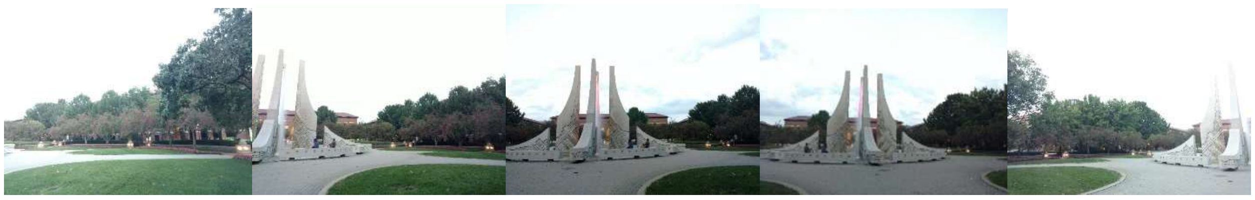
Homework 5

Theory question

1. To differentiate between inliers and outliers when using RANSAC for homography estimation, the algorithm randomly selects a subset of interest points from two images and estimates a homography matrix. Points that align well with the estimated homography, having a reprojection error below a predefined threshold, are considered inliers, while those that do not align (i.e., have larger reprojection errors) are classified as outliers.
2. The Levenberg-Marquardt (LM) algorithm combines the reliability of Gradient Descent (GD) and the speed of Gauss-Newton (GN) to offer a method that is both fast and numerically stable. It achieves this by acting like GD when the solution is far from optimal (to ensure stability) and gradually transitioning to GN as it gets closer to the minimum (for faster convergence). This balance makes LM particularly useful in minimizing cost functions efficiently.

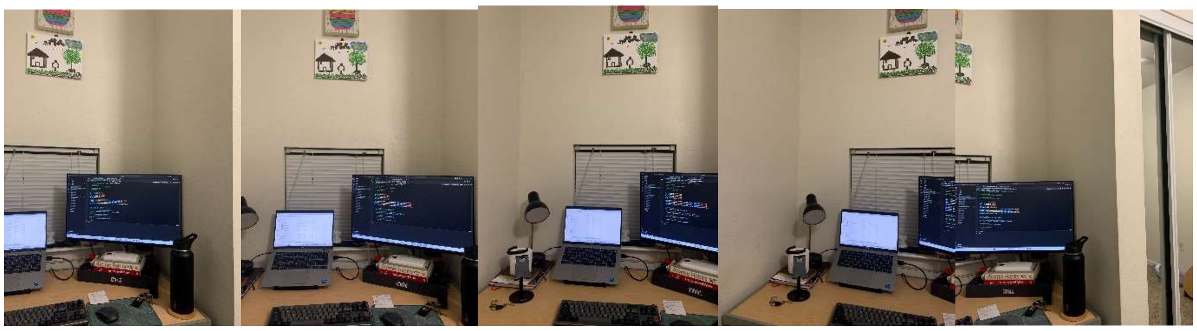
Input images

The input images used for this homework can be seen in Figure 1 and Figure 2



(a) (b) (c) (d) (e)

Figure 1. Input images provided with the homework

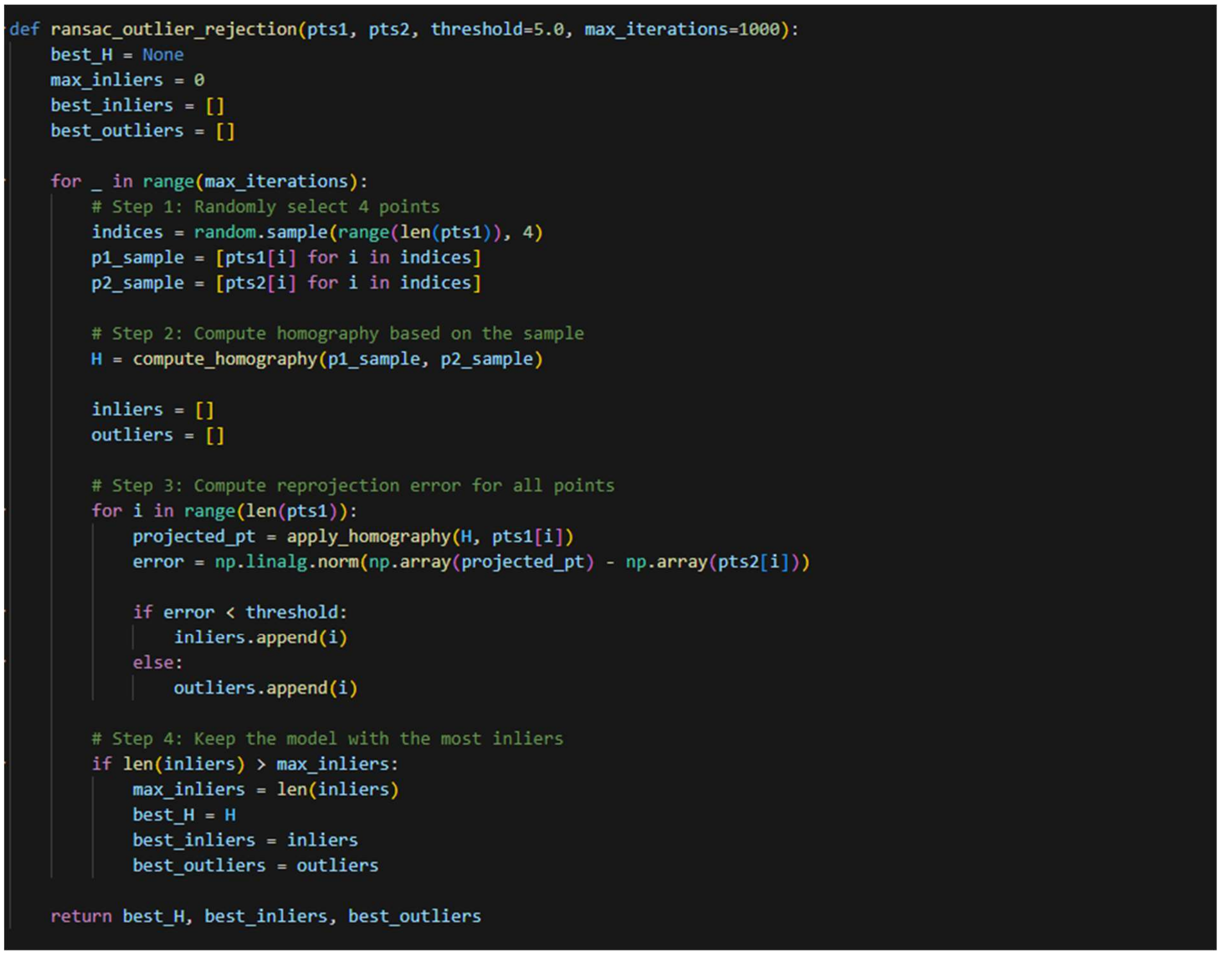


(a) (b) (c) (d) (e)

