Jared Waters and Leo Chen
CE 264
Project 2 Report

Part 1: Camera Calibration

Cameras are not all uniform, they have varying degrees of distortion and intrinsic parameters. In part 1 we take many photos of a calibration image and use the OpenCV Camera Calibration application to get the K matrix and distortion coefficients. These parameters are then used to undistort out images so that they are flat; ie straight lines in the real world will be straight lines in the image. The K matrix is essential in completing the rest of the project.

Methods used and Results obtained while calibrating the camera:

Initially we calibrated the camera using the matlab camera calibrator. What we expected to see was reasonable values for the focal length and a principal point which was around the middle of the screen. This was the case, the K matrix given from the matlab camera calibrator worked looked good.

However, we had two problems with the matlab method. The first is that matlab can only provide either two or three distortion coefficients. The second is that no matter how many calibration images we added to the processing list, the reprojection error would never drop below 1. We attempted calibrating with as many as 80 images with no change. For these reasons we decided to switch to opencv 3.4.1 with python.

Getting the software configured turned into a several day ordeal. Eventually I resorted to installing Ubuntu on my PC and using opencv in the linux environment. This was a good choice, but I wish I had done it sooner. Many hours were wasted trying to get the windows environment working. The linux installation process for opencv was painless and straightforward, allowing us to finally get to work. I followed the guide by PyImageSearch for Ubuntu 18 and opencv 3.4.1.

For calibration I initially attempted to use the code examples from the opencv 3.1 documentation. The code ran, finding and displaying corners correctly, but the K matrix and distortion coefficients generated made no sense. The K matrix had focal lengths is the 55000 range and focal point at (300,800) for an image which was 2016x1512 pixels. This was obviously incorrect. In addition, the distortion coefficients were in the 10^3 range, when I expected coefficients in the 10^-2 or 10^-3 range. After attempting to debug the code, I found a more recent example in the documentation for opencv 3.4 (1). This worked immediately, producing results that were consistent with expectations and the physical system.

To take the calibration images, we printed out a picture of a chessboard pattern. We taped it onto the table in front of us and took many pictures with Leo's iPhone at a variety of poses. The images were then resized to reduce processing time. Care was taken to get images at many different distances and angles. We took many photos, but for the process in opencv only used 35 as this produced a good reprojection error of .11 pixels. The files are in the main project directory.

My adapted code can be found in the project directory as cameraCalib2.py. The program reads in all jpegs in the project directory. For each, it will attempt to find the chessboard corners. If it is successful, it sets a flag that allows the corner (x,y) points to be appended to an array of such points. The point found is only approximate. In order to refine it, it takes a 22x22 pixel box around the approximate corner point and computes the areas of greatest gradient. Where these gradients abruptly change direction with respect to x and y is the location of the precise corner point.

The program then takes this array of points and calculates the camera intrinsic matrix and distortion coefficients using calibrateCamera. I then use these parameters to undistort image1 and image2.

Results:

(Pictures of the chessboard pattern are provided in the 'ce264project2.zip' file)

Reprojection error is .11 pixels.

K= 1644, 0, 1027 *Rounded to integer for report. Double precision used for project*
0, 1638, 748
0, 0, 1

Distortion Coefficients: 0.2004, -1.196, -0.00393, 0.00001873, 2.013

Part 2: Taking the pictures

We took the pictures with Leo's iPhone in a study room. There was a lot of loose items on the table we took a picture of, providing adequate texture for deriving feature points. In addition, the wall behind the table served to ensure that all distances are finite in the system. His phone does not zoom, so that was not a concern.

Results: The images can be found in the main project directory. They are IMG_0849.JPG and IMG_0850.JPG.

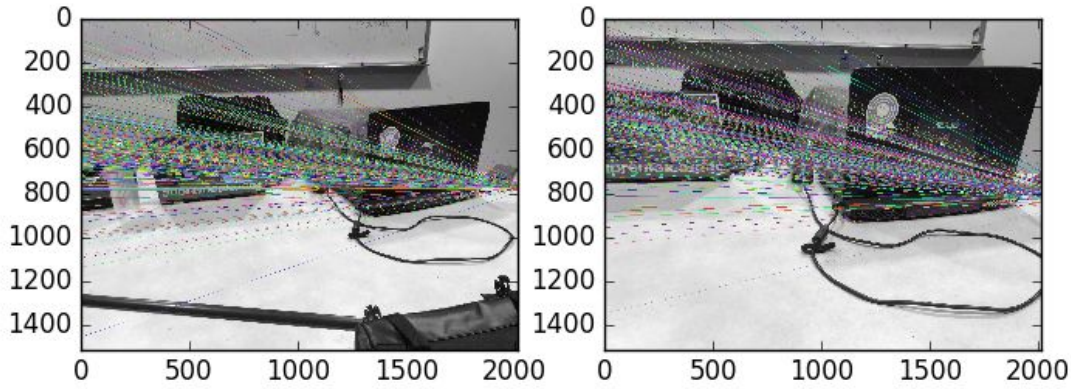
Part 3: Compute the relative camera pose R^R_L, r^R

Once you have the camera intrinsic matrix K and the undistorted images you can proceed to generate matching points and the fundamental matrix. The fundamental matrix can then be decomposed into rotation and translation components. Points can then be triangulated

to calculate their depth. The second image can be projected onto the first to show the accuracy of the design.

