ENPM673 Project 2

Brendan Neal

March 8, 2023

Contents

1 Introduction

This project has students using the concepts of homography and Hough transform in order to identify the corners of a piece of paper and calculate the camera's roll, pitch, yaw, X, Y, and Z translations over the course of a video. Additionally, this project has students use feature matching and homography in order to stitch four images together to form a panorama.

2 Problem 1

Given a video, we have to estimate camera pose using homography in over the course of a video.

2.1 Process

2.1.1 Image Processing Pipeline

The first step to solving this problem is to design an image processing pipeline before you compute the Hough transform. While the video is opened, first I convert the image to gray scale using cv2.cvtColor(). Next, I use cv2.GaussianBlur() to blue the gray-scale image using a 7x7 kernel. After that, I use cv2.Canny() on the Gaussian blurred image in order to detect the edges in the image frame. I set my lower threshold to 100, and my upper threshold in cv2.Canny() to be 300. These lower and upper thresholds gave me the best edge detection with minimum extraneous noise and no dropping of the edges for all frames.

2.1.2 Hough Transform Pipeline

In order to perform the required Hough transform steps to identify and characterize each of the four paper edges, I created three separate functions. The first function is a HoughTrans() function that takes the binary edge-detected image as an input and outputs the Hough accumulator (H), the tested Ds and tested Thetas. Within this function, I would get the height and width of the binary image. Next, I needed to initialize my accumulator. To do this, I needed to calculate all of my possible Ds and Thetas. The maximum D a line could be in an image is the diagonal length of the image. Using the height and width of the image, I calculated the diagonal length. Then, using np.arange() I created an array with a range of Ds from -Diagonal Length to +Diagonal Length. Next, I used np.arange() again to initialize all possible thetas from 0 to 180 in radians. Using these two arrays, I initialized H to be an array of zeros of size (All Ds x All Thetas). Next, I would pull the X and Y coordinates from an edge detected image using np.nonzero(). I repeated this process for every X and Y coordinate on an edge: for every edge point, for every theta, I would calculate D. Then, in the accumulator, I would add 1 vote at the positional index of D and Theta.

Once I have a completely full accumulator, lines occur at the peak indices of the accumulator. Next, I developed a FindHoughPeaks() function to find the peaks in the Hough Accumulator. This function takes the accumulator, number of desired peaks, and the neighborhood size as inputs and outputs the indices where the peaks occur. First, I initialize the peak indices array and copy the accumulator so as to not change the contents of the accumulator when finding peaks. For each of the desired peaks, I index the max value in the accumulator copy, append those indices to the peak indices array, and then set the accumulator copy value at that index to zero in order to look for another peak in the for loop. Finally, I added some logic to ensure that I wouldn't find lines that appear super close to already detected lines using the neighborhood size input. Essentially, the logic states, "If a peak in the accumulator has been detected and you find another accumulator peak within the neighborhood size number of indices, do not include in the peak identification."

Once I have the peak indices, I made a function that would generate the Hough lines to be displayed on screen as well as calculate the intersection points later. I knew that to display lines on screen, I would have to use cv2.line(), which takes the starting (X,Y) and ending (X,Y) values as critical inputs. My CreateHoughLines() function takes the image frame, accumulator peak indices, possible Ds, and possible Thetas as inputs and it would draw the lines in the image frame as well as output an array called LineInfo. First, I initialize four arrays: starting Xs and Ys and ending Xs and Ys. Then, I initialize a 4x6 array called LineInfo(). Then, for each of the four peaks, I would index the D and Theta at which the peak appears. Then I would use the D and Theta for that peak to calculate two constants: A and B. The constants are calculated using the following equations:

$$A = cos(\theta)$$

$$B = sin(\theta)$$

Once I have those constants, I can calculate the Cartesian point where that line appears with the following equations:

$$X_0 = A * D$$

$$Y_0 = B * D$$

Since this gives me only a single point, I need to extend the line to fit the page, as well as initialize my start and end points for display. To initialize my start points, I used the following equations:

$$X_{start} = X_0 + 2500(-B)$$

$$Y_{start} = Y_0 + 2500(A)$$

To initialize my end points, I used the following equations:

$$X_{end} = X_0 - 2500(-B)$$

$$Y_{end} = Y_0 - 2500(A)$$

I chose an arbitrary 2500 because we do not know the exact start and end points yet (we do not know the X and Y coordinates of the corners). Thus, I wanted to expand the lines so they appear to extend "off the image frame." since the line itself is the same, and the intersections are what matter the most. Finally, using my start and stop points, I would calculate the slope and Y intercept for each of the lines to be used later in finding the corners.

