ECE 66100 Homework #5

by

Adrien Dubois (dubois6@purdue.edu)

October 23, 2025

Contents

1 Theory Questions

1.1 Question 1:

Conceptually speaking, how do we differentiate between the inliers and the outliers when using RANSAC for solving the homography estimation problem using the interest points extracted from two different photos of the same scene?

When using the RANSAC algorithm, we can differentiate between the inliers and the outliers through the use of the δ parameter. This controls the threshold for the pixel-pixel distance tolerated between the

points in the target domain, and the points from the source domain warped by the estimated homography. Therefore, for n points the criteria for an inlier vs outlier assignment is:

$$\begin{cases} \sqrt{\sum_i^n (x' - Hx)^2} < \delta) & \rightarrow \textbf{inlier} \\ \text{else} & \rightarrow \textbf{outlier} \end{cases}$$

1.2 Question 2:

As you will see in Lecture 13, the Gradient-Descent (GD) is a reliable method for minimizing a cost function, but it can be excruciatingly slow. At the other extreme, we have the much faster Gauss-Newton (GN) method but it can be numerically unstable. Explain in your own words how the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to give us a method that is reasonably fast and numerically stable at the same time.

The gradient descent algorithm can be very slow since, as you get closer to the minimum point along the surface you are optimizing, the gradients go to 0. Therefore, the closer you are to the minimum, the slower the algorithm will approach it. However, the Gradient Descent algorithm is very stable and will always go to a minimum if it has ran for enough time. On the other hand, the Gauss-Newton algorithm goes to the minimum point solution in one step, but it is very unstable. Therefore, the Levenberg-Marquardt algorithm combines the best of both algorithms by introducing a dampening factor μ. We then are able to heuristically go between both the GD and GN algorithms:

$$\left(\boldsymbol{J}_f^T \boldsymbol{J}_f + \mu \boldsymbol{I}\right) \delta_p = \boldsymbol{J}_f^T \epsilon(\vec{p_k})$$

So, if μ is much larger than the diagonal elements of $\boldsymbol{J}_f^T \boldsymbol{J}_f$ the solution will be close to gradient descent. On the other hand, if μ is much smaller than the diagonal elements, then the solution will be closer to Gauss-Newton. Lastly, we determine the value for μ by pre-computing the solution to GN and determining a quality factor for that solution. This is determined through the sign of $C(p_k) - C(p_{k+1})$ where C is the cost function:

$$C(p) = ||X - f(p)||^2$$

2 RANSAC Algorithm

2.1 Algorithm Description

In previous projects, we have used keypoint detection tools such as the Harris Corner Detector, and keypoint matching tools such as NCC (normalized cross correlation). However, some key-point matches created through the use of these tools still have significant error. If we used all of these noisy correspondences to estimate the homographies we would get a very wrong answer. Therefore, when performing automatic homography estimate we need to use an error correction tool like RANSAC to remove the false-correspondences while keeping correct correspondences. It is also important to note that all correspondences will have some level of noise in their re-projection so algorithms like RANSAC have been implemented to differentiate between noise-correspondences and false-correspondences.

With this information, my RANSAC implementation works through the following steps:

1. We first define the following variables:
   (a) ϵ , the probability that a random point is an outlier
   (b) p, the probability that at least on trial has no outliers
   (c) n, the number of samples randomly selected during each trial
   (d) δ, the threshold distance for inlier assignment
2. Calculate N, the number of trials required for there to be a p % change of at least one trial having no outliers:
$$\frac{\ln(1-p)}{1-(1-\epsilon)^n}$$
3. For each trial, perform the following steps:
   (a) Randomly sample n correspondences

(b) Compute the homography from those correspondences using linear least-squares.

(c) Map each point in the source domain to the target domain using the estimated homography

(d) Compute the euclidean distance between the mapped points and the ground truth target points. Then assign them to the outliers or the inliers set using the following rule:

$$\begin{cases} \sqrt{\sum_i^n (x' - Hx)^2} < \delta) & \rightarrow \textbf{inlier} \\ \text{else} & \rightarrow \textbf{outlier} \end{cases}$$

4. Lastly, we choose the trial that had the largest set of inlier correspondences within all trials.

2.2 RANSAC Parameters

For my implementation of RANSAC, I used the following parameter values:

- n = 4
- p = 0.99
- $\epsilon = 0.3$
- N = 16
- $\delta = 3 * \sigma = 3*1.2$

```
def RANSAC_homography_estimation(sigma, match_list):
    num_samples = 4 # Number of points to randomly select during each trial
    p = 0.99 # probability that at least one trial has no outliers
    epsilon = 0.3 # probability that a random point is an outlier
    N = int(np.log(1-p)/np.log(1-(1-epsilon)**(num_samples))) # The number of trials
    required,
    delta = 3*sigma

    best_points = {"Inlier_Percent": -np.inf, "Points":[]} # Error Score, [((x11,y11), (
    x21,y21)), ((x12, y12), (x22, y22)), ...]

    # Go through N trials
    for trial_idx in range(N):
        # Sample 4 random matches from the list
        sampled_points = random.sample(match_list, k=num_samples)

        # Convert the points into HC format
        source_points = [Point(coordinate[0][0], coordinate[0][1]) for coordinate in
    sampled_points]
        target_points = [Point(coordinate[1][0], coordinate[1][1]) for coordinate in
    sampled_points]

        # Compute homography using the sampled points
        H = Homography().estimate_projective_homography(source_points, target_points)

        # Compute the matches to check error. Matches are computed for all interest
    points
        # We calculate the distance from the target_known_point to the estimated x_prime
     point for all interest points
        # We will keep the point with the highest percentage of inlier points
        inlier_count = 0
        inliers = []

        for source, target in match_list:
            # Get x and x'_true points in HC
            source = Point(source[0], source[1])
            target = Point(target[0], target[1])

            # Estimate x' using estimated homography
            x_prime = Point.from_hc(H@source.hc)

            # Calculate the estimation error using Euclidean distance
            error = np.linalg.norm(x_prime.hc - target.hc, ord=2)

            # Check if the estimate is an inlier
```

```
            if error < delta:
                inlier_count += 1
                inliers.append((source, target))

        # Add the results to the dict
        if inlier_count/len(match_list) > best_points["Inlier_Percent"]:
            best_points["Inlier_Percent"] = inlier_count/len(match_list)
            best_points["Points"] = inliers

    return best_points["Inlier_Percent"], best_points["Points"]
```

3 Linear Least Squares

It is known that the relationship between a point x in the physical space, and its corresponding pixel x' in the image plane can be written as $x' = Hx$ by representing the two points x and x' using homogeneous coordinates.

