Homework 8

Theory Question

1. This idea is fundamental because it is the principle from which we are able to estimate the intrinsic parameters. Since we can get the homographies per camera pose using the chessboard pattern, we can also estimate the parameters of the image of the absolute conic, $\boldsymbol{\omega}$, as shown in Eq. 3, and that allows us to build a linear system of equations which we can solve to get these each of the values of the matrix $\boldsymbol{\omega}$.

2. This is algebraically proven by the fact that $\boldsymbol{\omega} = \mathbf{H}^\top \mathbf{I}_{3\times3}\mathbf{H}^{-1}$. Expanding this expression:

$$\begin{aligned}
\boldsymbol{\omega} &= \mathbf{H}^\top \mathbf{I}_{3\times3}\mathbf{H}^{-1} \\
&= (\mathbf{KR})^{-\top}\mathbf{I}_{3\times3}(\mathbf{KR})^{-1} \\
&= ((\mathbf{KR})^\top)^{-1}(\mathbf{KR})^{-1} \\
&= (\mathbf{R}^\top\mathbf{K}^\top)^{-1}(\mathbf{R}^{-1}\mathbf{K}^{-1}) \\
&= \mathbf{K}^{-\top}\mathbf{R}^{-\top}(\mathbf{R}^{-1}\mathbf{K}^{-1}) \\
&= \mathbf{K}^{-\top}(\mathbf{R}\mathbf{R}^\top)^{-1}\mathbf{K}^{-1} \\
&= \mathbf{K}^{-\top}\mathbf{K}^{-1}
\end{aligned} \tag{1}$$

   Since we know that $\mathbf{K}$ is a triangular matrix, so will its transpose. Additionally, any triangular matrix multiplied by its transpose will always result in a positive-definite matrix. So this proves that the image of the absolute conic will not contain any real pixel locations, all of them are imaginary.

Task: Zhang's algorithm

First of all, we define the 3D points for the pattern corners. For the first dataset we define the length of the black squares as 1, and the grid shape is 8×10. The order of the points is left to right, top to bottom.

Corner detection

Each image is opened as grayscale, which is then passed through Canny algorithm in order to find the edges of the black squares. The edges are then passed through a Hough transform in order to find the lines. The lines are then sorted depending on the parameters θ and ρ to check if they are either a horizontal line or a vertical line. For horizontal lines we have that $\theta \leq \pi/4$ and $\rho \geq 0$ or $\theta \geq 3\pi/4$ and $\rho \leq 0$, and if they are not horizontal then they are vertical. We then sort horizontal and vertical lines depending on their intersection to the y and x axis (the very top and the far left of the image) respectively. We find lines that have intersections that are within an specified distance of each other, we define this as the distance tolerance value, and we use 15 for the first dataset.
Now that the lines are all sorted in their respective groups, we focus on the groups of lines that have more than 1 line. To handle such cases, we get the intersection of each of the lines

in a group with the top and bottom of the image in the case of vertical lines, and with the left and right of the image in the case of horizontal lines. Then we find the midpoint between these intersections for the lines within a group. Finally, we get our final line, one that passes through the midpoints and keep that. This is all done with lines in their homogeneous coordinates representation.
After our filtering and grouping, we loop through the lines and get the x and y intercept, but this time we get the sorted arguments, so then we sort them with this. Now that the lines are sorted, we use a nested loop, with the inside looping through the vertical lines and the outside loop through the horizontal ones. Since these are in homogeneous coordinates, we can easily find the intersection between them, we store each intersection in an array. The index of each of these intersections in the array belong to the label of the corner.

Calculating Homographies

We use the corners of each of the images and we take the 3D points (without the Z values) and get the corresponding homography for each image with point to point correspondences. We then store each homography for later use.

Estimating intrinsic parameters

We have that the image of the absolute conic Ω_∞ is given by

$$\boldsymbol{\omega} = \mathbf{K}^\top \mathbf{K}^{-1} \tag{2}$$

Any plane in the world frame samples the absolute conic at exactly two points, and plugging these two points into the equation above gives us:

$$\begin{aligned} \mathbf{h}_1 \boldsymbol{\omega} \mathbf{h}_1 &= \mathbf{h}_2 \boldsymbol{\omega} \mathbf{h}_2 \\ \mathbf{h}_1 \boldsymbol{\omega} \mathbf{h}_2 &= 0 \end{aligned} \tag{3}$$

We know that the structure of the matrix $\boldsymbol{\omega}$ is:

$$\boldsymbol{\omega} = \begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{pmatrix}$$

But since it is symmetric, there are only 6 parameters we need to find. Expanding the equations in Eq. 3, we arrive at

$$\begin{aligned} \mathbf{v}_{12}^\top \mathbf{b} &= 0 \\ (\mathbf{v}_{11} - \mathbf{v}_{22})^\top \mathbf{b} &= 0 \end{aligned} \tag{4}$$

With $\mathbf{b} = \begin{pmatrix} \omega_{11} & \omega_{12} & \omega 22 & \omega_{13} & \omega_{23} & \omega_{33} \end{pmatrix}^\top$ Where $\mathbf{v}$ is given by:

$$\mathbf{v}_{ij} = \begin{pmatrix} h_{i1} h_{j1} \\ h_{i1} h_{j2} + h_{i2} h_{j1} \\ h_{i2} h_{j2} \\ h_{i2} h_{j3} + h_{i3} h_{j2} \\ h_{i1} h_{j3} + h_{i3} h_{i1} \\ h_{i3} h_{j3} \end{pmatrix}$$

Rewriting Eq. 4, we have:

$$\mathbf{V}\mathbf{b} = \mathbf{0} \tag{5}$$

Where $\mathbf{V} = \begin{pmatrix} \mathbf{v}_{12}^{\top} & (\mathbf{v}_{11} - \mathbf{v}_{22})^{\top} \end{pmatrix}^{\top}$ This is a 2×6 matrix of values that we can calculate. Each position of the camera showing the $Z = 0$ plane will give us 2 equations, since we have 6 unknowns we need 3 camera positions at least. We can stack together the all the equations from the camera positions (homographies) and solve using linear least squares.
After getting the elements of $\boldsymbol{\omega}$ we can get the intrinsic parameters. The matrix $\mathbf{K}$ has the following form:

$$\mathbf{K} = \begin{pmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{6}$$

Which arrives us at the relationship between these parameters and $\boldsymbol{\omega}$

$$\begin{aligned} y_0 &= \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2} \\ \lambda &= \omega_{33} - \frac{\omega_{13}^2 + y_0(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}} \\ \alpha_x &= \sqrt{\frac{\lambda}{\omega_{11}}} \\ \alpha_y &= \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}} \\ s &= -\frac{\omega_{12}\alpha_x^2\alpha_y}{\lambda} \\ x_0 &= \frac{s y_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda} \end{aligned} \tag{7}$$

So with this our intrinsic matrix is found.

Estimating extrinsic parameters

The relationship between the intrinsic parameters, the homographies and the extrinsic parameters is given by:

$$\mathbf{K}^{-1} \begin{pmatrix} \mathbf{h}_1 & \mathbf{h}_2 & \mathbf{h}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t} \end{pmatrix} \tag{8}$$

So now, we expand and get a number of equations

$$\begin{aligned} \mathbf{r}_1 &= \zeta \mathbf{K}^{-1}\mathbf{h}_1 \\ \mathbf{r}_2 &= \zeta \mathbf{K}^{-1}\mathbf{h}_2 \\ \mathbf{r}_3 &= \mathbf{r}_1 \times \mathbf{r}_2 \\ \mathbf{t} &= \zeta \mathbf{K}^{-1}\mathbf{h}_3 \end{aligned} \tag{9}$$

The variable ζ is a scale factor to enforce that all the columns of the $\mathbf{R}$ matrix are of unit magnitude, with a value of $\zeta = 1/\left\|\mathbf{K}^{-1}\mathbf{h}_1\right\|$.