2.1.3 Identifying Corners Pipeline

I created multiple functions in order to sort and identify my lines. I noticed that the indices of the "strongest lines" would change per frame. I.E. The top edge of the paper would not always be the first peak indexed by my find peaks function. Thus, I had to create two functions to sort and identify the lines. I reasoned for this project, I could sort them based on Y intercept. The top edge of the paper would always have the smallest Y intercept, the bottom edge of the paper would always have the second smallest Y intercept, the left edge of the paper would always have the second largest y intercept, and the right edge of the paper would have the largest y intercept. See Figure 1 below for more clarification:

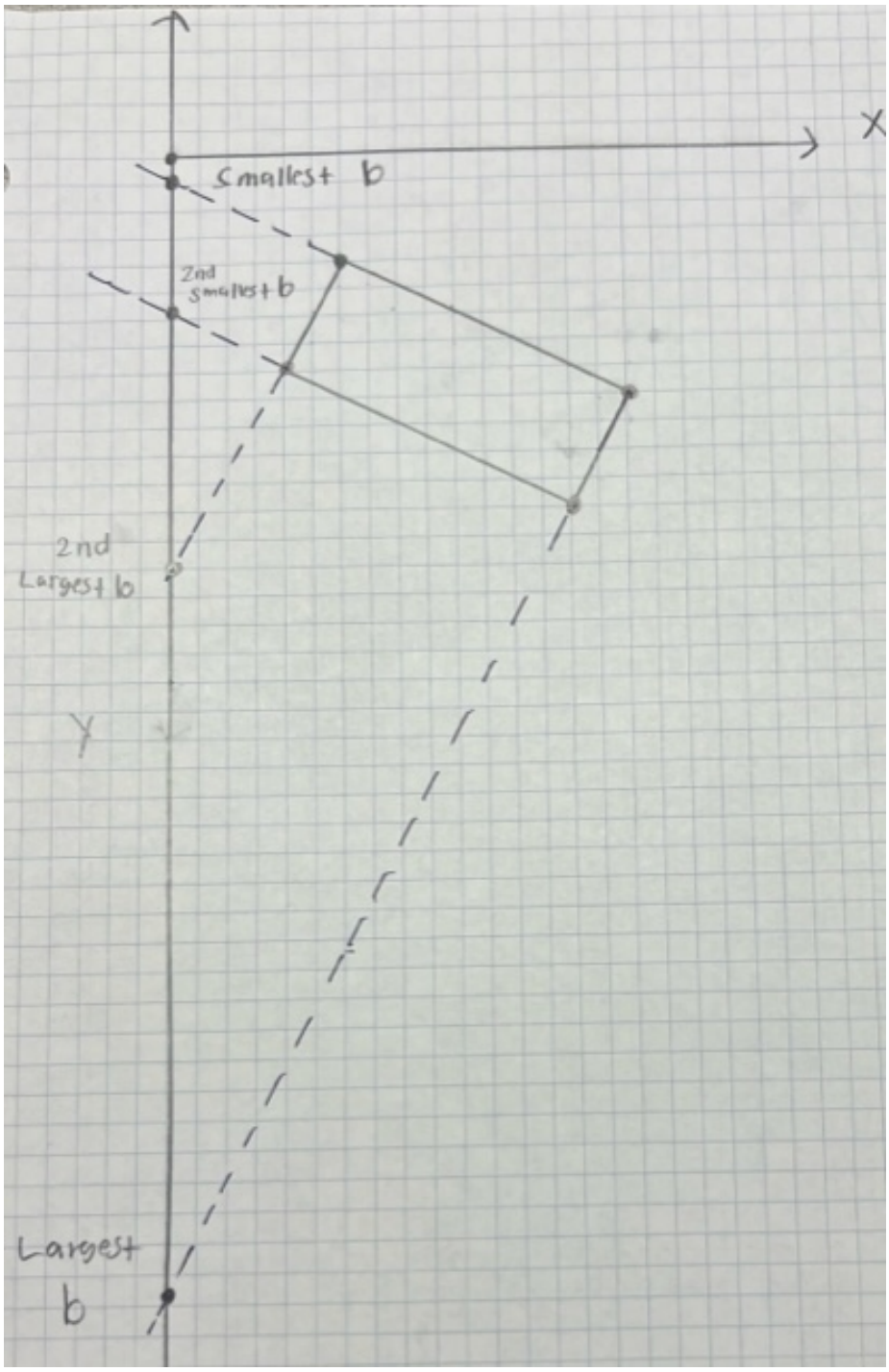

Figure 1: All Y Intercept Clarification

Knowing this, I created a SortLines() function that would take the LineInfo array from before and sort them from smallest to largest based on the Y intercept. Next, I created an InitializeLines() function that would set the sorted line information into their respective known lines. See Figure 2 below for more clarification:

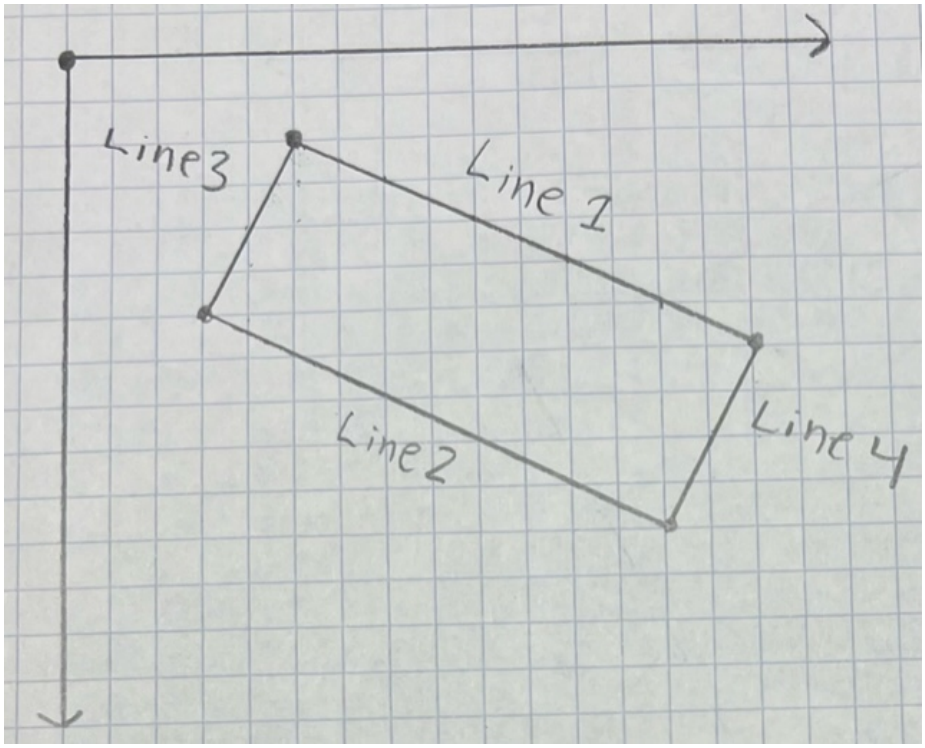

Figure 2: Line Definitions

To solve the the X and Y coordinates for each of the corners, I created a SolveIntersections() function that would take two lines' information as an input, and output the X and Y coordinates of the intersections. This function initializes matrices from the Line Information and solves for the intersections using Cramer's Rule. Now to get all corners, I applied the function four times: Between Lines 1 and 3, Lines 2 and 3, Lines 1 and 4, and Lines 2 and 4. Finally, I would now display the corners on screen for a visual representation.

2.1.4 Camera Pose Estimation Pipeline

The first step to estimating the camera pose is to define my new coordinate frame. To do so, I chose the top-left corner of the paper to be my new origin. Then I performed the following calculation for all of the corners:

$$X_{PaperFrame} = X_{Current} - X_{TopLeft}$$

$$Y_{PaperFrame} = Y_{Current} - Y_{TopLeft}$$

Next, using the dimensions of the paper, I set up the corner coordinates in the world frame using the top left corner as my origin (0,0). I converted the dimensions to mm to match the camera intrinsics dimensions. Now that I have 4 corresponding points, I can now calculate the homography from the pixel coordinates to the world frame. I created a function that would take the corresponding pixel corner points and world frame corner points as inputs and outputs the homography matrix H. The function takes these corresponding coordinates and formulates matrix A of the following form:

$$\begin{bmatrix} X_{World} & Y_{World} & 1 & 0 & 0 & 0 & -X_{Pixel} * X_{World} & -X_{Pixel} * Y_{World} & -X_{Pixel} \\ 0 & 0 & 0 & X_{World} & Y_{World} & 1 & -Y_{Pixel} * X_{World} & -Y_{Pixel} * Y_{World} & -Y_{Pixel} \end{bmatrix}$$

Because we have four pairs of corresponding points, Matrix A has dimensions of 8x9. I then use SVD to find the eigenvector of ATA with the smallest eigenvalue, which is the estimated homography matrix H. I then reshape H to be a 3x3 matrix rather than a 9x1 array and normalize the values by dividing every element by the value of the last element before returning.

