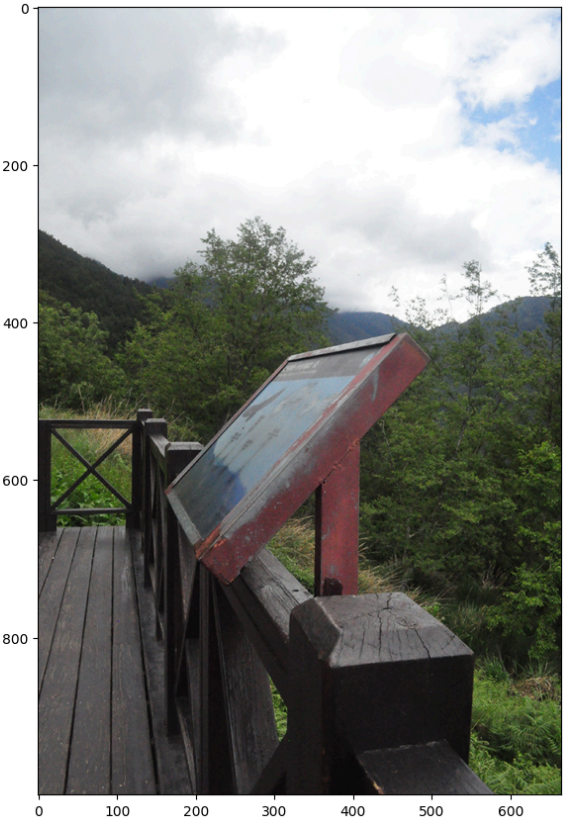
Right stitch (img1 is the warped image**)**

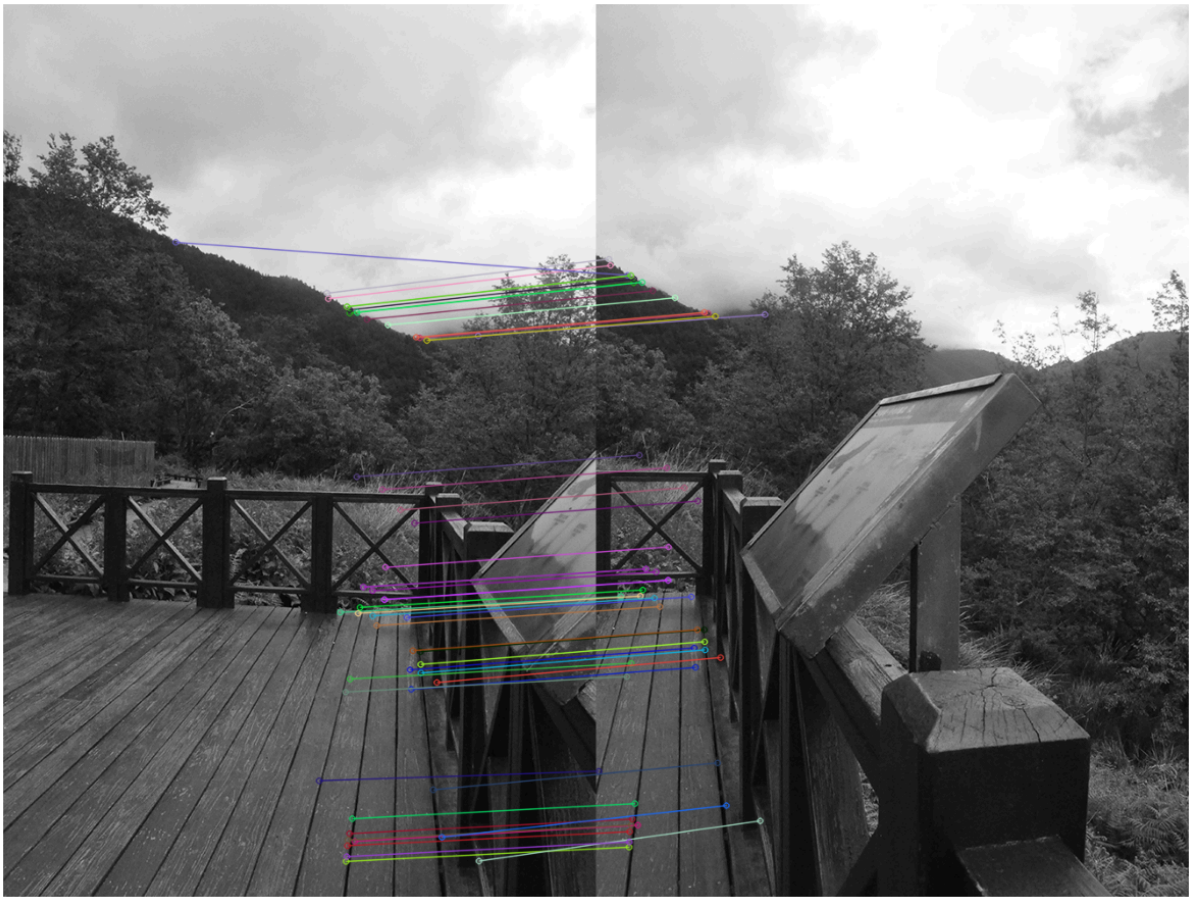
Left stitch (img2 is the warped image**)**



Another Example

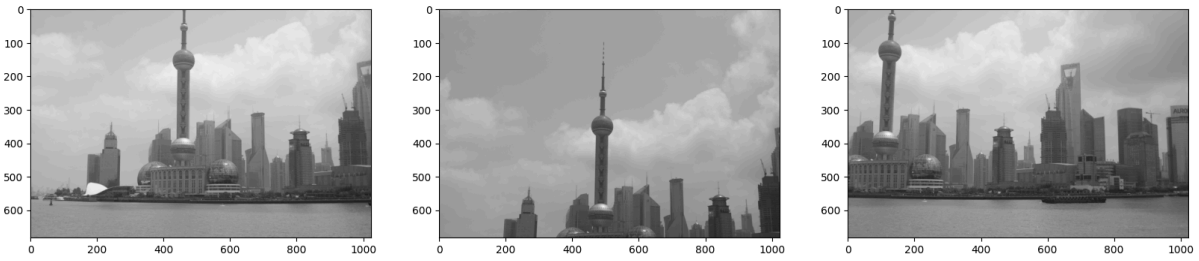






Stitching 3 Images (BONUS)

→ Stitch 2 images then stitch the output with the third image



Another Example