Now $\mathbf{R}$ and $\mathbf{t}$ are known. However, there is a few more steps we must follow. The rotation matrix obtained is still not guaranteed to be orthonormal. For this, we decompose $\mathbf{R}$ with SVD, resulting in $\mathbf{R} = \mathbf{UDV}^\top$ and set $\mathbf{R} = \mathbf{UV}^\top$.
For our next step we need to have another representation of the rotation matrix, as its DoF is 3, but has 9 values. We want the number of values of this representation to be the same number as the DoF. For this we get the Rodrigues vector, $\mathbf{w}$, given by:

$$\begin{aligned} \mathbf{w} &= \frac{\phi}{2\sin\phi}\begin{pmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{pmatrix} \\ \mathbf{R} &= \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \end{aligned} \tag{10}$$

And to go from the Rodrigues vector back to the rotation matrix we follow:

$$\begin{aligned} \mathbf{R} &= \mathbf{I}_{3\times3} + \frac{\sin\phi}{\phi}[\mathbf{w}]_X + \frac{1-\cos\phi}{\phi^2}[\mathbf{w}]_X^2 \\ \phi &= \|\mathbf{w}\| \\ [\mathbf{w}]_X &= \begin{pmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{pmatrix} \end{aligned} \tag{11}$$

Parameter Refinement

Now that we have all our intrinsic and extrinsic parameters we need to refine these. We do so with Levenberg-Marquadt algorithm. For this we need to optimize the reprojection error, which will be our Euclidean distance. For this we separate all parameters and optimize for it. One thing that we consider is that the intrinsic parameters are shared between all images, so we implement our optimization to reflect that.

Plotting Camera Poses

After refining the parameters with non-linear least squares, we can now find the position of the camera, the camera plane and the direction of the camera's coordinate axis. For that we make use of the extrinsic parameters in the following manner

$$\begin{aligned} \mathbf{C} &= -\mathbf{R}^\top\mathbf{t} \\ \mathbf{X}_{cam} &= -\mathbf{RX} + \mathbf{t} \end{aligned} \tag{12}$$

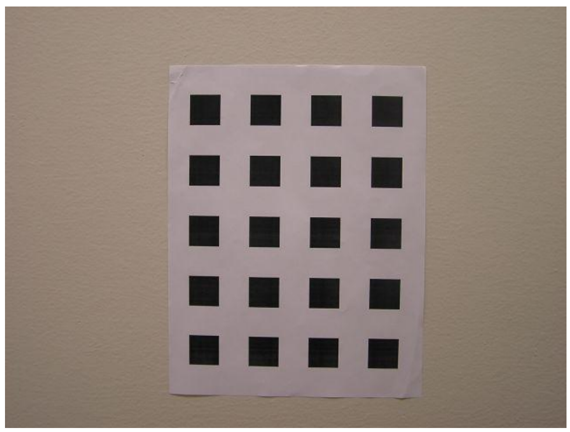
$$\begin{aligned} \mathbf{X} &= \mathbf{R}^{-1}\mathbf{X}_{cam} - \mathbf{R}^{-1}\mathbf{t} \\ &= \mathbf{R}^\top\mathbf{X}_{cam} - \mathbf{R}^\top\mathbf{t} = \mathbf{R}^\top\mathbf{X}_{cam} + \mathbf{C} \end{aligned} \tag{13}$$

$$\mathbf{X}^x_{cam} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \mathbf{X}^y_{cam} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \mathbf{X}^z_{cam} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{14}$$

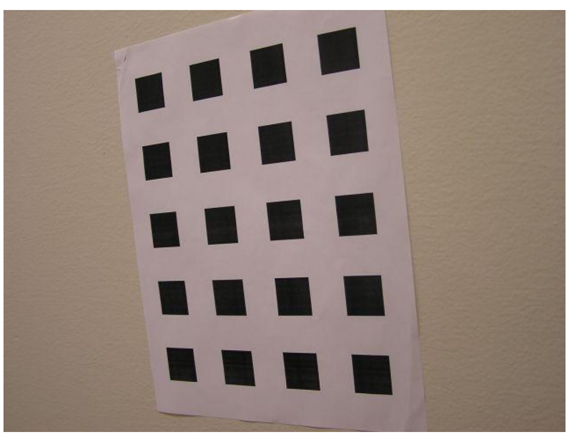
With this we now have the camera plane (the plane that contains $\mathbf{X}^x_{cam}$ and $\mathbf{X}^y_{cam}$. One modification we do for the purpose of plotting is that we don't consider adding $\mathbf{C}$ in Eq. 13, as the library we are using to make the 3D plot takes in a the origin of the vector and the direction, so there is no need here to add $\mathbf{C}$ back in.

Results: Dataset 1

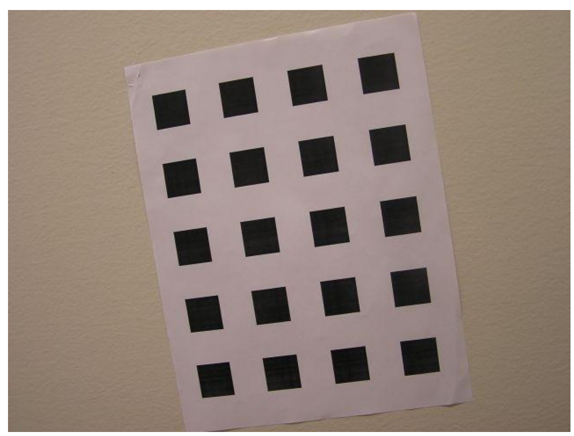
We will be using 'Pic_19.jpg' and 'Pic_31.jpg' to show the corners, the reprojection, the comparison between the refined parameters and the unrefined ones, and 'Pic_11.jpg' is our 'fixed image'.



(a) 'Pic_11.jpg', our fixed image



(b) 'Pic_19.jpg'.



(c) 'Pic_31.jpg'.

Figure 1: Example poses.

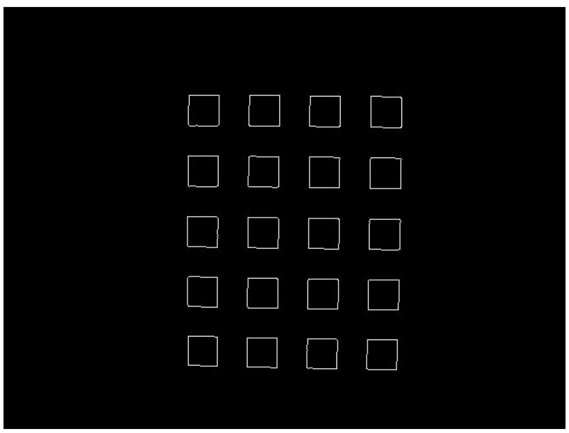
Corner detection

As mentioned before, the process of detecting corners is:

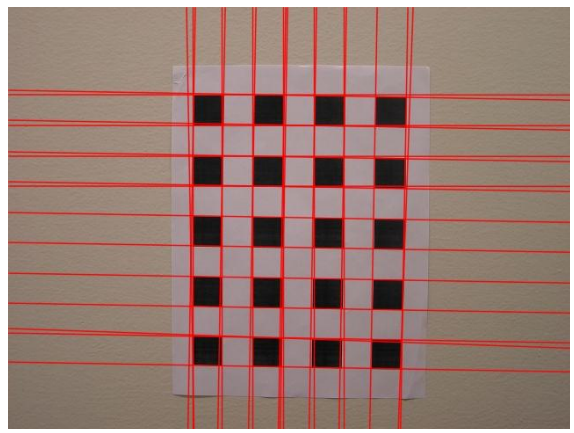
1. Use Canny algorithm on the grayscale image. We make use of OpenCV's Canny function, with thresholds of 300 and 350.

2. Use Hough transform on the Canny edges. Again, we use OpenCV's HoughLines function, with a parameter of 45.
3. We separate lines into vertical and horizontal, group those which are close and find the "midpoint" line for each group.
4. Now we sort all the vertical and horizontal lines depending on their x and y intercept.
5. Get the intersection of each vertical and horizontal line.

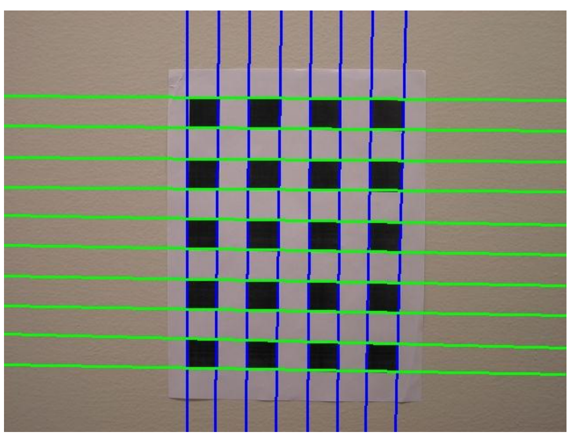
The process can be seen for the fixed image and our other 2 images in Figs. 2, 3 and 4.