Expanding this out, we get:

$$\vec{\boldsymbol{x'}} = \boldsymbol{H}\vec{\boldsymbol{x}}$$

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{31} \\ h_{21} & h_{22} & h_{32} \\ h_{31} & h_{23} & h_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Which is the same as:

$$x'_1 = h_{11}x_1 + h_{12}x_2 + h_{13}x_3$$

$$x'_2 = h_{21}x_1 + h_{22}x_2 + h_{23}x_3$$

$$x'_3 = h_{31}x_1 + h_{32}x_2 + h_{33}x_3$$

However, since we are measure (x,y) pixel coordinates, we have to re-write this into the physical space context where $x = \frac{x_1}{x_3}$ and $y = \frac{x_2}{x_3}$

So the physical coordinates of the image pixel in terms of the parameters of the general projective homography are:

$$x' = \frac{h_{11}x_1 + h_{12}x_2 + h_{13}x_3}{h_{31}x_1 + h_{32}x_2 + h_{33}x_3} \text{and} y' = \frac{h_{21}x_1 + h_{22}x_2 + h_{23}x_3}{h_{31}x_1 + h_{32}x_2 + h_{33}x_3}$$

Since we are setting $x_3 = 1$ when converting our points from the physical space in $\mathbb{R}^2$ to the homogeneous representational space in $\mathbb{R}^3$, we can divide the RHS of the equation by x_3 and rewrite the above equation just in terms of the physical points:

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \text{ and } y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}$$

Therefore, each measurement in the physical $\mathbb{R}^2$ space, gives us two equations. Since, h_33 is set equal to 1, we need 4 point correspondences to evaluate the 8 remaining unknowns of the H matrix. The full expansion is as follows:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x'_2x_2 & -x'_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y'_2x_2 & -y'_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x'_3x_3 & -x'_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y'_3x_3 & -y'_3y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x'_4x_4 & -x'_4y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y'_4x_4 & -y'_4y_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

This can be represented in the form $AH = b$. We can then solve for the homography through the pseudo-inverse:

$$H = (A^T A)^{-}1A^T b$$

```python
class Homography():
    def __init__(self):
        pass
    def get_first_six_cols(self, x_points):
        zero_vec = np.zeros(3)
        eqn1_first6cols = np.vstack([np.hstack((point.hc, zero_vec)) for point in x_points])
        eqn2_first6cols = np.vstack([np.hstack((zero_vec, point.hc)) for point in x_points])
        return np.vstack([[eqn1, eqn2] for eqn1, eqn2 in zip(eqn1_first6cols, eqn2_first6cols)]) # Stack the rows in an interleaved fashion

    def estimate_projective_homography(self, x_points, x_prime_points):
        # Lets build the first 6 columns of the matrix we are interested in:
        first6cols = self.get_first_six_cols(x_points)

        # Now we only need the last 2 columns
        # x' is in the shape: x1' y1' 1, x2' y2' 1 etc
        # x is: x1 y1 1, x2 y2 1, ...
        # So we take the outer product of the first two elements in the homogeneous form to get the remaining portion:
        last2cols_list = []
        for x, x_prime in zip(x_prime_points, x_points):
            last2cols_list.append(-np.outer(x.get_hc()[:2], x_prime.get_hc()[:2]))
        last2cols = np.vstack(last2cols_list)
        full_matrix = np.hstack((first6cols, last2cols))

        x_prime_matrix = np.vstack([np.vstack((x_prime.hc[0], x_prime.hc[1])) for x_prime in x_prime_points])
        H = np.linalg.pinv(full_matrix) @ x_prime_matrix
        H = np.vstack((H, np.array(1))).reshape((3,3))
        return H
```

4 Levenberg-Marquardt

4.1 General Explanation:

The Levenberg-Marquadt algorithm works by minimizing over a non-linear cost function:

$$C(p) = ||X - f(p)||^2$$

We can do this by combining both Gauss-Netwon and Gradient Descent methods using a damping coefficient. The equation to do this is included below, and its explanation is included in section 1.2. We first start from the Gauss-Newton solution:

$$\vec{\delta}_p = \left(\boldsymbol{J}_{\vec{f}}^T \boldsymbol{J}_{\vec{f}}\right)^{-1} \boldsymbol{J}_{\vec{f}}^T \epsilon(\vec{p})$$

We can rewrite this pseudo-inverse solution as the original Ax = b solution like so:

$$\left(\boldsymbol{J}_{\vec{f}}^T \boldsymbol{J}_{\vec{f}}\right) \vec{\delta}_p = \boldsymbol{J}_{\vec{f}}^T \epsilon(\vec{p})$$

While the previous solution is still for Gauss Newton, it can be easily recognized that if the $\left(\boldsymbol{J}_{\vec{f}}^T \boldsymbol{J}_{\vec{f}}\right)$ matrix were purely diagonal, we would get the Gradient-Descent solution instead. So, these two solutions can be defined in one solution that utilizes a damping coefficient μ to switch from GN to GD. The equation for the LM solution is therefore:

$$\left(\boldsymbol{J}_{\vec{f}}^T \boldsymbol{J}_{\vec{f}} + \mu \boldsymbol{I}\right) \vec{\delta}_p = \boldsymbol{J}_{\vec{f}}^T \epsilon(\vec{p})$$

So, we start with an initial guess from a linear least-squares solution. Then, at each iteration of LM we set a value for the damping coefficient to determine if the following step will be a GD step or if we are close enough to the minimum point to use GN. WE then set $\vec{\delta}_p$ using the previous equation and the solution for the next step would be:

$$\vec{\boldsymbol{p}}_{k+1} = \vec{\boldsymbol{p}}_k + \vec{\boldsymbol{\delta}}_p$$

