Nicholas Novak

ENPM 673 Project 2
Homography, Hough Lines, and Stitching

Problem 1:
This problem asked us to use Hough transforms to extract edges and corners from a known paper on the ground in a video. Then using homography, we could get the rotation and translation of the paper. I did this problem second so I recycled my homography calculator from problem 2 here. Too see how it works, see my report on problem 2 below. To get the hough accumulator, which was the voting schema on the edges, I would create an accumulator with dimensions (-90:90, diagonal length of frame). This would store rho (sometimes referred to ad d) and theta values corresponding to normal lines from the origin. Then, I lopped through all nonzero pixels in my edge image and calculated the corresponding rho to each theta from -90 to 90 using $rho = x * \cos(theta) + y * \sin(theta) + diagonal\_length$. By adding the diagonal length, I made sure that the value was always positive, and I avoided errors where my index would wrap around to the front of my matrix. I would later remove this through subtraction when calculating the x and y values.

Next, I found the maximum values in this matrix by doing a k-nn search with a defined radius. I found the optimum to be around 30 pixels. This would search through for a specified number of peaks and grab the maximum for the array and then ignore it and the surrounding points when searching for the next maximum. This gave me pretty consistent results. The rho and theta values at this location could be used to find the slope and intersect of the normal line, and then this line was plotted. The plotted results are seen in the attached video. Note that not every frame is always accurate at detecting the lines, but enough are that the overall shape and corners are usually seen.

Next, these 4 line's intersection points and 4 points corresponding to the corners of the paper, given the dimensions, in centimeters were used to find the homography. Again, the homography function from 2 was used, but this time without RANSAC.

Finally, this homography was combined with the given K matrix to find the Extrinsic matrix, E. This was done by multiplying the inverse of K with H: $E = K^{-1} * H$. I could then find lambda by averaging the normal of the first 2 columns of this matrix. Dividing E by this lambda would provide my transformation matrix, without the z column. To find the z column, I crossed the first and second columns, r1 and r2, respectively. I could then use the 3x3 matrix of rotation vectors to find the roll, pitch, and yaw. The third E column would provide me with the x, y, and z translations. These were displayed on the video frame with the Hough lines and saved out.

Figures:

Figure 1: The matrix of the accumulator, resized and displayed as an image for a given frame of the video

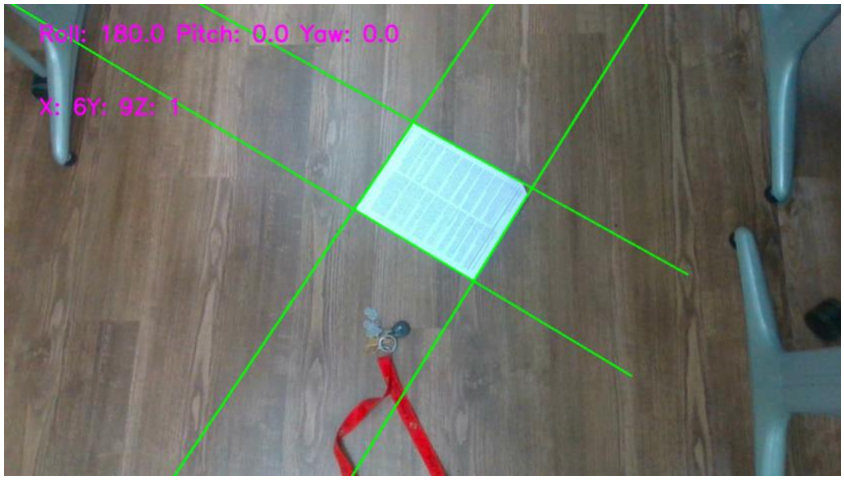

Figure 2: Results of Houghlines and camera pose estimations relative to the paper. R, P, and Y are in degrees, for a given frame

```
Homography:  [[-5.93094366e-05 -3.14255610e-01  1.18618873e+02]
 [-1.46033751e-01  2.43247666e-01  2.16524488e+00]
 [ 6.76027167e-03 -1.12620161e-02  1.00000000e+00]]
E_matrix, divided by lambda [[-1.00000000e+00 -0.00000000e+00 -1.13520000e+09]
 [-0.00000000e+00  4.10133159e+03 -1.15437014e+07]
 [ 0.00000000e+00 -0.00000000e+00  1.39606789e+07]]
```

Figure 3: Homography and Extrinsic matrix of a given video frame

Code: See attached files and README.md for execution information

Problems:

This problem was my most difficult. It took me a very long time, upwards of 7 hours, to get a working Hough transform. Tuning the canny edge parameters, iterating over the image properly, and plotting the resulting pixels was difficult. Ultimately, the most helpful solution for me was to stop trying to use matplotlib to plot the sinusoids and instead plot them into an image. A further problem was encountered when determining how best to draw the lines. For close to 4 hours my lines were drawing in all the wrong places. It took a joint solution to get correct edge and corner detection. First was figuring out that the d and theta could represent normal lines outside the image frame, and the resulting normal line would not display in frame. Second was revisiting my masking to optimally tune my canny edge detection. My initial solution was picking up ever so small pixels from the sides of the frame and that was causing my hough line plotter to pick up the wrong points as the most voted peak. As such, making a better mask was crucial to getting the desired results. One other issue was that I initially was finding maximums from a preset slice of the Hough accumulator. As such, each time I would rescale an image, or change a parameter, I had to re tune these regions. Finally, I had enough of this process and built a peak detector that looks near itself so that multiple local maxima can be detected in the plot without an especially bright region overpowering the maximums.

Problem 2:

This problem asked us to stitch 4 images together. To do this, three main steps were taken:

1. Find the features and match them between images. Feature extraction was done with the function $cv2.SIFT\_create(400)$. This would give a list of 400 keypoints and descriptors of those points per image. These keypoints then needed to be matched. To match them, the function $cv2.DescriptorMatcher\_create(cv2.DESCRIPTOR\_MATCHER\_FLANNBASED)$ was used. This would then match the descriptors between each image with $flann.knnMatch(d1, d2, 2)$. The results of this were compared to see if they were within a threshold of 0.7 * each other. If so, they were labeled a good match.
2. Find Homography. The good matches between 2 of the images were taken and 4 points were sampled randomly from each image. These points were then used to create a rotational homography for the images. This homography was then compared to each of the keypoints in each of the images. First, the source image point and the homography were dot-producted. Then, this resulting vector was divided by its last position and the normal was taken. The result of this was compared to a threshold value of 10, and if it was less than 10 it counted as an inlier, and the points were tracked. This comprised the RANSAC usage in this step as the iteration with the most inliers was used to calculate the Homography by passing in all of the inliers of the maximum iteration. In total, 200 iterations were performed per pair of frames.
   Next, to actually calculate the homography, I created a function to take in any number of points. To do this, the number of points was looped through and a row of the homography matrix was created with their x and y values. Then, the SVD of this was taken. The U, S, and V^H matrices were returned. I took the V^H matrix and constructed the last row divided by it's last value into a 3x3 matrix. This result was the homography matrix
3. Stitch images. Finally, using the homography matrix, I could call $cv2.warpPerspective(img, H, shape\ )$ to warp one image into the pose of the other. However, if I were to just warp each image to frame 1, they would become incredibly stretched. Not to mention the fact that very few features were found between image 4 and 1. As a result, I stitched images 2 and 3 together, and images 3 and 4 together. I had the most success stitching 1 and 2 together as they shared the most good features. As such, I stitched image 2-3 and image 3-4 together and then found their homography with respect to image 1. I them stitched these together. After warping, image stitching is as easy as replacing non-black pixels in the warped image with the base image. I warped everything with resect to image 1, as seen in figure 5 and 6.

Figures:

Figures 4: Features drawn onto image 3 and then stitched with image 4. Note how many features there are, which is why limiting sift to 400 was implemented

Figure 5: Images 2, 3, and 4 stitched together and warped with respect to 1, before the final stitch

Figure 6: Images 2, 3, and 4 stitched onto 1

Problems:

Problem 2 posed a couple of major problems for me. Firstly, I started with this problem so I created the homography here first. Learning how to find the matrix from any number of points was challenging. I estimate that part alone took upwards of 5 hours. Next, I ran into issues warping and stitching frames properly. After visiting online resources and repositories that included somewhat related stitching algorithms, I was able to stitch 1 and 2 together. However, 3 and 4 were only stretching and rotating to insane dimensions. My ultimate solution is listed above, where I construct intermediate frames and then stitch them all into one. I estimate fixing the stitching problems took me another 4 hours.

Overall Remarks:

This project was a rather long one. Between debugging, relearning linear algebra, and learning best ways to find homography and Hough lines, it took up a lot of time. Unfortunately, my problem 1 solution did not appear to have accurate r, p, or y values for rotation. As a result of my time spend developing the rest of this project, I was unable to address this issue. However, I am confident my calculations elsewhere are accurate. I hope to take these functions and use them for personal projects, such as 360 degree camera construction or line detection algorithms.