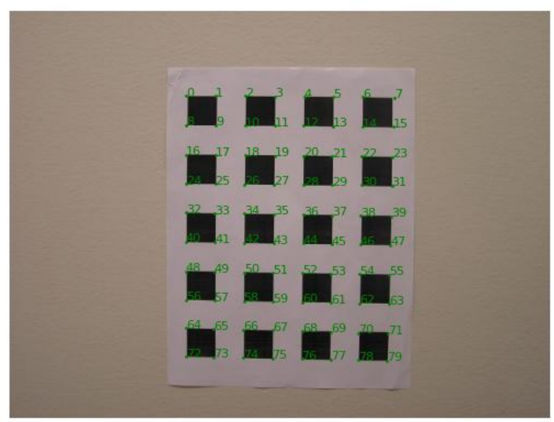
(a) Edges from Canny algorithm.



(b) Detected lines with Hough line transform.

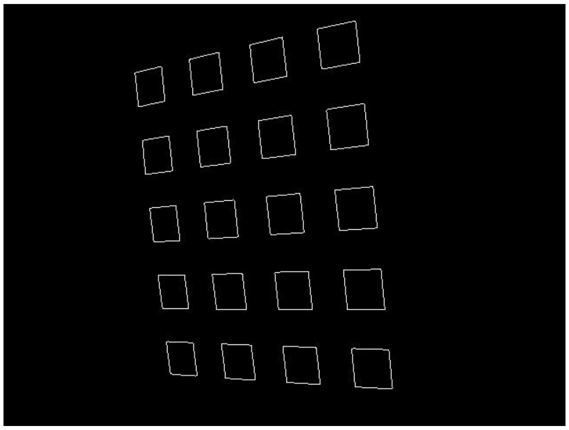


(c) Filtered and 'averaged' lines, vertical lines are shown in blue and horizontal are shown in green.

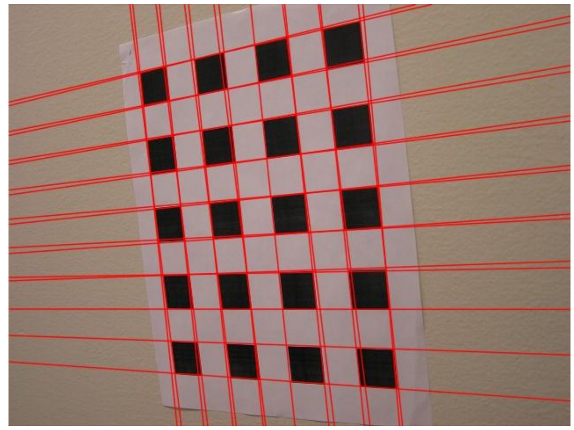


(d) Detected corners.

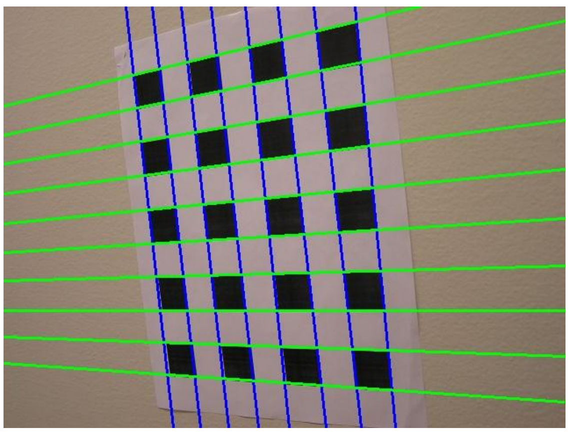
Figure 2: Corner detection process with 'Pic_11.jpg'.



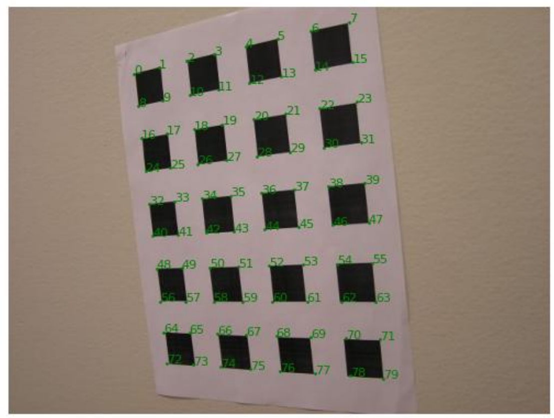
(a) Edges from Canny algorithm.



(b) Detected lines with Hough line transform.

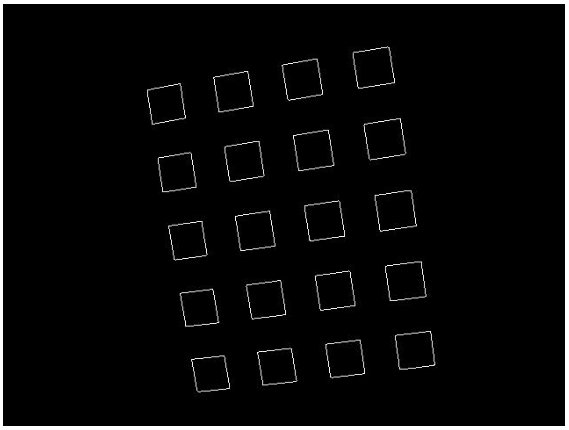


(c) Filtered and 'averaged' lines, vertical lines are shown in blue and horizontal are shown in green.

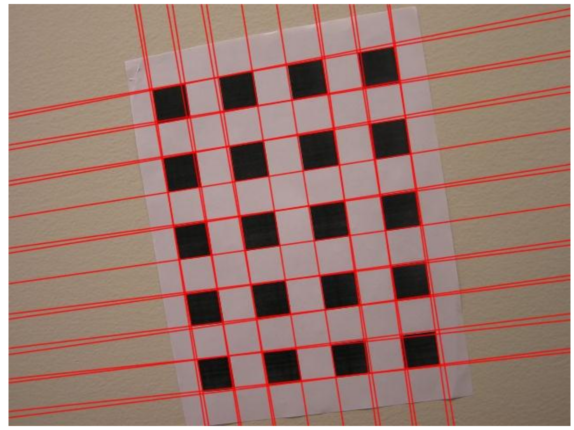


(d) Detected corners.

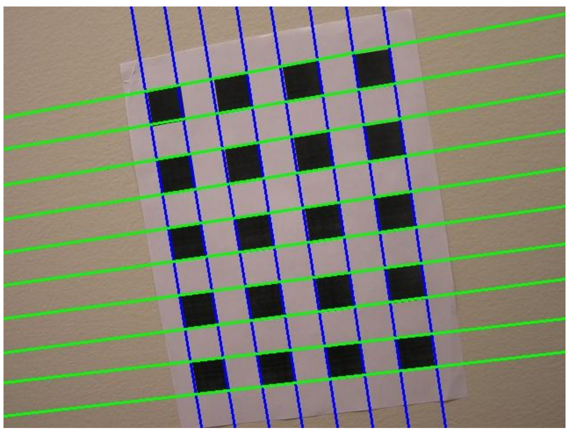
Figure 3: Corner detection process with 'Pic_19.jpg'.



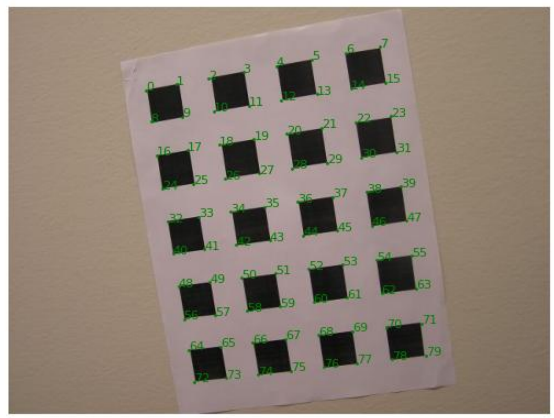
(a) Edges from Canny algorithm.