Once I have my homography matrix, I have to decompose it to get my rotation and translation matricies. I created a DecomposeHomography() function that takes the estimated homography matrix (H) and camera intrinsic matrix (K) as inputs and outputs the rotation and translation matrices. Obtaining the rotation and translation matrices involves solving the following equations:

$$H = \lambda * K * [r_1, r_2, t]$$

Since I know that r1 and r2 are unit vectors, that means their magnitude is 1. And since lambda is just a constant, I can solve the following equation for lambda:

$$\lambda = \frac{1}{(NormofFirstColumnofK^{-1}H)}$$

To get a more accurate lambda, I perform the same calculation for r2 and take the mean. Now that I have lambda, I can solve for r1, r2, and t using the following equations:

$$T = \lambda * (FirstColumnofK^{-1}H)$$

$$r_1 = \lambda * (SecondColumnofK^{-1}H)$$
$$r_2 = \lambda * (ThirdColumnofK^{-1}H)$$

To complete the rotation matrix, I took the cross product of r1 and r2 and appended it to my rotation matrix to get [r1, r2, r3].

To compute the roll pitch, and yaw of from the rotation matrix, I used the SciPy toolbox "Rotation" to return the Euler angles of the camera in the coordinate frame of the paper. I then run all of these functions in a loop, save the important data for plotting, then plot the data once the whole video has been analyzed.

2.2 Results

2.2.1 Edge Detection

Figure 3 below is a frame of the video after undergoing edge detection.

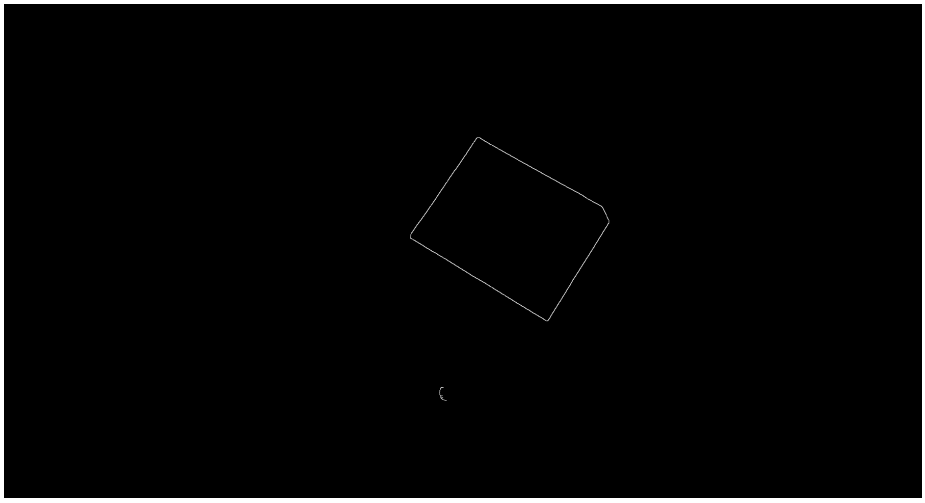

Figure 3: All Edge Detection

2.2.2 Hough Lines Detection

Figure 4 is a frame of the video after undergoing the series of Hough lines detection functions.

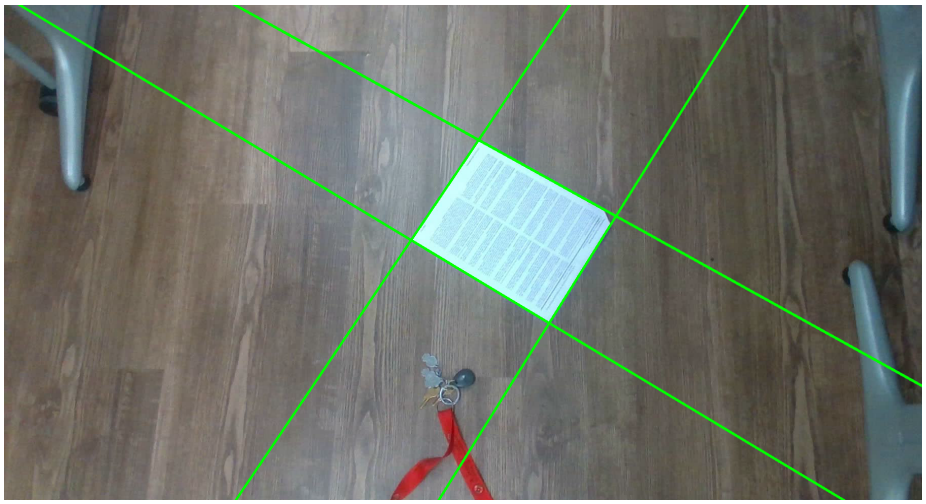

Figure 4: Hough Lines

2.2.3 Corner Detection

Figure 5 is a frame of the video after undergoing the series of corner detection functions.