The last remaining challenge is to figure out what value the damping coefficient needs to be set to. From section 1.2, we know that a larger μ will bring the solution closer to GD. Therefore, we first evaluate solution if we took a GN step, and determine its quality using the following equations:

$$\rho_{k+1} = \frac{C(\vec{p}_k) - C(\vec{p}_{k+1}}{\vec{\boldsymbol{\delta}}_p^T \boldsymbol{J}_{\vec{f}}^T \vec{\epsilon}(\vec{p}_k) + \vec{\boldsymbol{\delta}}_p^T \mu_k I \vec{\boldsymbol{\delta}}_{\boldsymbol{p}}}$$

For the next iteration, we then use the value of ρ_{k+1} to calculate μ_{k+1}

$$\mu_{k+1} = \mu_k * max\left\{\frac{1}{3}, 1 - (2\rho_{k+1} - 1)^3\right\}$$

4.2 BONUS: My Implementation of LM

My implementation closely follows the example included in the instructions at the following link:
https://users.ics.forth.gr/ lourakis/levmar/levmar.pdf
My re-implementation of the pseudo-code included in that article is included below.

```
def levenberg_marquadt(cost_func, opt_H, inliers):
    # Cost function for printing results:
    cost_values = []

    # Constants:
    k = 0
    v = 2
    tau = 1e-3
    epsilon_1, epsilon_2, epsilon_3 = 1e-5, 1e-5, 1e-5
    kmax = 100

    p = opt_H.flatten()
    J = calculate_jacobian(cost_func, p, inliers)
    A = J.T@J
    epsilon = cost_func(p, inliers)
    cost_values.append(epsilon)
    g = J.T@epsilon

    rho = -np.inf
    stop = np.linalg.norm(g, np.inf) < epsilon_1
    mu = tau * np.max(np.diagonal(A))

    while not stop and k < kmax:
        k=k+1
        while (rho <= 0) and (not stop):
            delta_p = np.linalg.pinv(A+mu*np.eye(9)) @ g
            if np.linalg.norm(delta_p, ord=1) <= epsilon_2* np.linalg.norm(p, ord=1):
                stop = True
            else:
                p_new = p + delta_p
                rho_numerator = np.linalg.norm(epsilon, ord=2)**2 - np.linalg.norm(
    cost_func(p_new, inliers), ord=2)**2
                rho_denominator = delta_p.T @ (mu * delta_p + g)
                rho = rho_numerator / rho_denominator

                if rho > 0:
                    p = p_new
                    J = calculate_jacobian(cost_func, p, inliers)
                    A = J.T@J
                    epsilon = cost_func(p, inliers)
                    cost_values.append(epsilon)

                    g = J.T@epsilon
                    stop = np.linalg.norm(g, np.inf) < epsilon_1 or np.linalg.norm(
    epsilon, ord=2) <= epsilon_3
                    mu = mu * np.max((1/3, 1-(2*rho-1)**3))
                    v = 2
                else:
                    mu = mu * v
                    v = 2 * v
    return p, cost_values
```

To compute the Jacobean matrix, I add small perturbations to the parameters of p and then compute their impact on the cost function residuals. This concept is similar to the estimation of the gradients fo the LoG through the use of difference of Gaussians in previous lectures. That implementation is included below:

```
def calculate_jacobian(cost_func, h, inliers):
    # We are going to approximate the derivates by adding a small pertubation to see how
     that affects the cost function
    # So: we add a very small value to each component of h and compute the change vector
     and add that into the jacobian matrix.
    homography = h.flatten()
    residuals = cost_func(homography, inliers)

    # Jacobian matrix will have n cols and m rows:
    # [[df1/dp1 ----- df1/dpn], ........, [dfm/dp1, ------, dfm/dpm]]
    J = np.zeros((len(residuals), len(homography)))
    epsilon = 1e-6

    for h_idx in range(len(homography)):
        h_cpy = np.copy(homography)
        h_cpy[h_idx] += epsilon

        perturbed_residuals = cost_func(h_cpy, inliers)

        J[:, h_idx] = (perturbed_residuals - residuals)/epsilon
    return J
```

Lastly, the Levenberg-Marquadt algorithm optimizes the homography parameters over a cost surface. This is the re-projection error from the source domain to the target domain. The re-projection error is calculated as follows:

$$||X - f(\vec{p})||^2$$

$$X = \text{Points from the target domain}$$

$$f(p) = \begin{cases} f_1(p) = x' = \frac{xh_{11}+h_{12}y+h_{13}}{h_{31}x+h_{32}y+h_{33}} \\ f_2(p) = y' = \frac{xh_{21}+h_{22}y+h_{23}}{h_{31}x+h_{32}y+h_{33}} \end{cases}$$

The following is my python implementation of the cost function in Levenberg-Marquadt:

```
def cost_func(h, inliers):
    # h has the form [h11, h12, h13, h21, h22, h23, h31, h32, h33]
    # We want to minimize ||X - f(p)||^2
    source_points = inliers[:, 1]

    x_hat = inliers[:, 0, 0]
    y_hat = inliers[:, 0, 1]

    divisor = x_hat*h[6] + y_hat*h[7] + h[8]
    x_prime = (x_hat*h[0] + y_hat*h[1] + h[2])/divisor
    y_prime = (x_hat*h[3] + y_hat*h[4] + h[5])/divisor

    # Shape is (n, 2) with x,y horizontally collated together [[x1,y1], [x2,y2], ...]
    F = np.vstack((x_prime, y_prime)).T
    return (source_points - F).ravel()
```

5 Panorama Creation

To create the panorama image, I first find the homography that maps pairs together in the following order:

$$\begin{bmatrix} img1 \rightarrow img2 \\ img2 \rightarrow img3 \\ img3 \rightarrow img3 \\ img4 \rightarrow img3 \\ img5 \rightarrow img4 \end{bmatrix}$$

Therefore, we can map each image to the center image with the following homographies:

$$\begin{bmatrix} \boldsymbol{H}_{23}\boldsymbol{H}_{12} \\ \boldsymbol{H}_{23} \\ \boldsymbol{I} \\ \boldsymbol{H}_{43} \\ \boldsymbol{H}_{43}\boldsymbol{H}_{54} \end{bmatrix}$$

Using these homographies, I can find the area required for the background panorama canvas by calculating the coordinates of the warped corner points. The results of this warping is shown in Section 6.4. Finding the bounding box around these points gives us the canvas, while the x_min and y_min values will be the offset for moving the top-left corner of the canvas from some negative value to (0,0).