(b) Detected lines with Hough line transform.



(c) Filtered and 'averaged' lines, vertical lines are shown in blue and horizontal are shown in green.



(d) Detected corners.

Figure 4: Corner detection process with 'Pic_31.jpg'.

Intrinsic Parameters

First we found the homographies for each image through the use of point to point correspondences with the point grid without considering the z coordinates and store them in a list. We then loop through this list and get the equations shown in Eq. 4. We stack all the matrices $\mathbf{V}$ found per image and solve it using Linear Least Squares. So now we get $\mathbf{K}$ using the $\boldsymbol{\omega}$ found. Our initial values found for $\mathbf{K}$ are:

$$\mathbf{K} = \begin{pmatrix} 717.9290664 & 0.80913725 & 236.27489463 \\ 0 & 715.18947792 & 321.37701144 \\ 0 & 0 & 1 \end{pmatrix}$$

Extrinsic parameters

We use the intrinsic parameters to calculate the rotation matrix and the translation vector according to the equations described in the extrinsic parameter section, we also condition the rotation matrix and then append both the rotation matrix and the translation vector to an array for further use. Here are two examples, this is the extrinsics for 'Pic_19.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.85453588 & 0.08678829 & 0.51209006 & -2.38912397 \\ -0.04449309 & 0.99454832 & -0.09430805 & -6.58271785 \\ -0.51748314 & 0.05780514 & 0.8537387 & 19.28381236 \end{pmatrix}$$

And for 'Pic_31.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.9795844 & 0.15225277 & 0.13127638 & -1.93246888 \\ -0.14268342 & 0.98656977 & -0.07950799 & -5.77755365 \\ -0.14161862 & 0.05915382 & 0.98815231 & 18.37295395 \end{pmatrix}$$

Optimizing the parameters

To refine our parameters, we place the intrinsic parameters $\alpha_x, s, x_0, \alpha_y, y_0$ in an array. Then we loop through the lists containing the rotation matrices and the translation vectors. We obtain the Rodrigues vector from each of the rotation matrices, and append the 3 parameters of this vector to the parameter array, we do the same with the 3 parameters of the translation vector. After having all the parameters in an array, we can finally call Scipy's Levenberg-Marquadt algorithm. The function we will optimize is a cost function that we define to take in the the parameters, the 3D coordinates and the corners obtained from the corner detection process. This function will use the parameters to obtain the $\mathbf{K}$, $\mathbf{R}$ and $\mathbf{t}$ then use these to obtain the projection matrix given by $\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}]$ per image.
To visually check the improvement, we used 'Pic_11.jpg' as our 'Fixed Image'. We projected the corners for both image 'Pic_19.jpg' and 'Pic_31.jpg' to the 3D space (but only on the $Z = 0$ plane), then we took these points and projected them into the fixed image using its projection matrix (only the first, second and last column were used).
After our optimization our $\mathbf{K}$ matrix looks like:

$$\mathbf{K} = \begin{pmatrix} 811.00158749 & 5.56088429 & 263.17953178 \\ 0 & 813.5051964 & 229.06999829 \\ 0 & 0 & 1 \end{pmatrix}$$

And now, the optimized extrinsics for 'Pic_19.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.84324895 & 0.10543803 & 0.52708067 & -2.97020936 \\ -0.0745388 & 0.9940363 & -0.0795978 & -3.93242111 \\ -0.53232996 & 0.0278328 & 0.84607928 & 20.94152782 \end{pmatrix}$$

And for 'Pic_31.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.97823119 & 0.15597661 & 0.136876 & -2.58454503 \\ -0.14537129 & 0.98577176 & -0.08438737 & -3.36038005 \\ -0.14809095 & 0.06265252 & 0.9869872 & 20.52779866 \end{pmatrix}$$

We can see that indeed, LM optimization significantly reduced the error. In fact, it is so accurate that it is hard to tell where the reprojected corners are in some of the cases. This

Image	Error mean	Error mean (after LM)	Error std	Error std (after LM)
'Pic_19.jpg'	10.36677	1.086268	4.89901	0.496179
'Pic_31.jpg'	2.620455	0.870828	1.08557	0.4628802

Table 1: Euclidean error mean and standard deviation for the reprojection error into the fixed image.

is shown visually in Figs. 5 and 6, the reprojected corners onto the fixed image are vastly better after LM. Another metrics we can use to gauge this error is by using the mean and the standard deviation, and as we can see in Table 1, LM algorithm massively improved both, lowering the error by several times and the standard deviation as well. Also just projecting the 3D coordinates to the images shows improvement, as seen in Figs. 7 and 8.

(a) Fixed image with reprojected corners before LM.

(b) Fixed image with reprojected corners after LM.

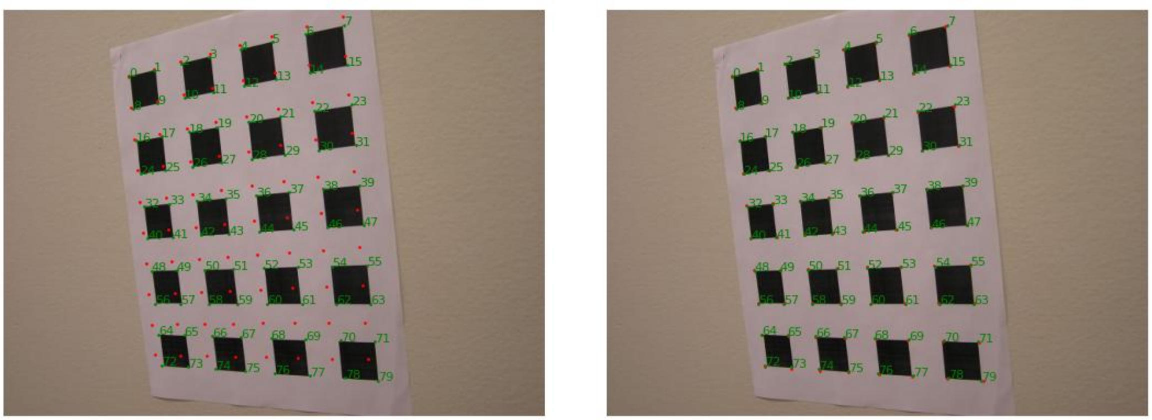
Figure 5: Visual reprojection from 'Pic_19.jpg' into the fixed image. The green dots are for the position of the detected corners and red is for the reprojected corners.

(a) Fixed image with reprojected corners before LM.

(b) Fixed image with reprojected corners after LM.

Figure 6: Visual reprojection from 'Pic_31.jpg' into the fixed image. The green dots are for the position of the detected corners and red is for the reprojected corners.

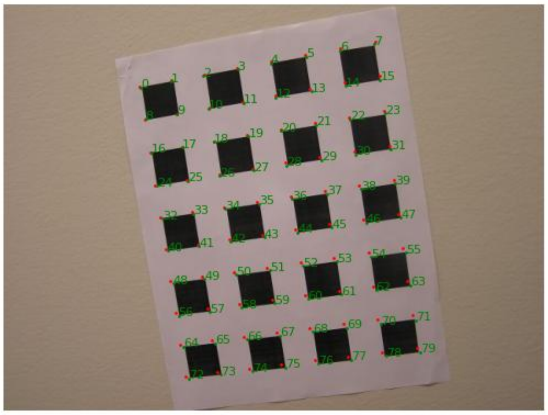
Now we will also try projecting the 3D points into the images using the projection matrix before and after LM optimization, and visually compare with the detected points.



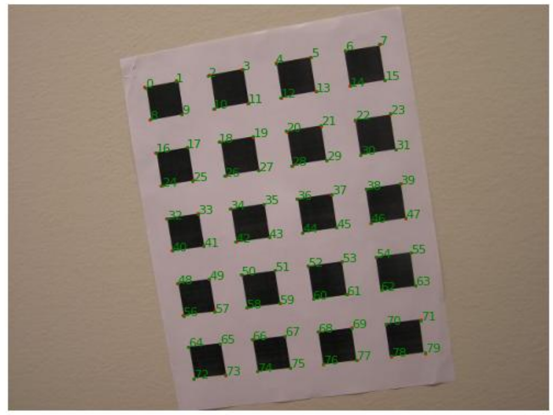
(a) Reprojected corners before LM.