Methods used and Results obtained:

The next step is to compute the feature points by using the SIFT-feature detector. The SIFT-algorithm is patented which means that we had to download and install the 'opencv-contrib' folder. We ran into a few issues of mismatching versions which cost us time. SIFT will detect keypoints and project them on the image. Another function we used from the tutorial was the FlannBasedMatcher() and the flann.knnMatch() function. Through this we are able to run an algorithm that matches the keypoints from the first and the second image. By running a short for-loop we filtered out the bad matches that had a distance above the threshold. Given the keypoints, we are now able to compute the fundamental matrix with the findFundamentalMat() function from OpenCV. With help of the points and the computed fundamental matrix, we were able to determine the corresponding epilines. The lines are drawn in the drawLines() function given to us in the tutorial. The deliverable for the **epipolar lines** are provided in the figure below.



Epipolar Lines on Images

The next step is to find the essential matrix by using the keypoints of both images, the focal length of the camera (**f = 1644**) and the principal point (**pp = (1027, 748)**). To decompose the essential matrix we used the singular value decomposition (SVD) to find our rotation matrices R1, R2, and the translation vector r. Now we have four possible solutions (R1, r), (R1, -r), (R2, r) and (R2, -r). We have to find the right combination of R1, R2, r, and -r. This can be done by triangulating the points with the triangulatePoints() function. Since we are looking for the depth of each pair of (R, r), we have to print out the Z-Coordinate. The set of coordinates has to be positive since we are looking at the depth of the second image with respect to the first.

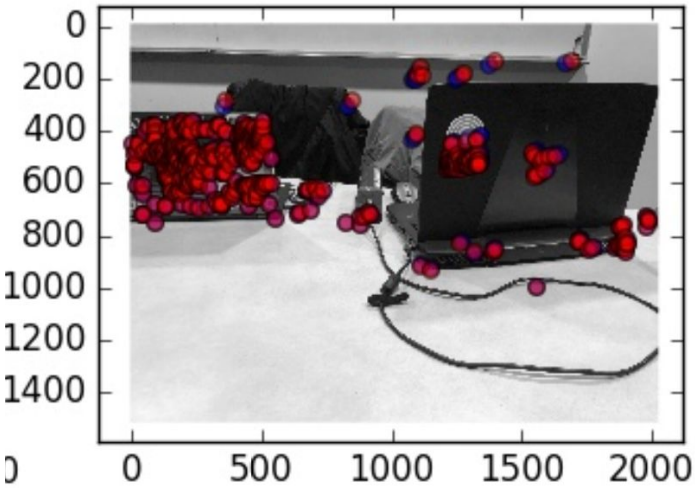
Negative values are useless because that would imply that the coordinates are behind the camera lens. The keypoints from both images as well as the new camera matrices

$$P1 = \begin{matrix} 1644, & 0, & 1027, & 0 \\ 0, & 0, & 1833, & 0 \\ 0, & 884, & 382, & 0 \\ 0, & 0, & 1, & 0 \end{matrix} \qquad P2 = \begin{matrix} 182, & -4.921, & 1929, & -1713 \\ 803, & -1579, & 318, & -662 \\ 0.891, & 0.0702, & 0.448, & -0.864 \end{matrix}$$

The matrices are defined as P1 = [K | 0] and P2 = [K □ [R1 | -r]]. Both matrices have a size of 3x4. Since P1 is our reference frame, we can just add zeros to the fourth column. The way to determine what to multiply K with for the P2-matrix, we looked for positive Z-coordinates as we discussed earlier. The **matrix $\mathbf{R^R_L}$** and the **translation vector $\mathbf{r^R}$** have the values

$$\mathbf{R^R_L} = \begin{matrix} \mathbf{-0.44583,} & \mathbf{-0.04686,} & \mathbf{0.89388} \\ \mathbf{0.08378,} & \mathbf{-0.99642,} & \mathbf{-0.01045} \\ \mathbf{0.89118,} & \mathbf{0.070235,} & \mathbf{0.44816} \end{matrix} \qquad \mathbf{r^R} = \begin{matrix} \mathbf{0.502124} \\ \mathbf{0.009395} \\ \mathbf{0.864744} \end{matrix}$$