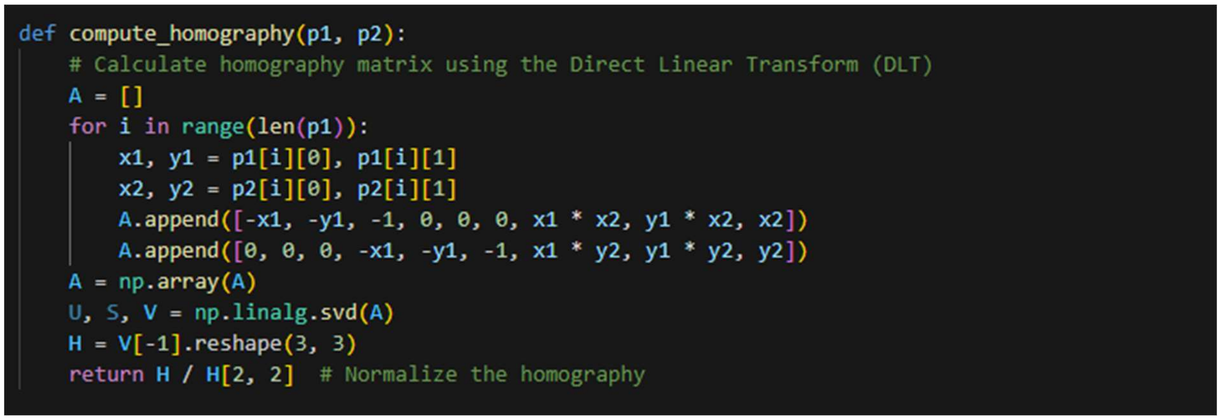
Figure 2. Input images personally collected

RANSAC implementation

The RANSAC implementation estimates a robust homography matrix between two sets of corresponding points (pts1 and pts2) while rejecting outliers. In each iteration, a random subset of four-point pairs is selected to compute a homography using the Direct Linear Transform (DLT). The homography is then applied to all points, and the reprojection error is calculated by comparing projected points to their corresponding points. Points with an error below the predefined threshold (5.0) are considered inliers, while others are outliers. The estimate with the maximum number of inliers is selected as the best homography, returning the inliers and outliers. The code snippet for implementation can be seem in the image below. Figure 3 - Figure 6 shows the output of inlier and outlier correspondence on the provided images after applying RANSAC rejection. The lines with blue edges represent the inliers while the lines with red edges represents the outliers. The best returned using RANSAC was refined using the inlier set



```python
def ransac_outlier_rejection(pts1, pts2, threshold=5.0, max_iterations=1000):
    best_H = None
    max_inliers = 0
    best_inliers = []
    best_outliers = []

    for _ in range(max_iterations):
        # Step 1: Randomly select 4 points
        indices = random.sample(range(len(pts1)), 4)
        p1_sample = [pts1[i] for i in indices]
        p2_sample = [pts2[i] for i in indices]

        # Step 2: Compute homography based on the sample
        H = compute_homography(p1_sample, p2_sample)

        inliers = []
        outliers = []

        # Step 3: Compute reprojection error for all points
        for i in range(len(pts1)):
            projected_pt = apply_homography(H, pts1[i])
            error = np.linalg.norm(np.array(projected_pt) - np.array(pts2[i]))

            if error < threshold:
                inliers.append(i)
            else:
                outliers.append(i)

        # Step 4: Keep the model with the most inliers
        if len(inliers) > max_inliers:
            max_inliers = len(inliers)
            best_H = H
            best_inliers = inliers
            best_outliers = outliers

    return best_H, best_inliers, best_outliers
```



```
def compute_homography(p1, p2):
    # Calculate homography matrix using the Direct Linear Transform (DLT)
    A = []
    for i in range(len(p1)):
        x1, y1 = p1[i][0], p1[i][1]
        x2, y2 = p2[i][0], p2[i][1]
        A.append([-x1, -y1, -1, 0, 0, 0, x1 * x2, y1 * x2, x2])
        A.append([0, 0, 0, -x1, -y1, -1, x1 * y2, y1 * y2, y2])
    A = np.array(A)
    U, S, V = np.linalg.svd(A)
    H = V[-1].reshape(3, 3)
    return H / H[2, 2]  # Normalize the homography
```