(b) Reprojected corners after LM.

Figure 7: Visual reprojection of the 3D coordinates into 'Pic_19.jpg'. Green is the detected corners and red is the projected corners.



(a) Reprojected corners before LM.



(b) Reprojected corners after LM.

Figure 8: Visual reprojection of the 3D coordinates into 'Pic_31.jpg'. Green is the detected corners and red is the projected corners.

Camera poses

We now take our optimized $\mathbf{R}$ and $\mathbf{t}$ to get the camera poses. We follow the equations and obtain both the camera coordinates and the direction of the camera axis. To plot them we use Matplotlib and art3d from mpl_toolkits.mplot3d. We also make sure to plot the calibration pattern in this 3d plot using Rectangle from matplotlib.patches. To plot this pattern, we notice that it "skips" a line, there is one row with 4 black squares, then the next row is blank, and then 4 black squares again. So we make sure to have this reflected in the plotted calibration pattern. Notice that all the camera centers are in the negative side of the z axis, this is due to when we define the coordinates of the calibration pattern, we set the origin to be at the top left side, and increment the x coordinate when we go right. Similarly, we increase the y coordinate when we go to to the bottom of the image, so the z axis is actually pointing towards the image, thus all the camera poses have negative z values. We also make sure to plot the camera plane as a rectangle, to do this we know that this plane has 2 orthonormal vectors, in our case it is the x and y direction vectors of the camera, which we already have. So to plot this rectangle, we need each of the corners. Each of this corners, c_i will be given by:

$$\begin{aligned} c_1 &= \mathbf{C} + \mathbf{X}_x + \mathbf{X}_y \\ c_2 &= \mathbf{C} + \mathbf{X}_x - \mathbf{X}_y \\ c_3 &= \mathbf{C} - \mathbf{X}_x - \mathbf{X}_y \\ c_4 &= \mathbf{C} - \mathbf{X}_x + \mathbf{X}_y \end{aligned} \tag{15}$$

Where $\mathbf{X}_x$ and $\mathbf{X}_y$ are the directions of the camera x and y axis respectively. We also make sure to randomize the color of the camera plane for each different image.

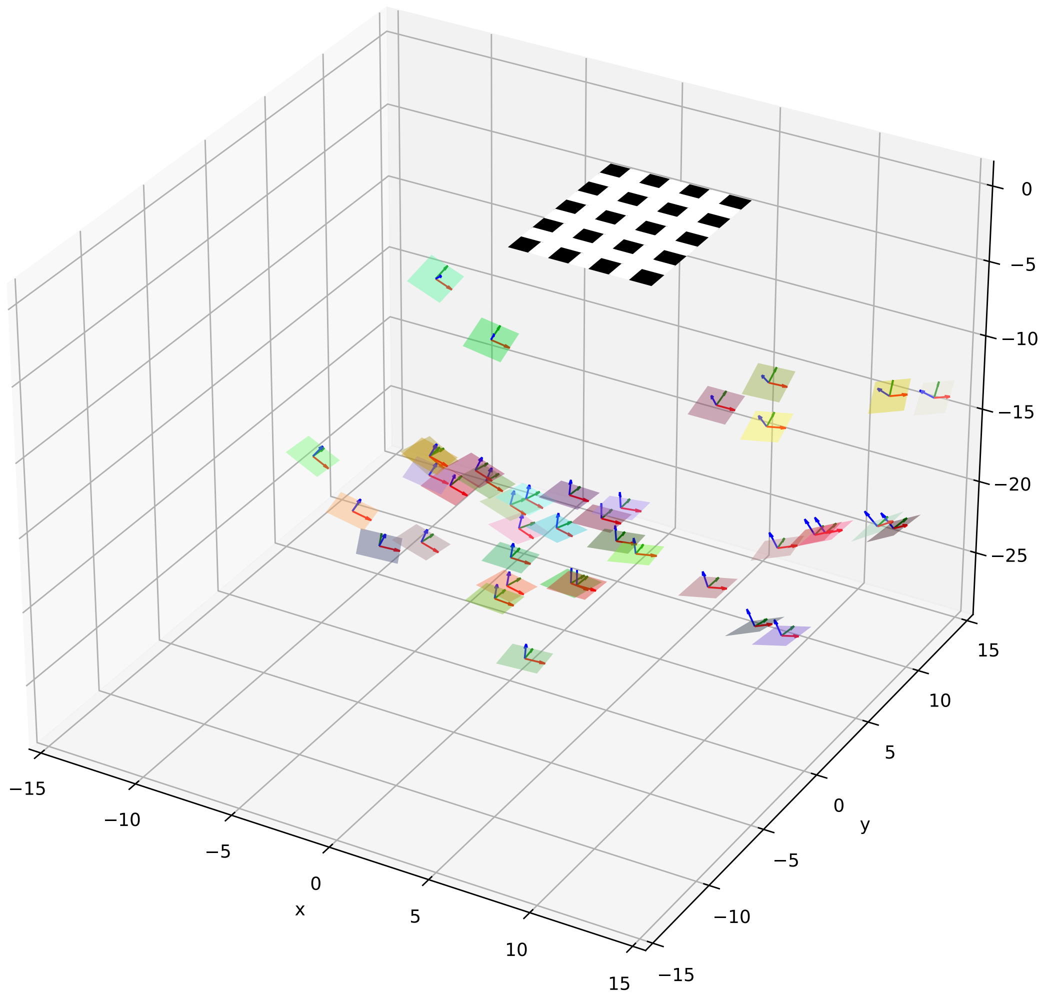
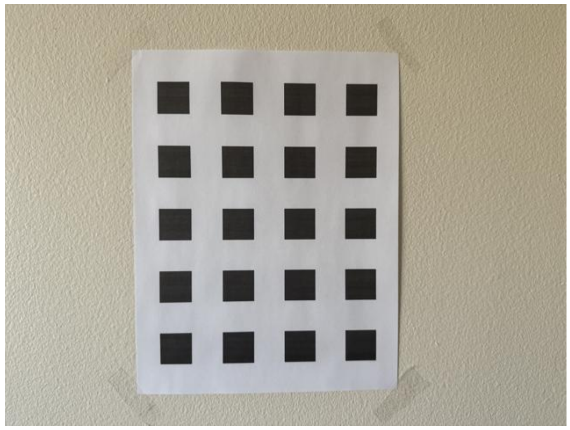


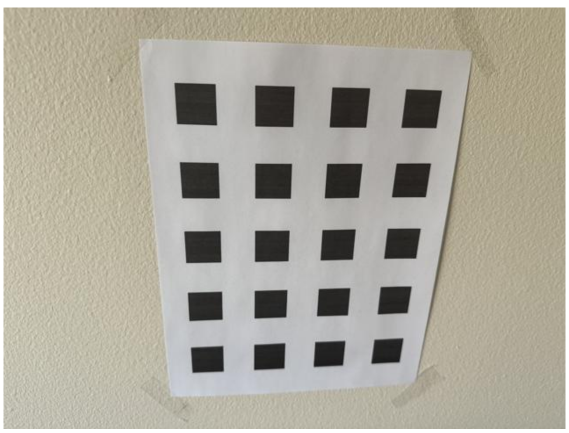
Figure 9: Camera poses for all the dataset 1 images.

Results: Own dataset

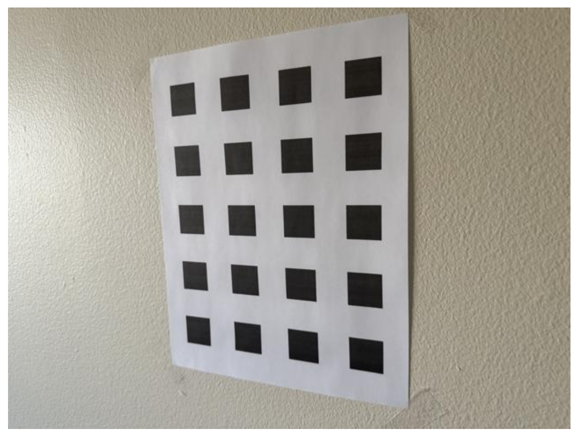
We largely follow the same procedure for the dataset we created. With the differences described in each subsection.