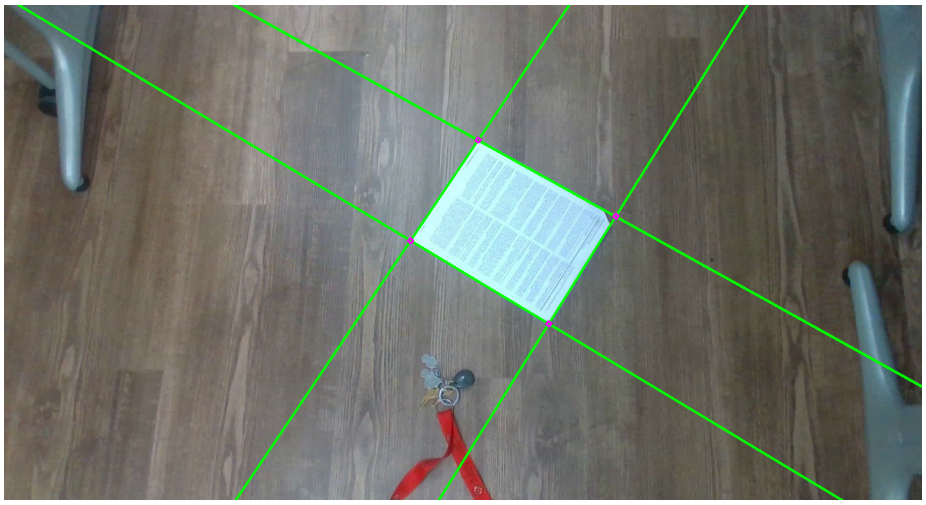

Figure 5: Hough Lines with Corner Detection

2.2.4 Camera Pose Estimation

Figure 6 is a plot showing the camera pose state over each frame of the video. These results make sense because the coordinate frame I set for the paper has X along Line 1, Y along Line 3, and Z into the page. As the camera gets closer, the z value becomes less negative. Additionally, the yaw constantly hovering around -30 degrees makes sense as well, since the camera is rotated along the -z axis.

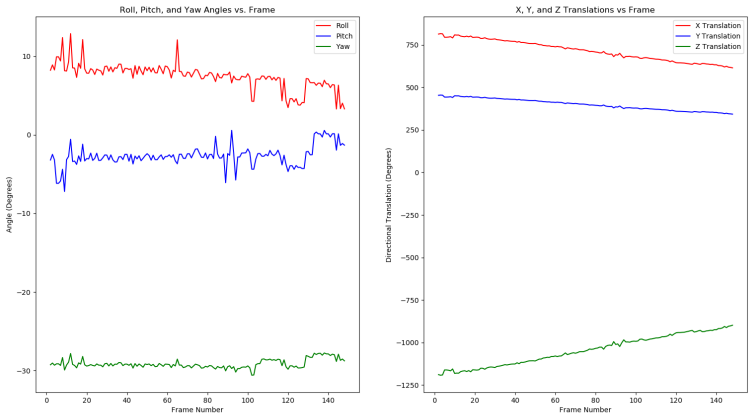

Figure 6: Plot Showing X, Y, Z, and Euler Angles as a Function over Frame Number

2.3 Problems Encountered

2.3.1 Edge Detection

I had to set the boundaries of my edge detection to be relatively loose because sometimes all edges in the image frame would drop out, causing the rest of the program to fail. As a result, I would rely more on the rest of my functions to identify and characterize the strong lines.

2.3.2 Hough Lines Detection

One of the problems I had during my Hough lines detection process was that I was finding the same peak four times, rather than finding four unique peaks. To solve this, I changed the algorithm to set a found peak to 0 after I saved the information for that peak before looping back through and finding another peak.

Another problem I was having was that I was finding lines "close to" already found lines. To fix this, I had to include the neighborhood algorithm defined earlier.

2.3.3 Corner Detection

For my corner detection algorithm, I had no way to identity which line was which after I discovered that the indices changed for each detected line. I originally had each line as different colors, and the top edge would not always be the "red" line. As a result, I implemented the line sorting algorithm introduced earlier.