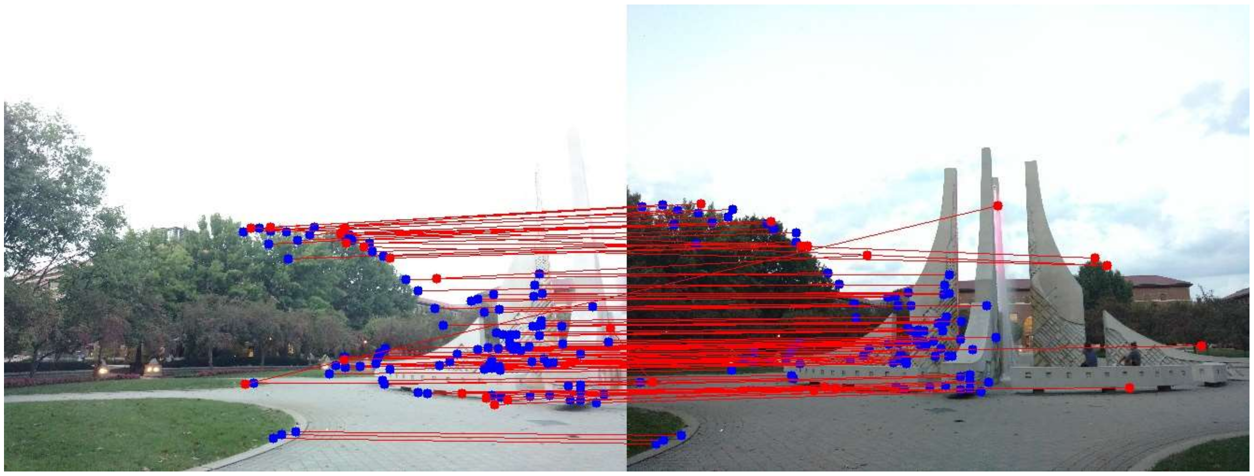


Figure 3. Inlier-outlier between images 1a and 1b

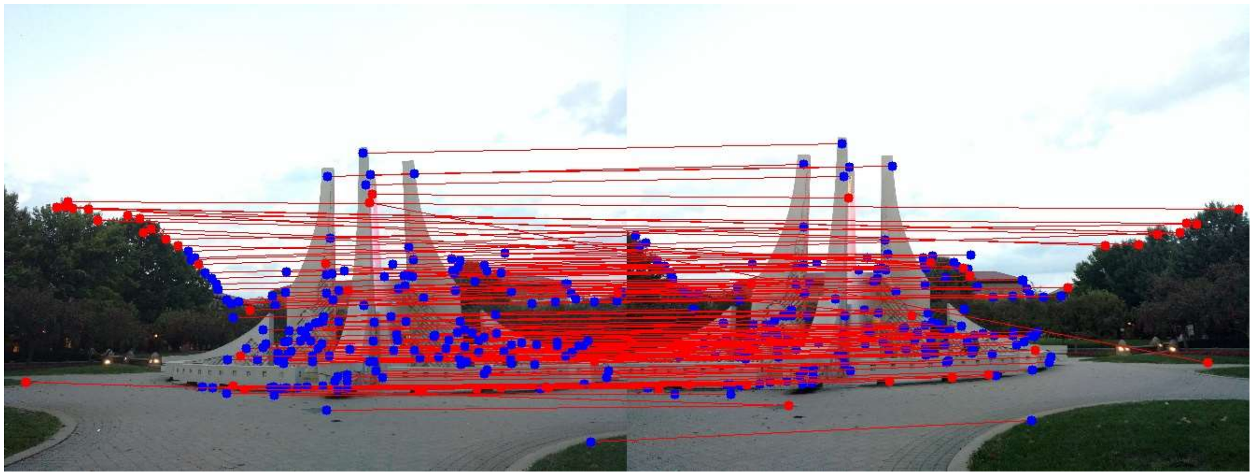


Figure 4. Inlier-outlier between images 1b and 1c

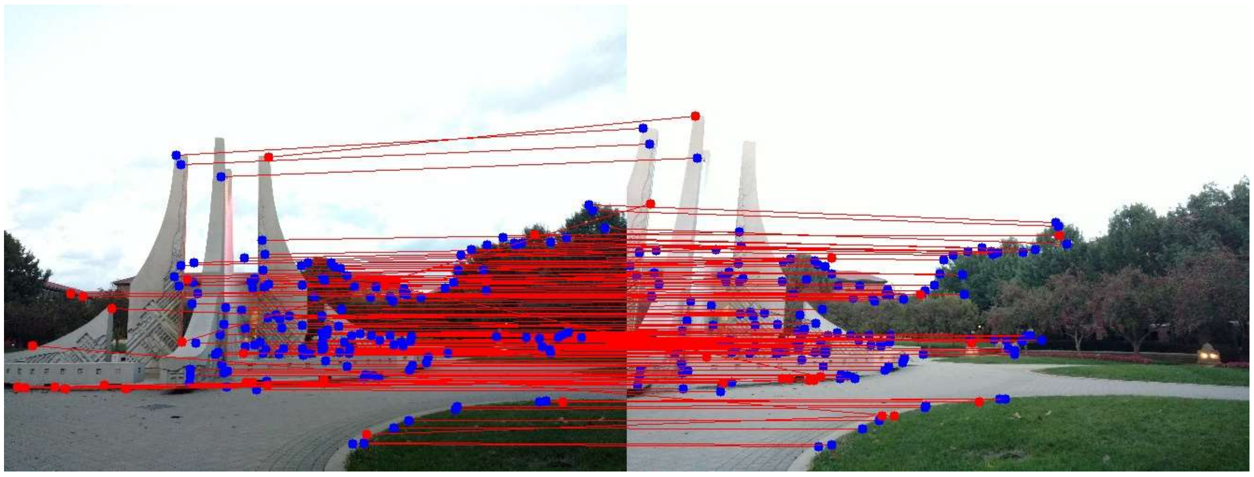


Figure 5. Inlier-outlier between images 1c and 1d

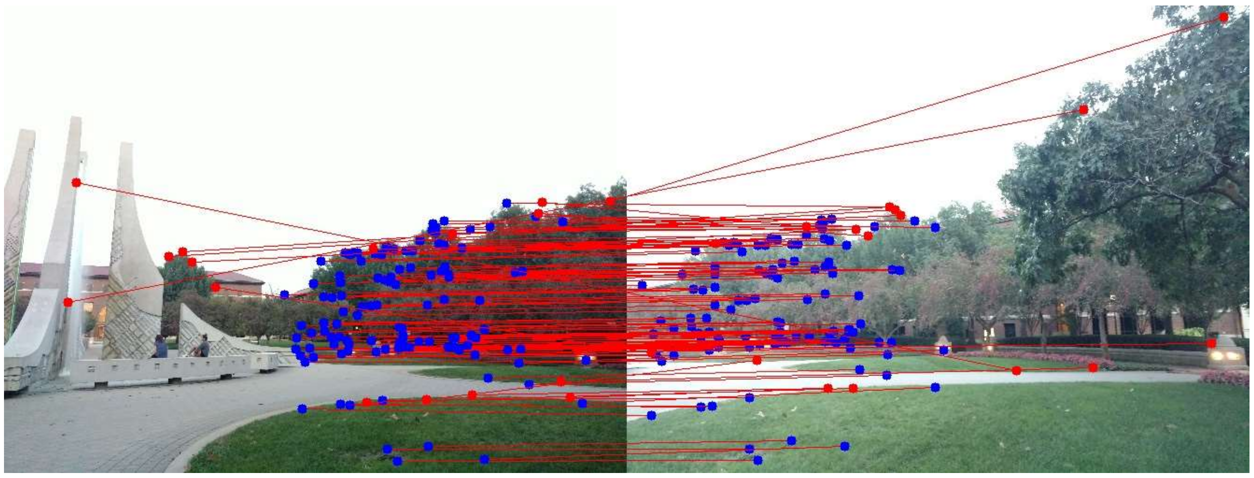
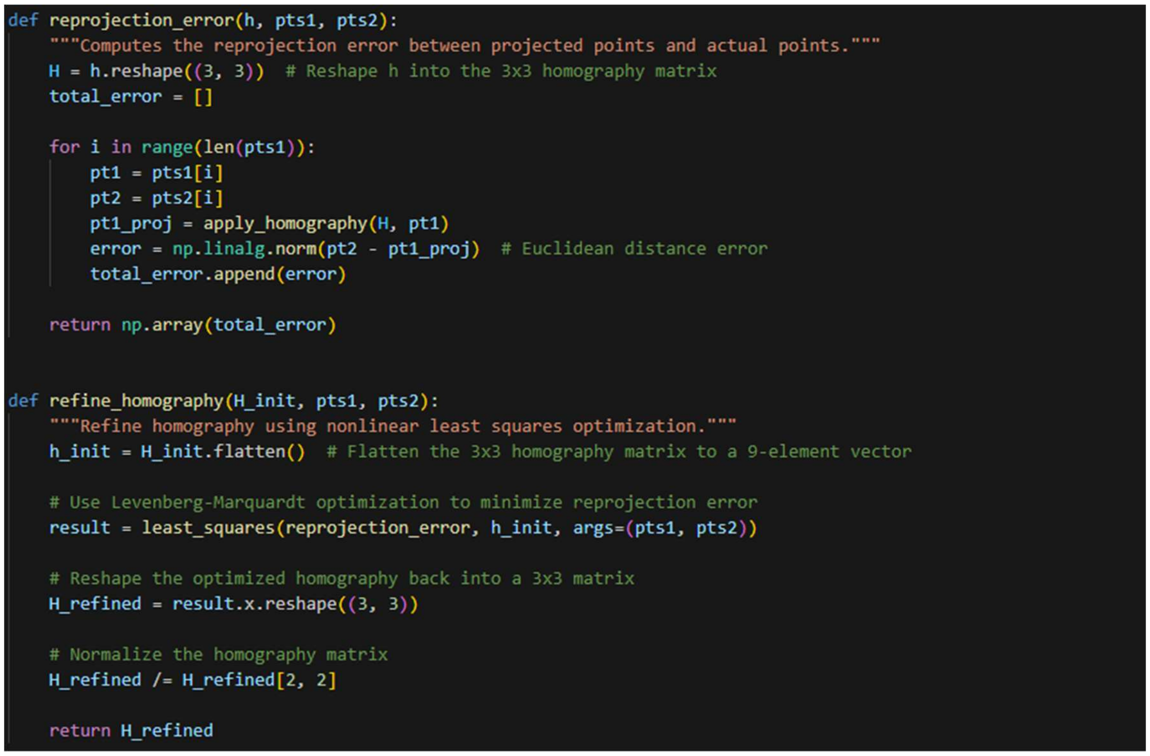


Figure 6. Inlier-outlier between images 1d and 1e

<u>Homography refinement</u>

The homography obtained from the RANSAC algorithm is further refined using nonlinear least squares optimization to minimize the reprojection errorbetween the projected points and actual points. This approach uses the Levenberg-Marquardt algorithm (via scipy library) to iteratively minimize the reprojection error. The optimized homography is reshaped into a 3x3 matrix and normalized before being returned as the refined result. The implementation can be found in the image below.



```python
def reprojection_error(h, pts1, pts2):
    """Computes the reprojection error between projected points and actual points."""
    H = h.reshape((3, 3))  # Reshape h into the 3x3 homography matrix
    total_error = []

    for i in range(len(pts1)):
        pt1 = pts1[i]
        pt2 = pts2[i]
        pt1_proj = apply_homography(H, pt1)
        error = np.linalg.norm(pt2 - pt1_proj)  # Euclidean distance error
        total_error.append(error)

    return np.array(total_error)