(a) 'fixed_img.jpg', our fixed image for our dataset



(b) 'pose9.jpg'.

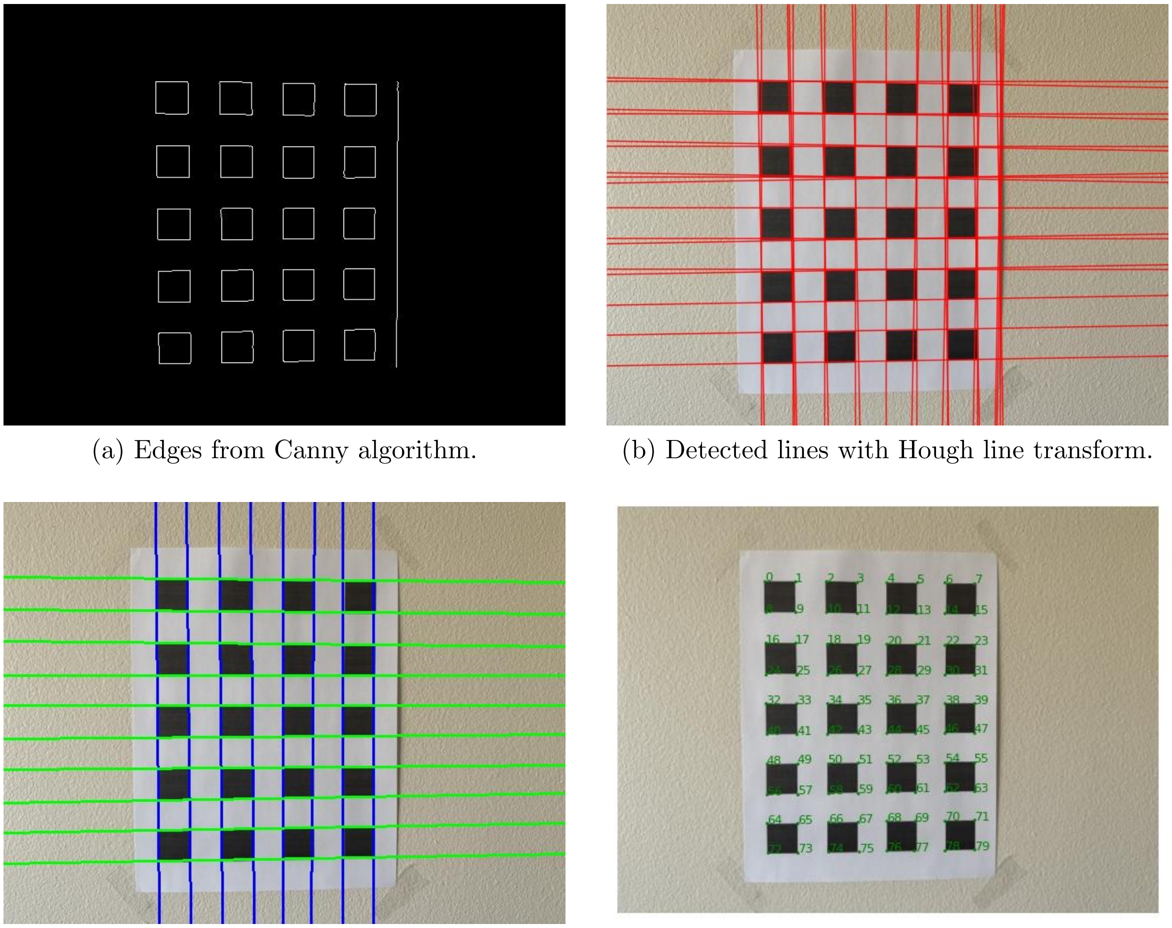


(c) 'pose33.jpg'.

Figure 10: Example poses for our own dataset.

Corner detection

We follow the same procedure, except due to the lighting conditions during the time we took the pictures, there is an additional step when filtering the lines. If the detected lines are more than the expected lines, we just take the first few, this can be seen in Figs. 11, 12, 13. The reason why we can do this is because the extra line are in the far right, there was also a case with a rogue line in the far bottom, but we clean it up in the same way. We use the same parameters for Canny and HoughLines as we did for dataset 1.



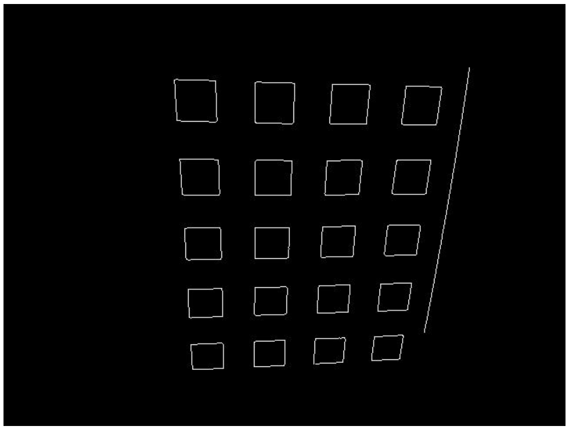
(a) Edges from Canny algorithm.

(b) Detected lines with Hough line transform.

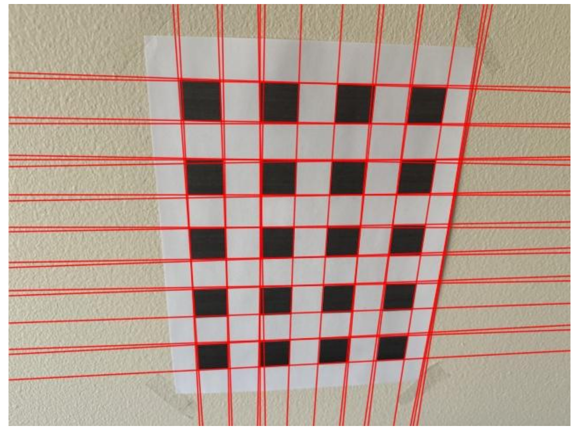
(c) Filtered and 'averaged' lines, vertical lines are shown in blue and horizontal are shown in green.

(d) Detected corners.

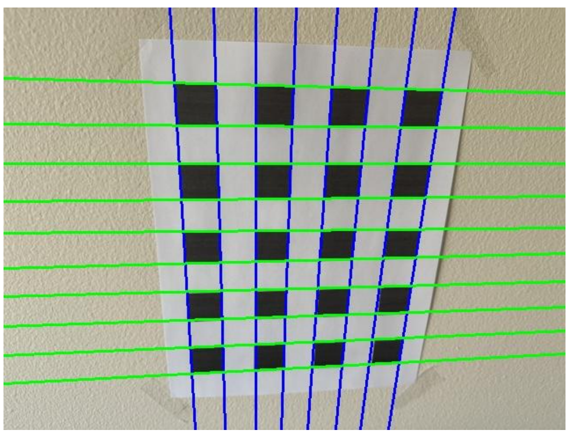
Figure 11: Corner detection process with 'fixed_img.jpg'.



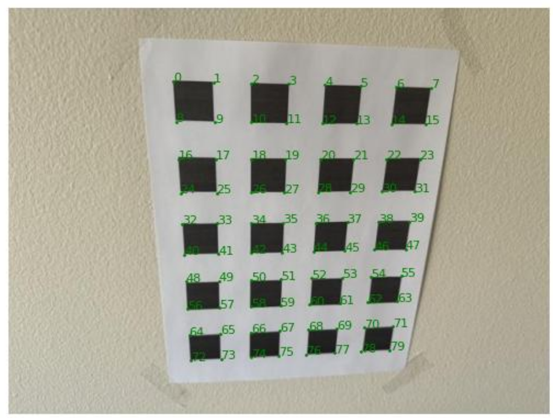
(a) Edges from Canny algorithm.



(b) Detected lines with Hough line transform.