We can then apply the color from the images to the panorama canvas through the use of the transform image function that was developed in homework 2 that utilizes the inverse homography to avoid a sparse color mapping.

```
class Panorama():
    def __init__(self, img_list, homography_list):
        self.img_list = img_list
        self.homography_list = homography_list

        # Calculate the output size using the image corners
        # We then return an empty image of the correct shape to fit all points
        panorama_image, C_pn = self.find_corner_points()

        # Plot the corners from the image to make sure it is correct
        self.plot_corners(panorama_image, C_pn)

        # Fill the color into the panoramage canvas
        self.panorama_img = self.generate_output(panorama_image, C_pn)

    def get_image_results(self):
        return self.panorama_img

    def find_corner_points(self):
        corner_mats = np.zeros((len(self.img_list), 3, 4))
        for i, img in enumerate(self.img_list):
            # Creates a matrix of shape [3,4] where each entry is the hc coordinate of a
     corner point
            # Points are (x,y,1)
            corner_mats[i] = np.array((Point(0,0).hc, Point(img.shape[1], 0).hc, Point(
    img.shape[1], img.shape[0]).hc, Point(0, img.shape[0]).hc)).T

        C_prime = np.zeros((len(self.img_list), 3, 4))
        for i in range(len(self.img_list)):
            C_prime[i] = self.homography_list[i]@corner_mats[i]

        # Corner points are still (x,y,1)
        C_pn = C_prime / (C_prime[:, 2, :][:, np.newaxis, :] + 1e-6)
        x_min, y_min, x_max, y_max = np.min(C_pn[:, 0, :]), np.min(C_pn[:, 1, :]), np.
    max(C_pn[:, 0, :]), np.max(C_pn[:, 1, :])

        # Make the points into integers for image processing
        x_min, y_min, x_max, y_max = int(x_min), int(y_min), int(x_max), int(y_max)

        background_space = np.zeros((y_max-y_min, x_max-x_min, 3), dtype=np.uint8)
        return background_space, C_pn

    def transform_image(self, hc_coords, homography):
        # hc_coords is a meshgrid of the panorama image
        H_inv = np.linalg.inv(homography)

        # Get normalized warped HC
        src_hc_coords = (H_inv @ hc_coords.T).T
        src_hc_coords = src_hc_coords / (src_hc_coords[:, 2][:, np.newaxis] + 1e-6)
        return src_hc_coords

    def plot_corners(self, panorama_image, C_pn):
        C_pn_translated = np.zeros_like(C_pn)
        C_pn_translated[:, 0, :] = C_pn[:, 0, :] - np.min(C_pn[:, 0, :])
        C_pn_translated[:, 1, :] = C_pn[:, 1, :] - np.min(C_pn[:, 1, :])
```

```
53
54          # Take out just the x,y points
55          points = C_pn_translated[:, :2, :]
56          # Convert the list of points so that the output is [[y1, x1], [y2,x2], [y3,x3]]
    instead of them being split
57          # accross multiple rows and within a batch of images.
58          points = np.hstack(points)
59          points = np.column_stack((points[0], points[1]))
60
61          y_vals, x_vals = points[:, 0], points[:, 1]
62
63          # Fill in the corner points into the panorama image:
64          fig, ax = plt.subplots()
65          ax.imshow(panorama_image)
66
67          # point_idx = self.get_point_ordering_for_corner_printing(len(y_vals))
68          # print(point_idx)
69          # for i in range(0, len(point_idx), 4):
70          for i in range(0, len(y_vals), 4):
71              # Get a random color to fill in the polygon
72              color = [[random.randint(0, 255)/255, random.randint(0, 255)/255, random.
    randint(0, 255)/255]]
73
74              points = [[y_vals[i],x_vals[i]], [y_vals[i+1],x_vals[i+1]], [y_vals[i+2],
    x_vals[i+2]], [y_vals[i+3],x_vals[i+3]]]
75              polygon = patches.Polygon(points, closed=True, color=color[0], fill=False,
    linewidth=2)
76              ax.add_patch(polygon)
77
78          plt.savefig("corners.jpg")
79
80      def generate_output(self, panorama_image, C_pn):
81          # Offset to subtract to points in images to bring the corners within the final
    image coordinate frame
82          x_min = np.min(C_pn[:, 0, :])
83          y_min = np.min(C_pn[:, 1, :])
84
85          h, w, c = panorama_image.shape
86          panorama_image = panorama_image.reshape(-1, 3)
87
88          # Create pixel coordinate grid for the target image (y, x) -> (x,y)
89          y_range, x_range = np.arange(h), np.arange(w)
90          y_range = y_range + y_min
91          x_range = x_range + x_min
92          x_1, x_2 = np.meshgrid(x_range, y_range) # Now we have (x, y)
93
94          # Flatten the grid and create homogeneous coordinates (x, y)
95          real_coords = np.vstack([x_1.ravel(), x_2.ravel()]).T
96          ones_column = np.ones((real_coords.shape[0], 1))
97          panorama_hc_coords = np.hstack((real_coords, ones_column))  # Homogeneous
    coordinates
98
99          for homography, img in zip(self.homography_list, self.img_list):
100             img_h, img_w, img_c = img.shape
101
102             # Get the panorama image in terms of img1's coordinate frame
103             warped_img_hc = self.transform_image(panorama_hc_coords, homography)
104             img_points = warped_img_hc[:, :2]
105
106             # Apply offset to bring all the points to the translated (0,0) based frame
107             src_x = img_points[:, 0]
108             src_y = img_points[:, 1]
109
110             valid_mask = (src_x >= 0) & (src_x < img.shape[1]) & (src_y >= 0) & (src_y <
     img.shape[0])
111             valid_x = src_x[valid_mask].astype(int)
112             valid_y = src_y[valid_mask].astype(int)
113             idx = valid_y * img_w + valid_x
114
115             img = img.reshape(-1, 3)
116             panorama_image[valid_mask] = img[idx]
117
118         panorama_image = panorama_image.reshape(h,w,c)
```

```
119
120         return panorama_image
```

6 Results

6.1 Original Images

For convenience and better document layout, I only include the first four images for this section. All five images will be shown in the panorama view. Additionally, the final image can be submitted separately if requested.