def refine_homography(H_init, pts1, pts2):
    """Refine homography using nonlinear least squares optimization."""
    h_init = H_init.flatten()  # Flatten the 3x3 homography matrix to a 9-element vector

    # Use Levenberg-Marquardt optimization to minimize reprojection error
    result = least_squares(reprojection_error, h_init, args=(pts1, pts2))

    # Reshape the optimized homography back into a 3x3 matrix
    H_refined = result.x.reshape((3, 3))

    # Normalize the homography matrix
    H_refined /= H_refined[2, 2]

    return H_refined
```

Mosaic

The implementation to creat mosaic using the refined homographies between consecutive pairs can be seen below. The middle image is used as the reference frame, and homographies for images to the left and right are computed relative to this reference. After determining the homographies, the output mosaic size is calculated based on image dimensions and homographies, including necessary translations. Each image is then warped into the mosaic frame using the calculated homography, and blended into the mosaic by replacing empty pixels with the warped image's pixels, thus forming a continuous panorama. The resulting mosaic can be seen in Figure 7

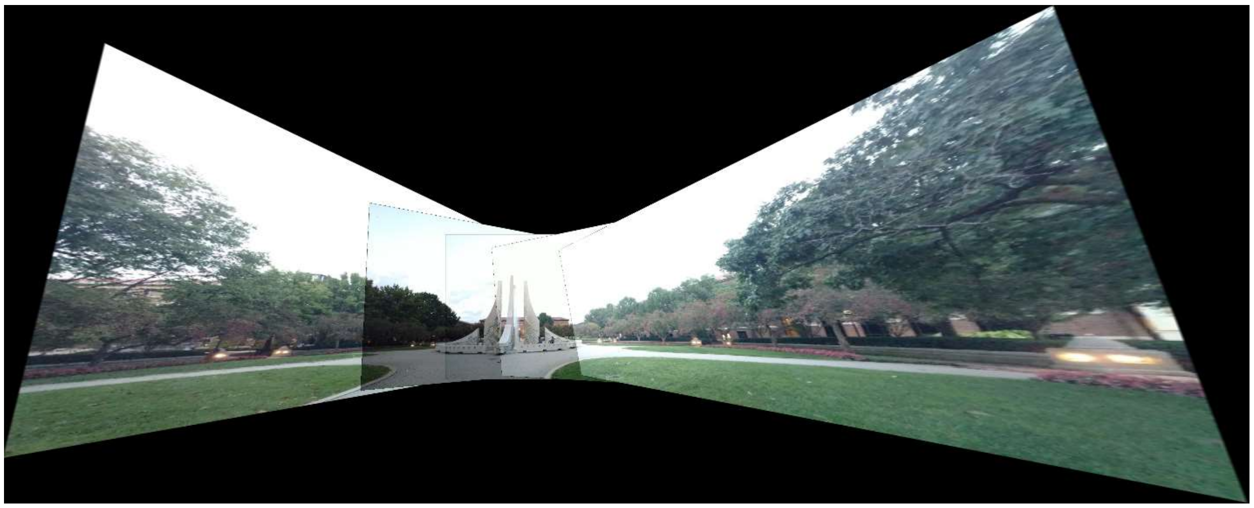


Figure 7. Mosaic of provided image

Figure 8-12 shows the results of inlier-outlier after using RANSAC and mosaic on the personally collected images.