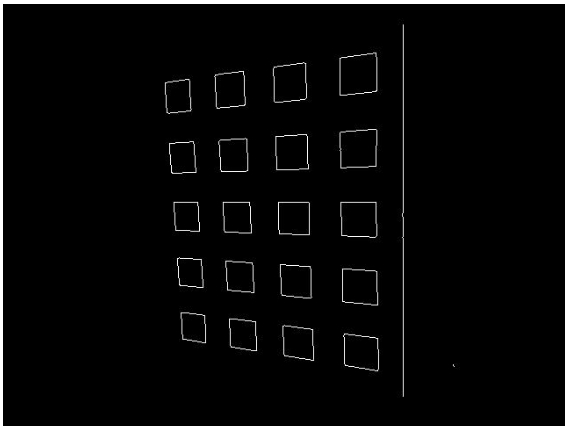


(c) Filtered and 'averaged' lines, vertical lines are shown in blue and horizontal are shown in green.

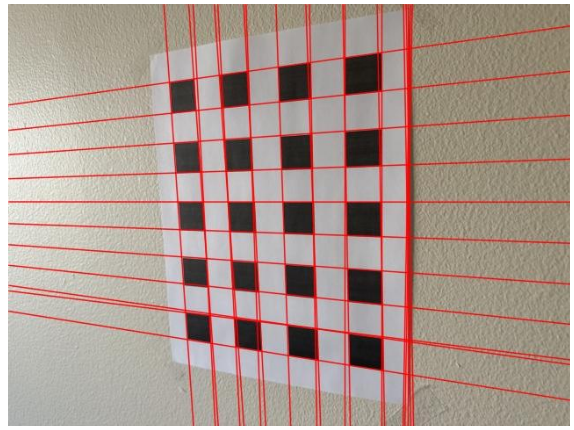


(d) Detected corners.

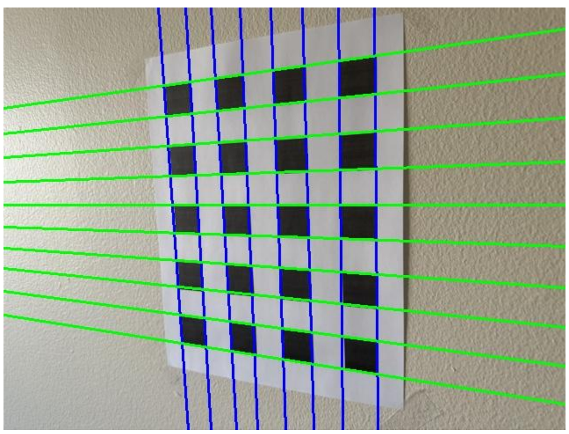
Figure 12: Corner detection process with 'pose9.jpg'.



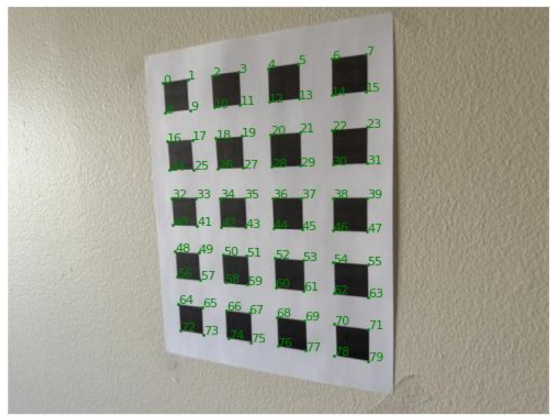
(a) Edges from Canny algorithm.



(b) Detected lines with Hough line transform.



(c) Filtered and 'averaged' lines, vertical lines are shown in blue and horizontal are shown in green.



(d) Detected corners.

Figure 13: Corner detection process with 'pose33.jpg'.

Intrinsic Parameters

The matrix **K** found is:

$$\mathbf{K} = \begin{pmatrix} 470.50104806 & -0.5421189 & 240.79357019 \\ 0 & 470.34740048 & 317.93433605 \\ 0 & 0 & 1 \end{pmatrix}$$

Extrinsic Parameters

We followed the same procedure as for dataset 1. This is the extrinsics matrix for our 'fixed_img.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.9983198 & 0.00993862 & -0.05708583 & -1.85545655 \\ -0.01205899 & 0.99924547 & -0.03691993 & -6.36269796 \\ 0.05667582 & 0.0375463 & 0.99768639 & 12.95199808 \end{pmatrix}$$

Since we are using a length of 1 inch for the black squares, the very last value of the translation vector means the pattern was 12.95 inches, which is close to the actual distance. We took this image approximately 13.2 inches away from the pattern. And for 'pose9.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.99403331 & 0.00932924 & -0.10867728 & -0.9852933 \\ -0.04960155 & 0.92602253 & -0.37419507 & -4.99084077 \\ 0.09714665 & 0.37735293 & 0.92095999 & 10.16441946 \end{pmatrix}$$

The extrinsics for 'pose33.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.89904183 & 0.02200944 & 0.43730923 & -1.7860764 \\ 0.04535019 & 0.98868418 & -0.14299286 & -7.15417657 \\ -0.43550791 & 0.14838862 & 0.8878703 & 14.6940454 \end{pmatrix}$$

Optimizing the parameters

Our optimized $\mathbf{K}$ is:

$$\mathbf{K} = \begin{pmatrix} 469.36153145 & -0.02628371 & 319.06039838 \\ 0 & 469.81047284 & 240.74777318 \\ 0 & 0 & 1 \end{pmatrix}$$

The extrinsics for 'fixed_img.jpg' are:

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.99829642 & 0.00181529 & -0.05831785 & -4.03596233 \\ -0.00386343 & 0.99937891 & -0.03502673 & -4.23588763 \\ 0.05821805 & 0.03519237 & 0.99768339 & 12.98397892 \end{pmatrix}$$

We can see that the last value of $\mathbf{t}$ is now closer to the measured value of 13.2. And for 'pose9.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.99403331 & 0.00932924 & -0.10867728 & -2.71841043 \\ -0.04960155 & 0.92602253 & -0.37419507 & -3.37535341 \\ 0.09714665 & 0.37735293 & 0.92095999 & 10.30325731 \end{pmatrix}$$

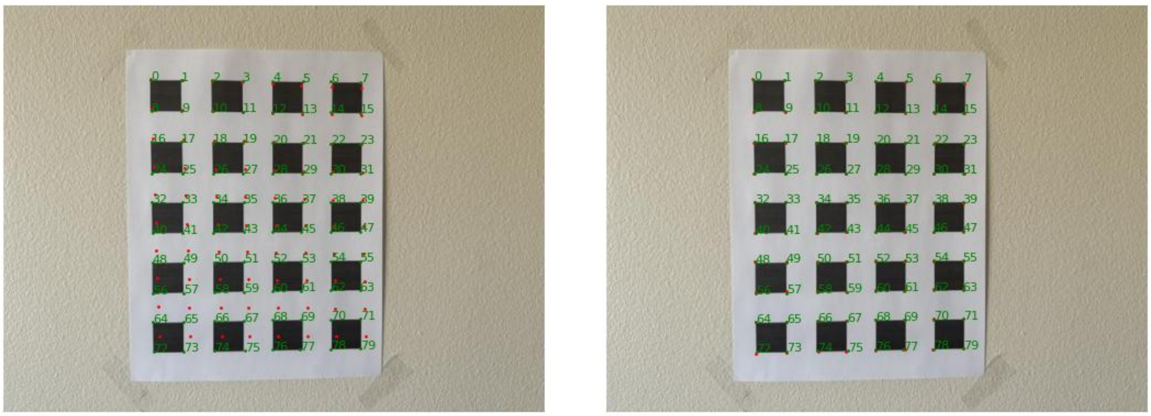
The extrinsics for 'pose33.jpg':

$$[\mathbf{R}|\mathbf{t}] = \begin{pmatrix} 0.9172927 & 0.03538065 & 0.39663876 & -3.99275192 \\ 0.01674614 & 0.9917367 & -0.12719232 & -4.49062442 \\ -0.39786136 & 0.12331476 & 0.90912035 & 13.85745156 \end{pmatrix}$$

Now, followed the same procedure of reprojecting corners into the fixed image, the results for the mean and the standard deviation are shown in Table 2.
We can visually see that the reprojection got significantly better after the optimization.