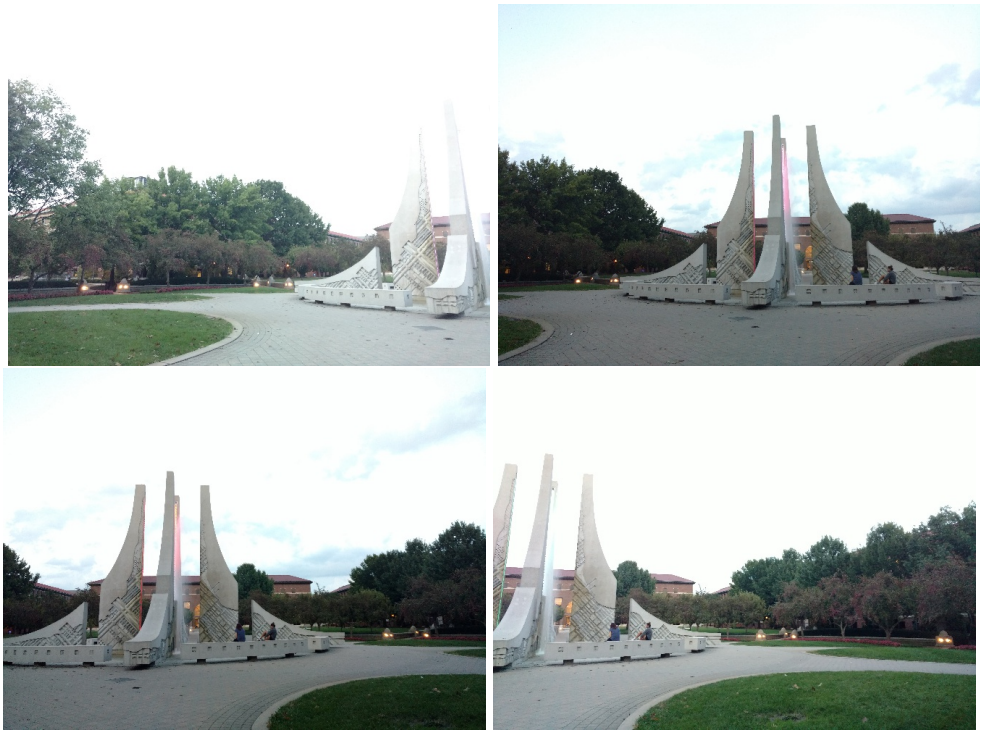

Figure 1: The source images for the engineering fountain provided by the instructor.

Figure 2: My own images of a rock-climbing wall for the panorama creation.

6.2 SIFT Keypoint Matches

For convenience, I will only include the first 4 image pairs in this section. Additionally, the final image can be submitted separately if requested. In this section, the colors of the lines are only included to help seperate the points and hold no underlying meaning.

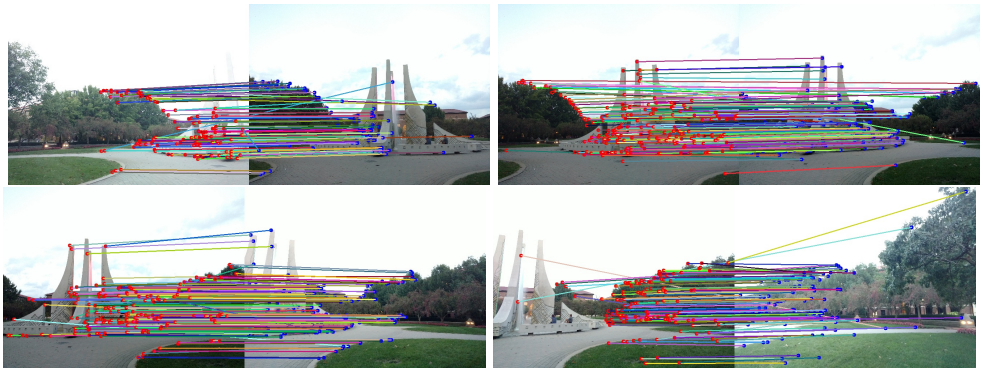

Figure 3: The sift correspondences on the provided engineering fountain images.

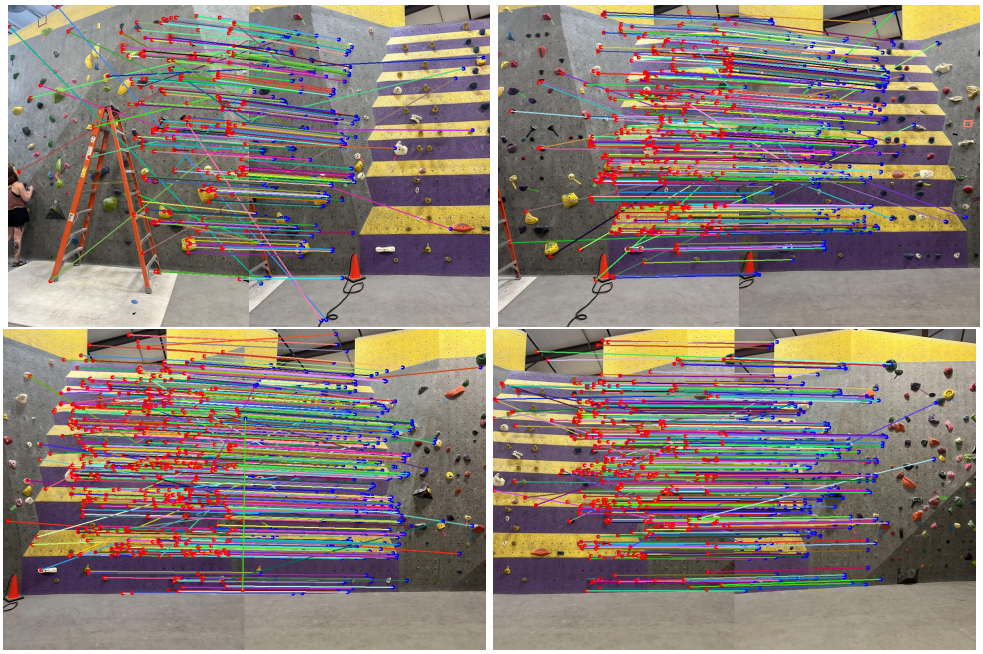

Figure 4: The sift correspondences on my own climbing wall images.