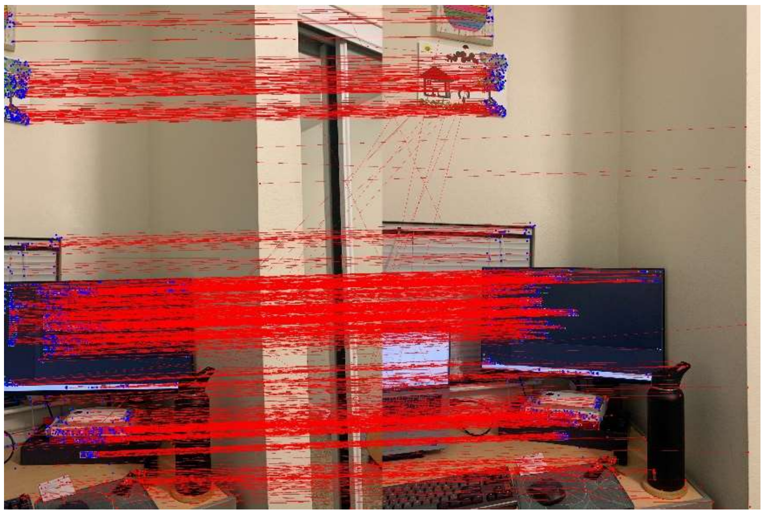


Figure 8. Inlier-outlier between Figures 2a and 2b

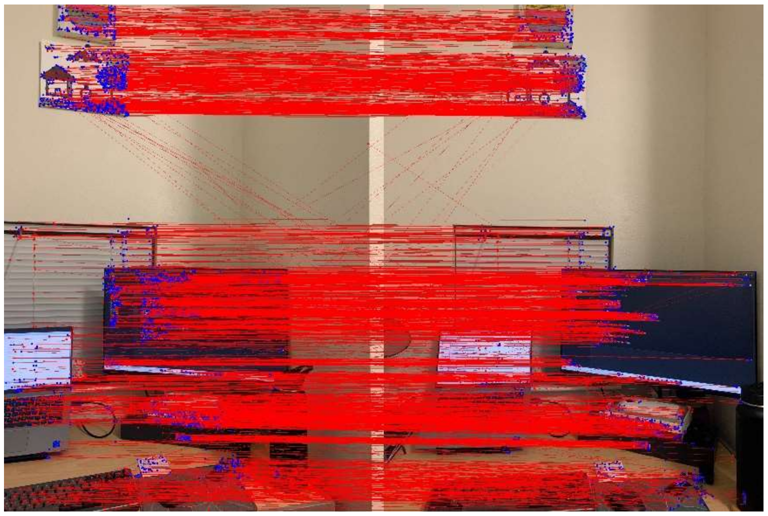


Figure 9. Inlier-outlier between Figures 2b and 2c

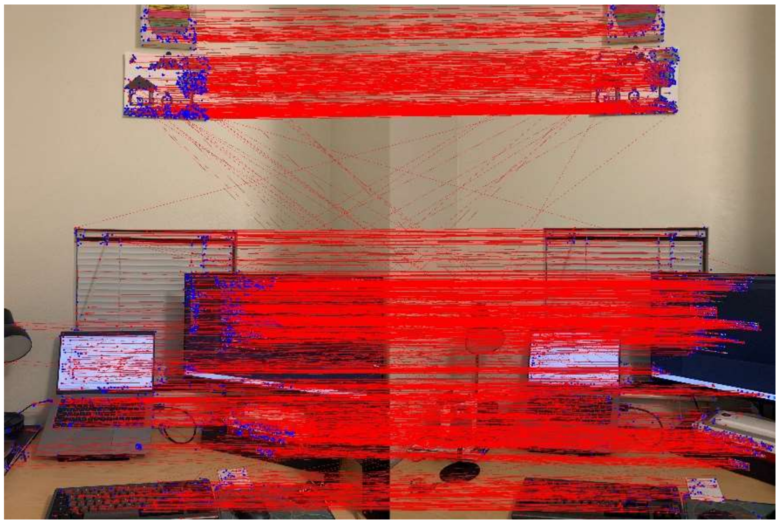