The triangulation function returns a 4xN matrix, which we had to normalize to get (X, Y, Z)-coordinates. Now we had to look at all the Z-coordinates and eventually we found out that [R1 | -r], which is a 3x4 matrix contained all positive values. Since we had to project the triangulations points on a 2D image, we had to again normalize by dividing the first two rows (X, Y) with Z. Now we have a 2D set of triangulation points which we then projected on the first image as you can see in the image below. The blue points are the triangulation points and the red points are the keypoints from the beginning. If you look closely you can see that most of the triangulation points overlap with the keypoints.



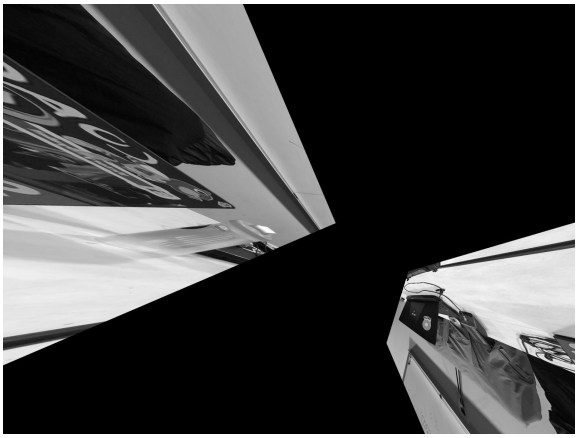
Reprojected feature points on the first image

Part 4: Plane-sweeping stereo

The last part of the project was to choose a set of planes in the reference frame of the first camera by taking 20 equispaced distances of planes that are orthogonal to the first images optical axis. We determined the equispaced distance by finding the min and max values of the Z-coordinates that were calculated from the triangulation in part 3. The min and max values are $\mathbf{d_{min} = 0.38714}$, $\mathbf{d_{max} = 0.7907}$. To have the equispaced distance we simply divide the difference of the min and max values by N = 20. To find the homography, we need to declare the camera matrices P1 and P2 by adding another fourth row of **n = [0, 0, -1, d]**. Then we multiply P1 with the inverse of P2 to get H = P1 * P2^(-1). There should be a set of 20 homography matrices.

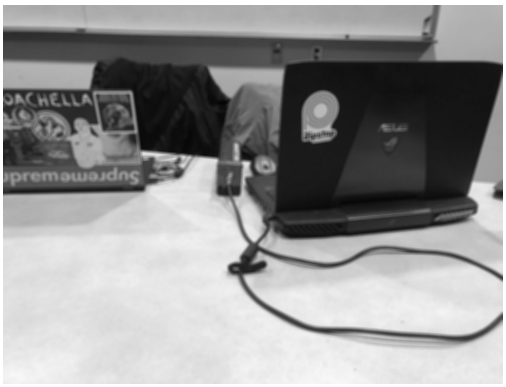
From this part one we started to have our issues with the warping. We did not have the right code to warp the images but what we wanted to get were 20 warped images. Each homography would map pixels on the plane at that depth onto the original image, meaning that they would have very low difference between the projected image and the original at the correct depth. We would first create 20 warped images. Each would then be subtracted from the original, creating a difference matrix. This difference matrix would be smoothed through block filtering to reduce noise. Those should then be subtracted from the reference image and then we wanted to compute the absolute difference of it. Part of a pseudo-code is provided in the project2.py file. For each pixel in the original, we could find the warped image which most closely resembled it in terms of brightness. The closest warp would be from the homography with a depth close to the depth at that pixel. In this way a 2 dimensional array of depths could be generated.

With the absolute value we now would have depth values for each pixel of the second image with respect to the first. The abs-diff-image in the code contains the depth values. To see the depth values correctly, we simply had to compare the 20 warped images and find the lowest absolute difference. Then we would blur the values so that the areas of depth would be more visible. This step is described as the block filter in the lab manual. A failed attempt of the warping is provided in the images below.



Fail attempt of warping images

This error most likely comes from the homography that had certain bugs we did not account for. Once we would find the depth from the absolute difference of the warped images and the reference image, we would assign the **white pixels to d_{max} and the black pixels to d_{min}.**



Warping did not work - we still used the block filter function

Conclusion:

In this lab we learned a lot about opencv and it's image processing libraries. We also learned the advantages of programming in a linux environment, which made setting up dependencies and installing all the required libraries easy.

This lab allowed us to explore the theory behind image transformations, feature detection, distortion, and intrinsic and extrinsic parameters in a practical setting