6.3 RANSAC Results

For convenience, I will only include the first 4 image pairs in this section. Additionally, the final image can be submitted separately if requested. In this section, the inlier matches after running RANSAC are shown in green, while the red lines represent false/outlier correspondences.

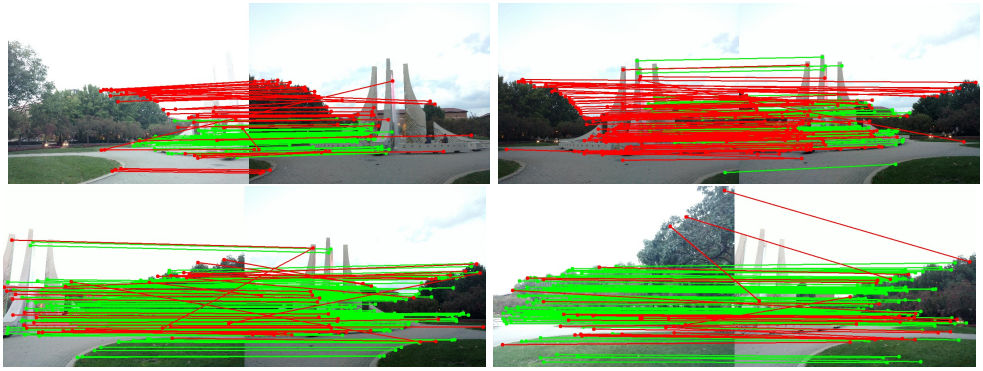

Figure 5: The RANSAC inlier and outlier correspondences on the provided engineering fountain images.

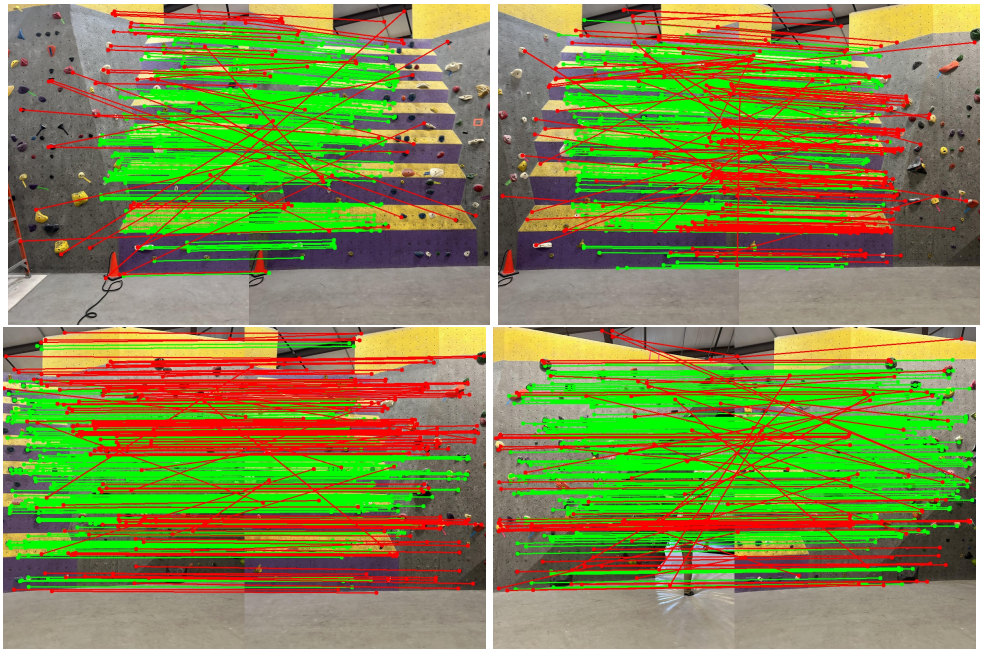

Figure 6: The RANSAC inlier and outlier correspondences on my own climbing wall images.

6.4 Corner Points for the Panorama

6.4.1 RANSAC+Linear Least-Squares corner points

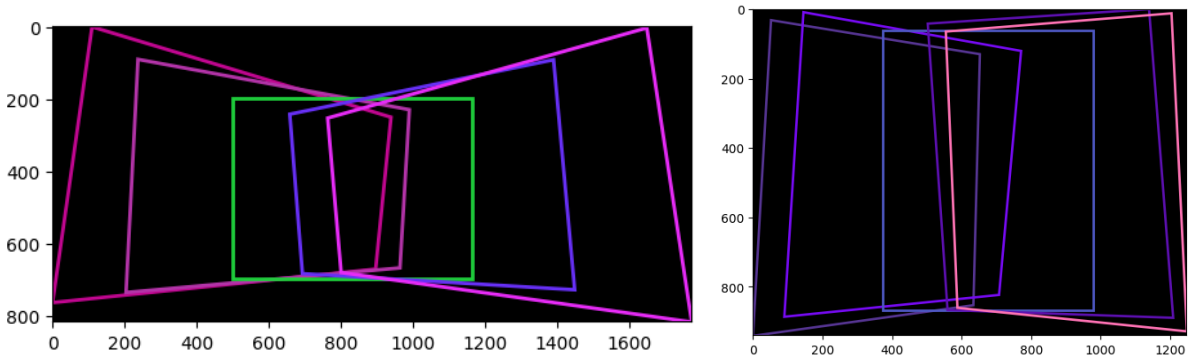

Figure 7: The panorama corner points calculated through the use of the homography from running RANSAC with linear least-squares optimization.