2.3.4 Camera Pose Estimation

For my camera pose estimation algorithm, I originally was getting 8 outputs for the homography matrix, but then I noticed I had swapped my ATA matrix in my code, and fixed it.

When calculating my roll, pitch, and yaw, I had noticed that I was getting random swapping of signs around 90 degrees. This was caused by the fact that I was using arc-tangent to calculate my roll, pitch, and yaw. To fix this, I used the SciPy toolbox to more accurately compute roll pitch, and yaw from my rotation matrix.

3 Problem 2

This problem had me use feature matching and homography in order to stitch four images together to form a panorama.

3.1 Process

3.1.1 Feature Detection

To detect the features, I read each of the four images into my workspace. Next, I converted all four images to grayscale. Then, I initialized my ORB object. Next, I identified the key points for each of the four images using ORB.detect(). Afterwards, I attached the descriptors of the key points using ORB.compute(). To display the key features, I used the cv2.drawKeypoints() function.

3.1.2 Feature Matching

In order to match the features, I decided to use a brute force matcher. I initialed my brute force mather using cv.BFMatcher() function. Then I matched the key points between images 1/2, 2/3, and 3/4 using the BF.match() function. To visualize the matched features, I used BF.knnMatch(). Once I matched the features, I would initialize my correlation matrices for each pair of images by indexing the match.queryIdx and the match.trainIdx points to get the pairs of matched images. I would append all the matched points into an array and convert said array into a matrix to be used in my homography calculation.

3.1.3 Homography Calculation

To calculate my homography, I used the function I developed in the previous section, but adjusted it to correctly parse the data from a matrix rather than additional points. Additionally, since the number of points can change, I used a for loop to index and append the rows following this format:

$$\begin{bmatrix} X_1 & Y_1 & 1 & 0 & 0 & 0 & -X_2 * X_1 & -X_2 * Y_1 & -X_2 \\ 0 & 0 & 0 & X_1 & Y_1 & 1 & -Y_2 * X_1 & -Y_2 * Y_1 & -Y_2 \\ .. & .. & .. & .. & .. & .. & .. & .. & .. \end{bmatrix}$$

I then use SVD to find the eigenvector of ATA with the smallest eigenvalue, which is the estimated homography matrix H. I then reshape H to be a 3x3 matrix rather than a 9x1 array and normalize the values by dividing every element by the value of the last element. This function returns the estimated homography for a set of corresponding points.

Due to the presence of outliers, I had to implement my RANSAC algorithm from the last project to take samples of the image and compute homography iteratively to find a "best model." My RANSACHomography() function takes the corresponding feature matrix, a set threshold, and the sample size as inputs. The function would randomly sample four corresponding point pairs (since that is the minimum number required to calculate homography). It would take those four points, calculate homography, and if the number of inliers was greater than the current inlier max, set that model as the best model. To calculate the number of inliers, I had to create an error calculation function that would take the corresponding feature list and the iteration's homography matrix as inputs. From there, it would index each point from the corresponding feature list from Image 2 and calculate an estimate of its corresponding point in Image 1 using the iteration homography matrix. Below is the equation:

$$Point2_{Est} = H_{iteration} * Point1$$

Once I have the estimated point 2, the function would calculate the error as a distance from where the actual corresponding point is and the estimated corresponding point in Image 2 is. If this distance was less than a set threshold,

it would be added to the inlier count, which is used to determine whether the model is ideal or not. The function would continue to search for a best model until the iteration number surpassed the max iterations, or it reached a manually set iteration count of 10000. The hard stop iteration count comes is to prevent the RANSAC function from running an unnecessarily long time.

3.1.4 Image Stitching

To complete the image stitching between the pairs of images, I used the cv2.warpPerspective() function. For example, I would warp the perspective of image 2 into image 1 using the calculated homography matrix. As for the size, I would keep the height of the resultant image the same, but extend the length of the resultant image to be the sum of the two input image widths. I would then layer the first image over top of the resultant image in order to create a smooth transition between the two images. This works because the warped image is based on the homography, which is based on the matched features.

In order to stitch all four images together, I initialized a picture frame the length of the first image, plus the lengths of the three resultant images. I would then index and layer the resultant images on top of each other with the intent of creating one smooth panorama.

3.2 Results

3.2.1 Feature Detection

Figure 7 shows the detected features of Image 1.