Figure 10. Inlier-outlier between Figures 2c and 2d

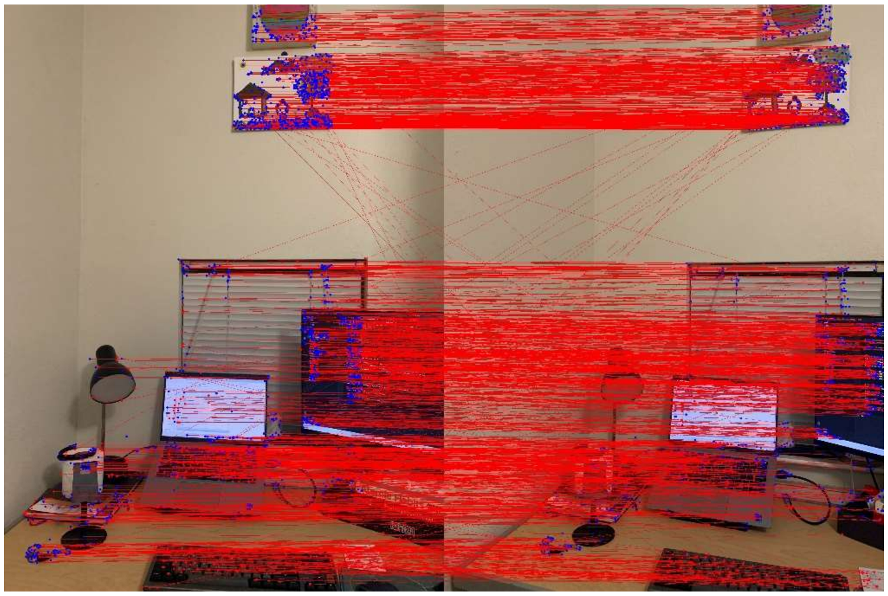


Figure 11. Inlier-outlier between Figures 2d and 2e

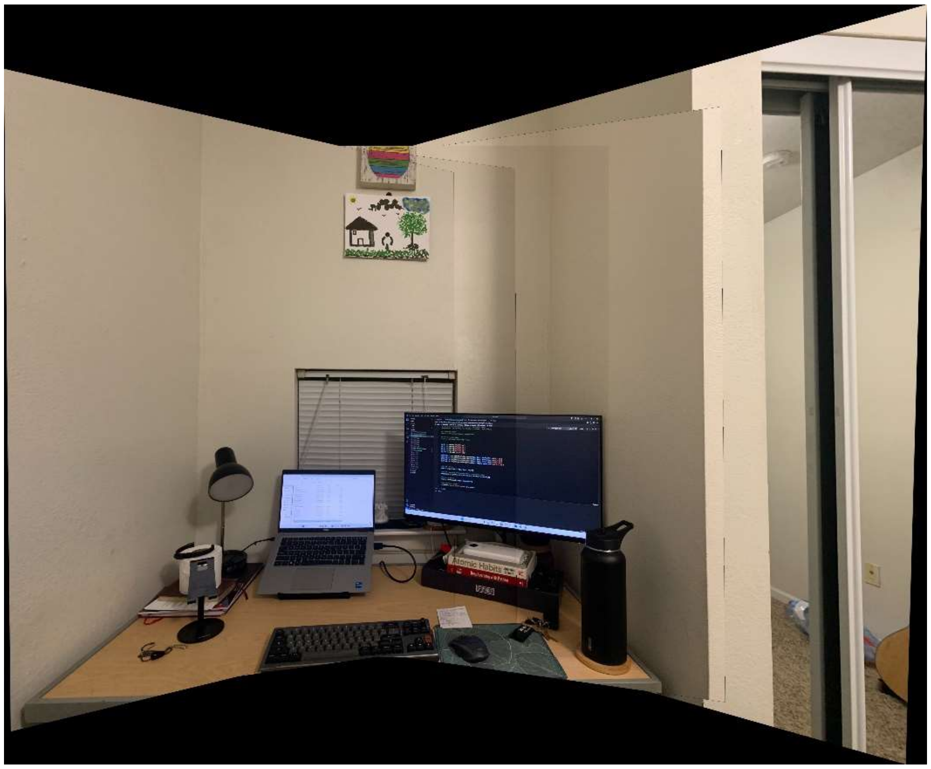
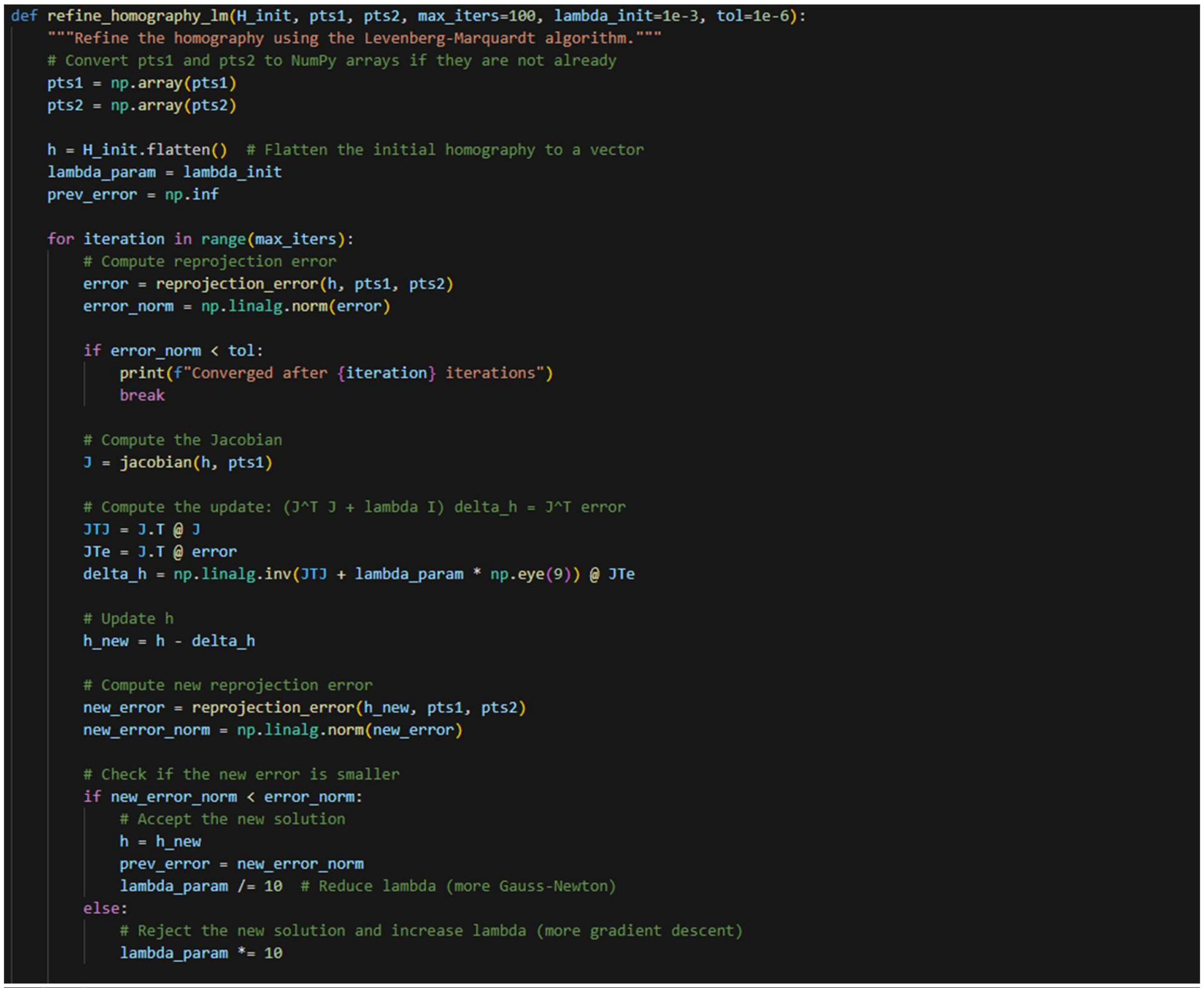