6.4.2 Levenberg-Marquadt corner points

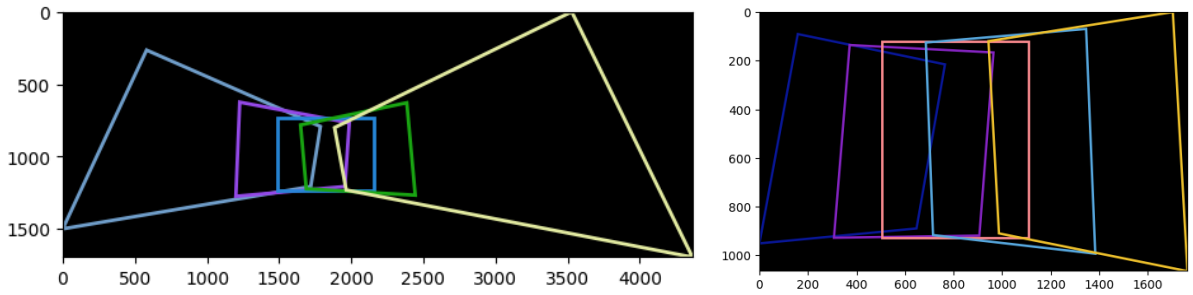

Figure 8: The panorama corner points calculated through the use of the homography from running non-linear least-squares optimization with Levenberg-Marquadt.

6.5 RANSAC Panorama

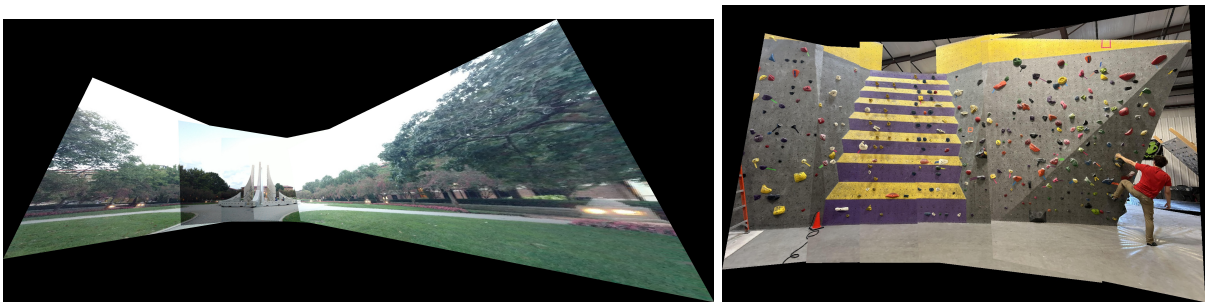

Figure 9: Panorama image created through the 5 provided and self-taken images using RANSAC and linear least-squares optimization.

6.6 Scipy based Levenberg-Marquardt Panorama

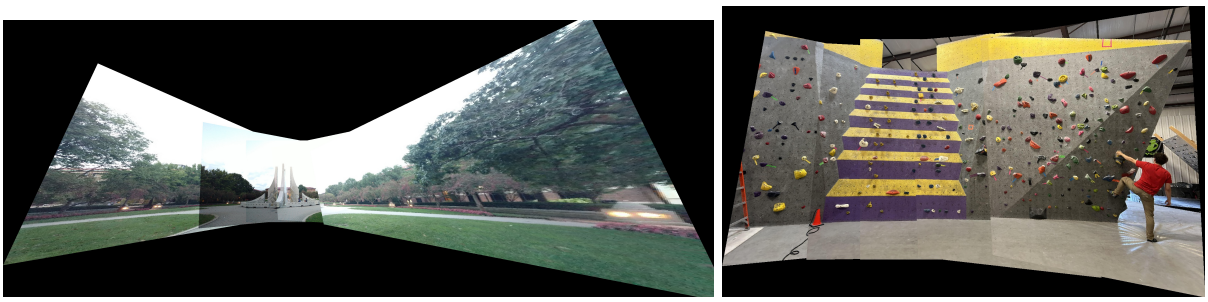

Figure 10: Panorama image created through the 5 provided and self-taken images using Levenberg-Marquadt non-linear least-squares optimization.

It was interesting that while I did notice some improvements from running the Levengerg-Marquadt algorithm for these image correspondences, the result did not largely improve as I thought it would. This demonstrates the power of the RANSAC algorithm to accurately determine inlier points for use in the homography estimation.

7 BONUS: My own Levenberg-Marquadt Panorama results

7.1 Image results:

Included below are the panorama images for my own implementation of the BONUS self implementation of Levenberg-Marquadt.

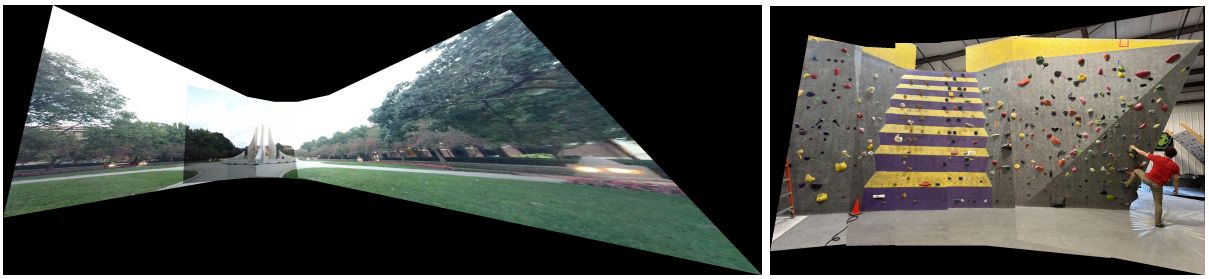

Figure 11: The panorama results using my own implementation of the Levenberg-Marquadt algorithm for non-linear least squares optimization.

The results of my panorama images are consistent with scipy's least squares implementation: they barely improved the results produced by the RANSAC implementation. A next step would be to test the RANSAC + linear least-squares and LM implementation with more complex correspondences to see if the non-linear least-squares solution can find a better panorama over the most complex image matching space.