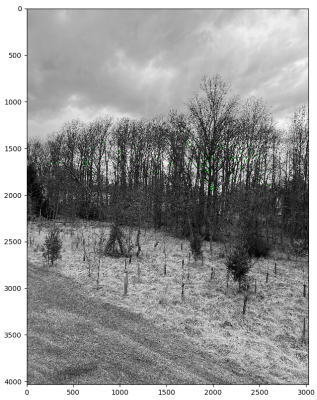

Figure 7: Detected Features

3.2.2 Feature Matching

Figure 8 shows the matched features of Images 1 and 2.

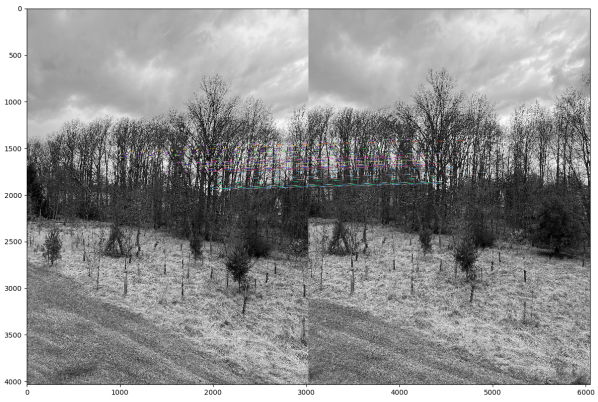

Figure 8: Matched Features Between Images 1 and 2

3.2.3 Image Stitching

These results are less than ideal. Please see Problems Encountered: Image Stitching - What Might Have Gone Wrong for more information.

Figure 9 shows the results of images 1 and 2 being stitched together.

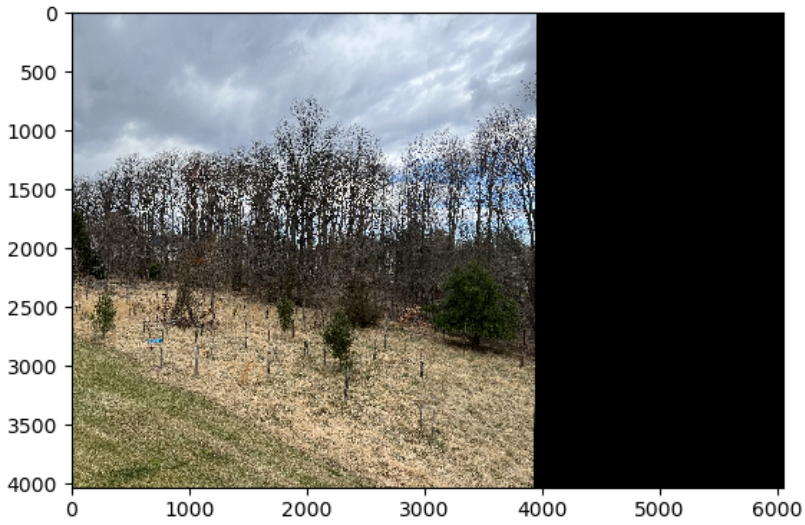

Figure 9: Images 1 and 2 Stitched

Figure 10 shows the results of images 2 and 3 being stitched together

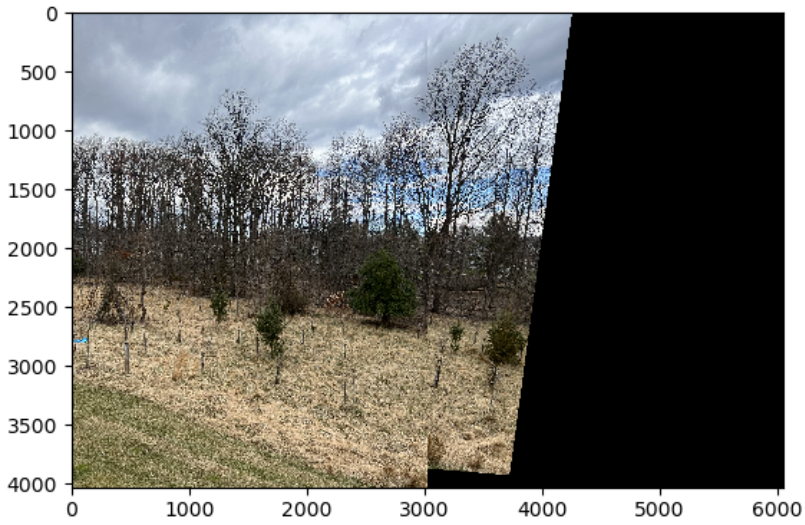

Figure 10: Images 2 and 3 Stitched

Figure 11 shows the results of images 3 and 4 being stitched together.