Figure 12. Mosaic of personally collected images

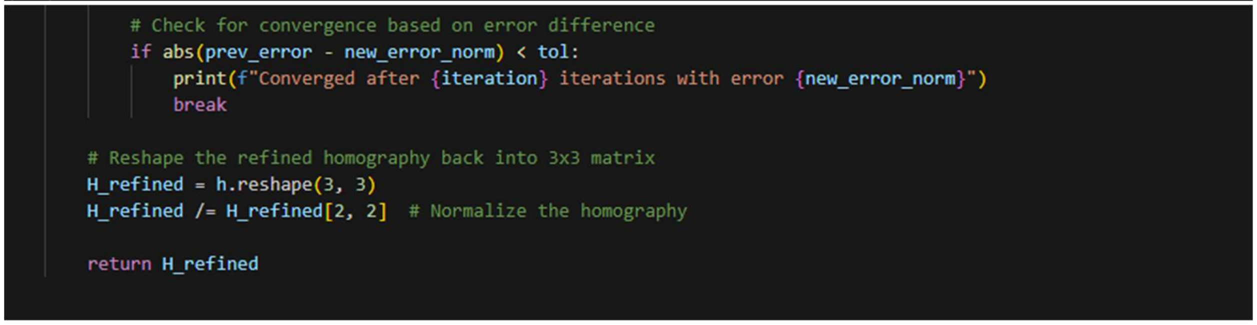
EXTRA CREDIT

Levenberg-Marquardt (LM) algorithm

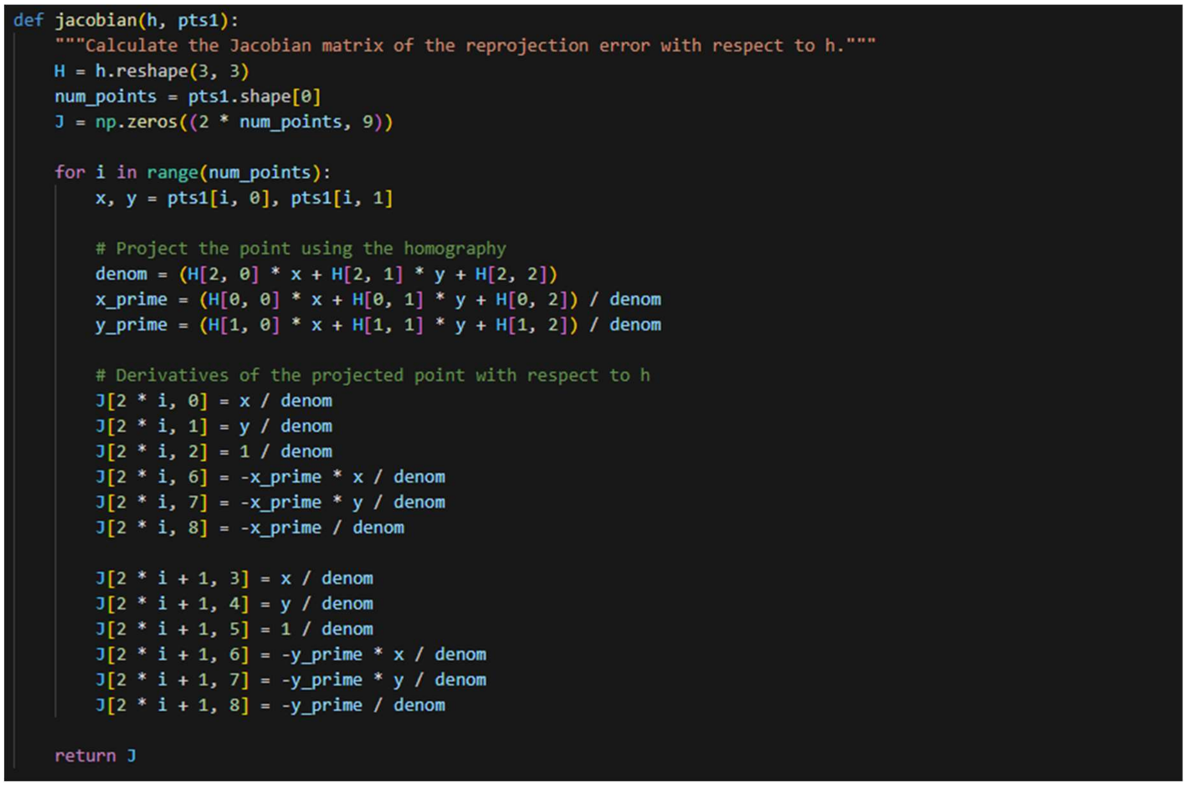
In the implementation below, a reprojection error function is defined to compute the difference between actual points and projected points using the homography matrix obtained from RANSAC. The algorithm combines gradient descent and the Gauss-Newton method. The LM algorithm iteratively refines the homography by minimizing the reprojection error between points in two images. A Jacobian matrix of the reprojection error is calculated for efficient error minimization, and the algorithm adjusts a damping factor (lambda) to balance between gradient descent and the Gauss-Newton method, ensuring convergence. The refinement process terminates when the error reduces below a threshold or the change in error becomes negligible, resulting in a refined homography matrix.