7.2 Error graph:

Additionally, I plot below the cost function values over the time steps required for Levenberg-Marquadt. I report two pairs per image type (fountain vs climbing). These results did not demonstrate any significant decrease in the error function over the trials; however, the cost function values were consistent with the error results returned through scipy's least-squares implementation:

Fountain Image Correspondences:

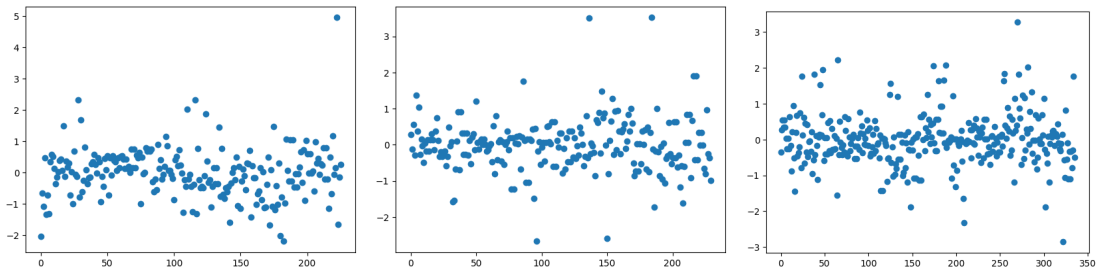

Figure 12: Three example graphs of the error values over all the trials of my Levenberg-Marquadt implementation for the fountain images.

Climbing Wall Image Correspondences:

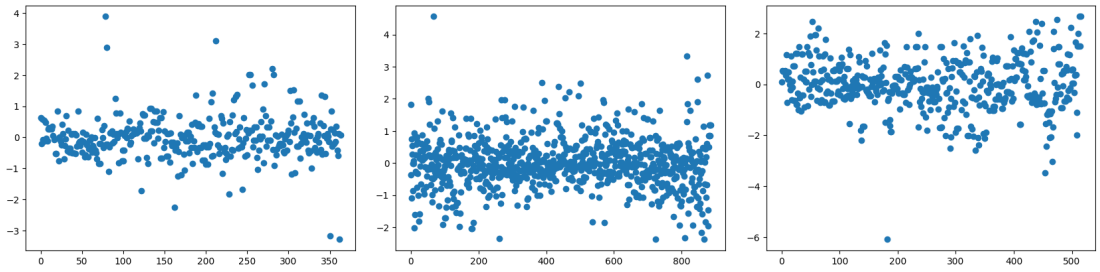

Figure 13: Three example graphs of the error values over all the trials of my Levenberg-Marquadt implementation for the climbing wall images.

8 Example code to run my scripts:

Included below is an example of the code that I used to run the panorama generation with Scipy's Levenberg-Marquadt non-linear least-squares implementation.

```
for key in keys:
    image_names = [key+str(i+1)+".jpg" for i in range(5)]

    image_homographies = []
    img_list = []
    for i in range(4):
        # I want the homographies for: 0->1->2 and 4->3->2
        if i < 2:
            # 0->1->2
            img1_path = input_prefix+image_names[i]
            img2_path = input_prefix+image_names[i+1]
        elif i==2:
            # All images are mapped to the middle image so we don't need to compute the homography from 2->2
            image_homographies.append(np.eye(3))

            # at i=2, we add the Identity matrix, then also get img 4->3
            img1_path = input_prefix+image_names[i+1]
            img2_path = input_prefix+image_names[i]
        else:
            # 4<-5
            img1_path = input_prefix+image_names[i+1]
            img2_path = input_prefix+image_names[i]

        # Get matching points for pairs of images
        match_list = get_SIFT_matching_points(img1_path, img2_path)

        # Save keypoint matches to a file
        # print_SIFT_matches(img1_path, img2_path, match_list, output_prefix+"SIFT_Matches/"+key+str(i)+"_"+str(i+1)+".jpg")

        # Apply RANSAC to get the largest set of inliers:
        inlier_percent, best_points = RANSAC_homography_estimation(sigma, match_list)
        match_set = set(match_list)
        best_point_xy = []
        for point_pair in best_points:
            point1 = (point_pair[0].hc[0], point_pair[0].hc[1])
            point2 = (point_pair[1].hc[0], point_pair[1].hc[1])
            best_point_xy.append((point1, point2))

        best_point_set = set(best_point_xy)
        bad_points_set = match_set - best_point_set

        # Print the inliers & outliers from ransac
        # print_RANSAC_matches(img1_path, img2_path, best_point_set, bad_points_set, output_prefix+"RANSAC/"+key+str(i)+"_"+str(i+1)+".jpg")

        # Get points for homography estimates
        source = [point_pair[0] for point_pair in best_points]
        target = [point_pair[1] for point_pair in best_points]

```

```
        # Compute homographies for those point pairs
        homography = Homography().estimate_projective_homography(source, target)

        # Apply Levenberg-Marquadt
        # Transform the homography into a 1,9 tensor
        optimal_homography = homography.flatten()
        inliers = np.array(best_point_xy)
        optimal_homography = least_squares(cost_func, optimal_homography, args=(inliers,), method='lm')
        optimal_homography = optimal_homography.x.reshape((3,3))

        # Homographies will be in the following order: h_12, h_23, I, h_43, h_54
        # We need to unpack the homography from the optimizedResult object by getting the x value
        image_homographies.append(optimal_homography)

    for image_name in image_names:
        img_list.append(np.array(Image.open(input_prefix+image_name)))

    # Map the homographies to the center image
    homography_list = []
    homography_list.append(image_homographies[1]@image_homographies[0])
    homography_list.append(image_homographies[1])
    homography_list.append(image_homographies[2])
    homography_list.append(image_homographies[3])
    homography_list.append(image_homographies[4]@image_homographies[3])


    panorama_img = Panorama(img_list=img_list, homography_list=homography_list).get_image_results()
    panorama_img = cv2.cvtColor(panorama_img, cv2.COLOR_RGB2BGR)
    cv2.imwrite(output_prefix+"Scipy_LM/"+image_name[:-5]+"pano.jpg", panorama_img)
```