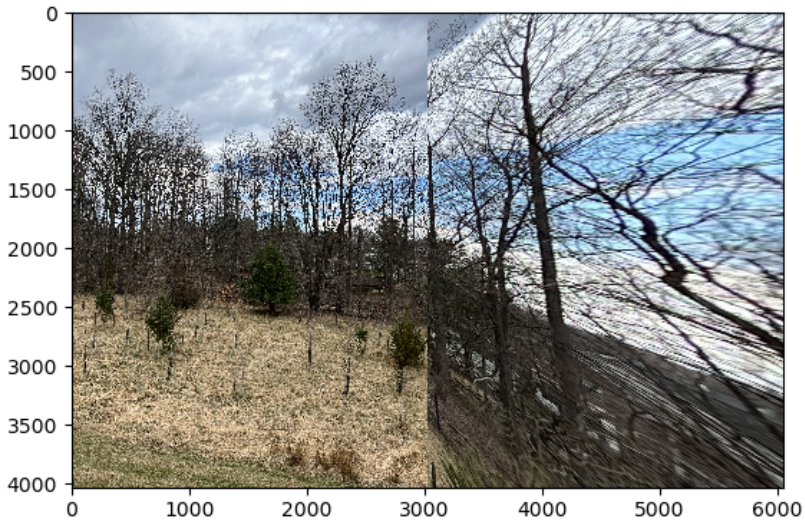

Figure 11: Images 3 and 4 Stitched

Figure 12 shows the final panoramic image.

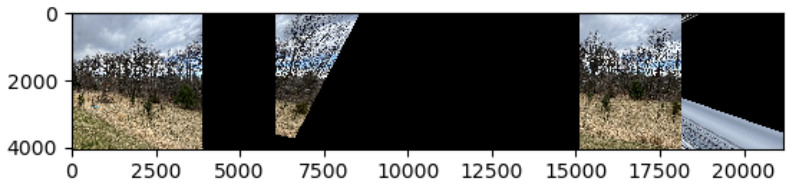

Figure 12: Final Panoramic Image

3.3 Problems Encountered

3.3.1 Feature Detection

Upon trying to use SIFT for my feature detection, I discovered that the SIFT license had expired for later versions of Python and OpenCV. As a result I had to use ORB, which is essentially OpenCV's version of SIFT.

3.3.2 Feature Matching

I had to implement a lot of data organization algorithms to convert the feature objects into usuable numbers for the homography calculation. Additionally, for calculations, I found it easier to use just BF.match() but for visualization it was easier for me to use BF.knnMatch().

3.3.3 Homography Calculation

The biggest challenge that came with creating my RANSAC homography was changing the error calculation function. Originally I only calculated the point estimation for a single point, but to get a more accurate, working inlier count, I had to project all points. From there, my RANSAC search algorithm worked properly.

3.3.4 Image Stitching - What Might Have Went Wrong

Obviously the results are not what were expected. Instead of discussing problems encountered in this section, I will discuss possible sources of error. First of all, the stitching of images 1 and 2 worked fairly well. This proves that my RANSAC and calculate homography functions were working as intended, as they produced one good result. I noticed that there were going to be some problems when the number of inliers for stitching images 1 and 2 averaged out to be between 165 and 185 inliers. While running the RANSAC homography for images 2 and 3, the average number of inliers fell between 75 and 90. This produces a stitched image of lower quality, but not too far from what it is supposed to look like. The main problem lies in stitching images 3 and 4. The average number of inliers was between 20 and 30, and this consistently returns poor results. If you observe the drawn matches between images 1 and 2, 2 and 3, and 3 and 4, the number of good feature matches decreases dramatically. Hence, the models returned also decrease dramatically in quality. Despite this, the fact remains that the problem might lie in the matched features, and not in any of my developed functions. I tried other feature matching functions, but none of them worked as well as ORB.

As for stitching the images together, the main problem lies in the large black regions outputted by the warpPerspective() function. I could not figure out how to properly get rid of the black in between space. I tried layering the stitched images in different orders, but I would always lose information. Furthermore, I tried indexing the nonzero values but the sizes would not align with the final image. Finally, I took an entirely different approach to stitching. I stitched images 1 and 2 together, and images 3 and 4 together. I then ran feature detection, matched features, and ran RANSAC homography between the two stitched images. Something was wrong here because my max inliers once again fell between 20-30. When I used warpPerspective() again, I still had information lost or a large black square separating the two images. This fault lies once again in my choice of feature matching software as well as my inability to use warpPerspective() correctly for more than two images.