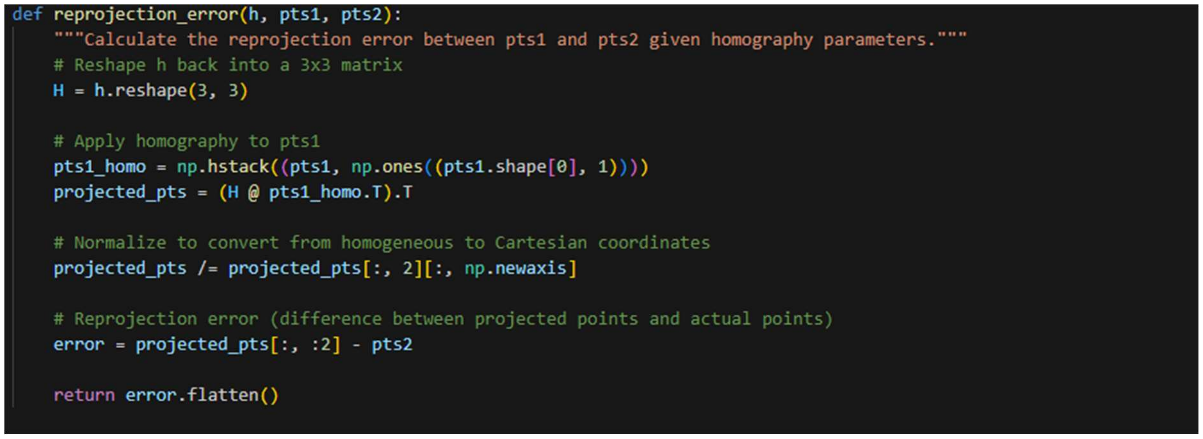
```python
def refine_homography_lm(H_init, pts1, pts2, max_iters=100, lambda_init=1e-3, tol=1e-6):
    """Refine the homography using the Levenberg-Marquardt algorithm."""
    # Convert pts1 and pts2 to NumPy arrays if they are not already
    pts1 = np.array(pts1)
    pts2 = np.array(pts2)

    h = H_init.flatten()  # Flatten the initial homography to a vector
    lambda_param = lambda_init
    prev_error = np.inf

    for iteration in range(max_iters):
        # Compute reprojection error
        error = reprojection_error(h, pts1, pts2)
        error_norm = np.linalg.norm(error)

        if error_norm < tol:
            print(f"Converged after {iteration} iterations")
            break

        # Compute the Jacobian
        J = jacobian(h, pts1)

        # Compute the update: (J^T J + lambda I) delta_h = J^T error
        JTJ = J.T @ J
        JTe = J.T @ error
        delta_h = np.linalg.inv(JTJ + lambda_param * np.eye(9)) @ JTe

        # Update h
        h_new = h - delta_h

        # Compute new reprojection error
        new_error = reprojection_error(h_new, pts1, pts2)
        new_error_norm = np.linalg.norm(new_error)

        # Check if the new error is smaller
        if new_error_norm < error_norm:
            # Accept the new solution
            h = h_new
            prev_error = new_error_norm
            lambda_param /= 10  # Reduce lambda (more Gauss-Newton)
        else:
            # Reject the new solution and increase lambda (more gradient descent)
            lambda_param *= 10
```



```python
        # Check for convergence based on error difference
        if abs(prev_error - new_error_norm) < tol:
            print(f"Converged after {iteration} iterations with error {new_error_norm}")
            break

    # Reshape the refined homography back into 3x3 matrix
    H_refined = h.reshape(3, 3)
    H_refined /= H_refined[2, 2]  # Normalize the homography

    return H_refined
```



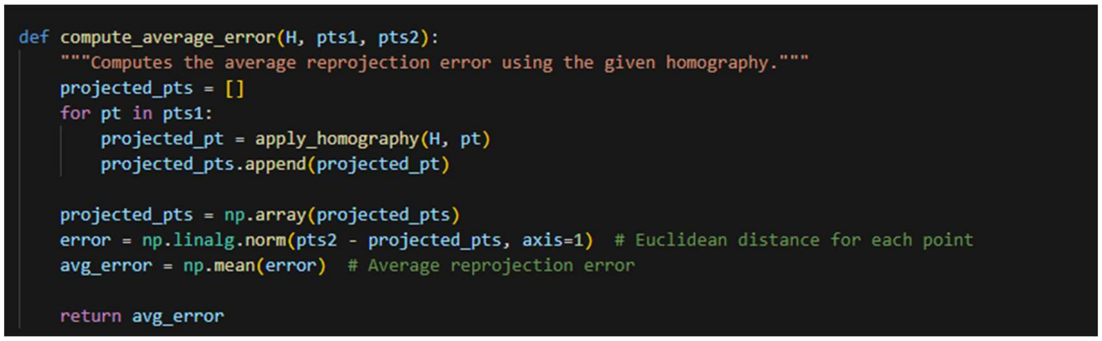
```python
def jacobian(h, pts1):
    """Calculate the Jacobian matrix of the reprojection error with respect to h."""
    H = h.reshape(3, 3)
    num_points = pts1.shape[0]
    J = np.zeros((2 * num_points, 9))

    for i in range(num_points):
        x, y = pts1[i, 0], pts1[i, 1]

        # Project the point using the homography
        denom = (H[2, 0] * x + H[2, 1] * y + H[2, 2])
        x_prime = (H[0, 0] * x + H[0, 1] * y + H[0, 2]) / denom
        y_prime = (H[1, 0] * x + H[1, 1] * y + H[1, 2]) / denom

        # Derivatives of the projected point with respect to h
        J[2 * i, 0] = x / denom
        J[2 * i, 1] = y / denom
        J[2 * i, 2] = 1 / denom
        J[2 * i, 6] = -x_prime * x / denom
        J[2 * i, 7] = -x_prime * y / denom
        J[2 * i, 8] = -x_prime / denom

        J[2 * i + 1, 3] = x / denom
        J[2 * i + 1, 4] = y / denom
        J[2 * i + 1, 5] = 1 / denom
        J[2 * i + 1, 6] = -y_prime * x / denom
        J[2 * i + 1, 7] = -y_prime * y / denom
        J[2 * i + 1, 8] = -y_prime / denom

    return J
```



```python
def reprojection_error(h, pts1, pts2):
    """Calculate the reprojection error between pts1 and pts2 given homography parameters."""
    # Reshape h back into a 3x3 matrix
    H = h.reshape(3, 3)

    # Apply homography to pts1
    pts1_homo = np.hstack((pts1, np.ones((pts1.shape[0], 1))))
    projected_pts = (H @ pts1_homo.T).T

    # Normalize to convert from homogeneous to Cartesian coordinates
    projected_pts /= projected_pts[:, 2][:, np.newaxis]

    # Reprojection error (difference between projected points and actual points)
    error = projected_pts[:, :2] - pts2

    return error.flatten()
```

Effectiveness of LM algorithm

The code snippet to compute the LM algorithm can be found in the image below



```python
def compute_average_error(H, pts1, pts2):
    """Computes the average reprojection error using the given homography."""
    projected_pts = []
    for pt in pts1:
        projected_pt = apply_homography(H, pt)
        projected_pts.append(projected_pt)

    projected_pts = np.array(projected_pts)
    error = np.linalg.norm(pts2 - projected_pts, axis=1)  # Euclidean distance for each point
    avg_error = np.mean(error)  # Average reprojection error

    return avg_error
```

Geometric error reduction

Image pairs	Reduction in geometric error (%)
Provided image Pair2_3	2.74
Provided image Pair3_4	0.29
Provided image Pair4_5	0.12
Collected image Pair7_8	0.04
Collected image Pair8_9	1.01
Collected image Pair9_10	0.48