Image	Error mean	Error mean (after LM)	Error std	Error std (after LM)
'pose33.jpg'	13.92340	1.346285	6.142546	0.703136
'pose9.jpg'	9.113768	1.042369	6.007312	0.51973

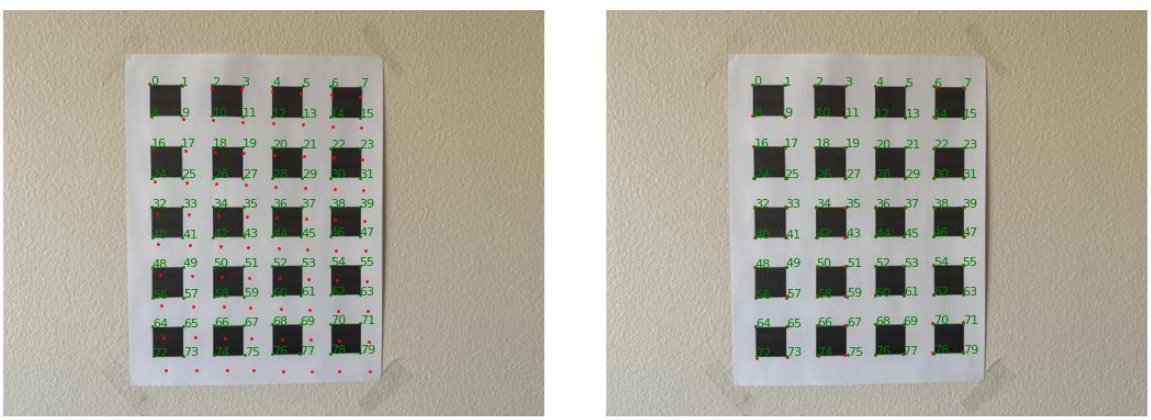
Table 2: Euclidean error mean and standard deviation for the reprojection error into the fixed image.



(a) Fixed image with reprojected corners before LM.

(b) Fixed image with reprojected corners after LM.

Figure 14: Visual reprojection from 'pose9.jpg' into the fixed image. The green dots are for the position of the detected corners and red is for the reprojected corners.



(a) Fixed image with reprojected corners before LM.

(b) Fixed image with reprojected corners after LM.

Figure 15: Visual reprojection from 'pose33.jpg' into the fixed image. The green dots are for the position of the detected corners and red is for the reprojected corners.

Now we will also try projecting the 3D points into the images using the projection matrix before and after LM optimization, and visually compare with the detected points.

(a) Reprojected corners before LM. (b) Reprojected corners after LM.

Figure 16: Visual reprojection of the 3D coordinates into 'pose9.jpg'. Green is the detected corners and red is the projected corners.

(a) Reprojected corners before LM. (b) Reprojected corners after LM.

Figure 17: Visual reprojection of the 3D coordinates into 'pose33.jpg'. Green is the detected corners and red is the projected corners.

Camera poses

We followed the same procedure, and plotted the camera poses in the same way.

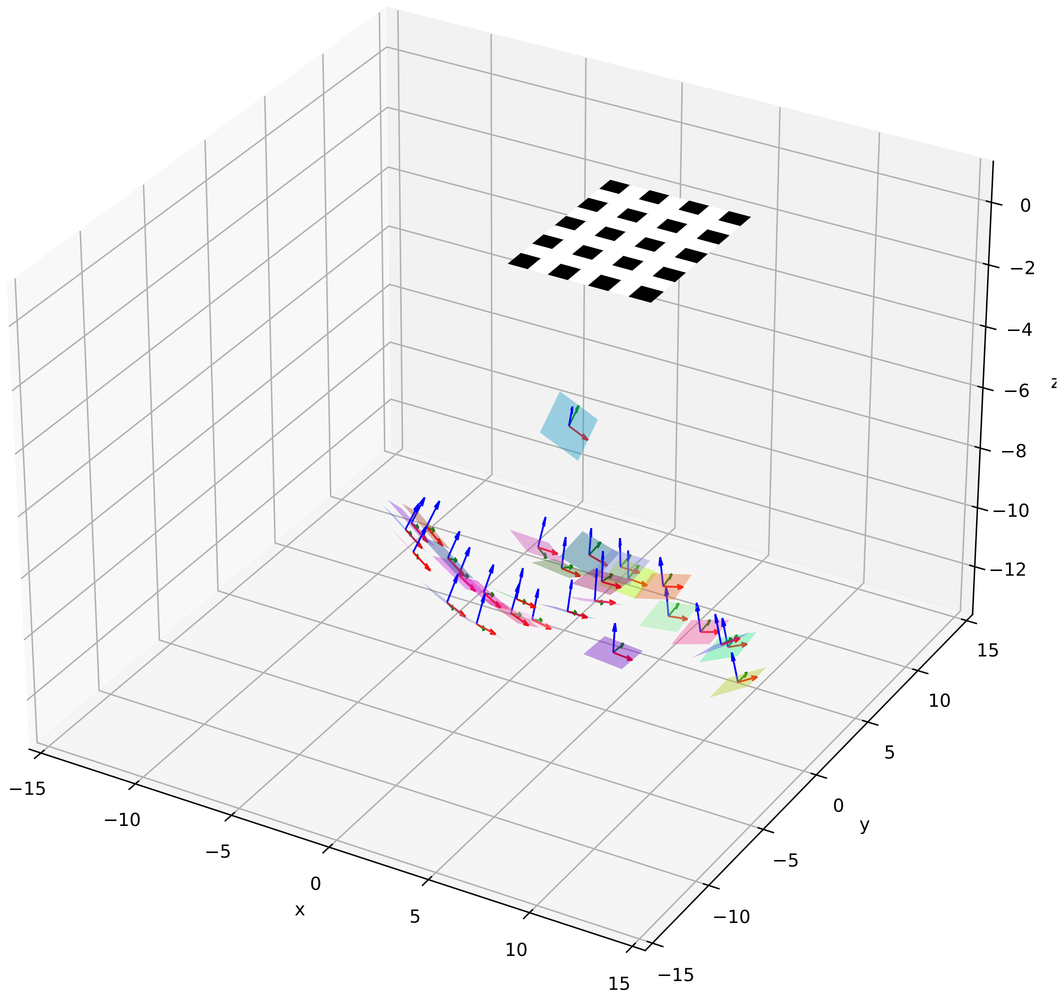


Figure 18: Camera poses for all the dataset 2 images.

Source code

```

import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares

# we use these to plot the camera poses
from mpl_toolkits.mplot3d.art3d import *
from matplotlib.patches import Rectangle

def get_point_or_line(point1,point2):
```

```python
    # first we check if these are already in homogeneous form or not, if
    not we put them in homogeneous representation
    if len(point1) == 2:
        point1 = np.array(point1)
        point1 = np.append(point1, 1)
    if len(point2) == 2:
        point2 = np.array(point2)
        point2 = np.append(point2, 1)
    # then get the cross product
    # this function has uses for us as both getting the line
    representation and
    # getting line intersections, as the math is the same
    r = np.cross(point1,point2)
    if r[2] != 0:
        return r/r[2]
    return r

def plot_hough_lines(img, lines, name):
    # this is just a function we use to plot the results of the hough line
     detector, prior to filtering and averaging
    img_ = img.copy()
    if lines is not None:
        for i in range(0, len(lines)):
            rho = lines[i][0][0]
            theta = lines[i][0][1]
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a * rho
            y0 = b * rho
            pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
            pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))
            img_ = cv2.line(img_, pt1, pt2, (0,0,255), 1, cv2.LINE_AA)
    cv2.imwrite(name, img_)

def plot_line_rep(line, image, horizontal=False):
    # this so we get the endpoints for the lines to plot after separating
    into vertical and horizontal
    # notice how we calculate different endpoints depending on if its
    vertical or not
    # this is because if its horizontal we want the intercepts with the
    left and rightmost part of the image,
    # and if its vertical we want the same but with top and bottom
    if horizontal:
        ref_1 = get_point_or_line([0,10],[0,15])
        ref_2 = get_point_or_line([image.shape[1]-1,10],[image.shape
    [1]-1,15])
    else:
        ref_1 = get_point_or_line([10,0],[15,0])
        ref_2 = get_point_or_line([10,image.shape[0]-1],[15,image.shape
    [0]-1])
    point_1 = get_point_or_line(line, ref_1)[:2]
    point_2 = get_point_or_line(line, ref_2)[:2]
    return point_1, point_2

```

```python
def cvl_to_homogeneous(cv2_line):
    # we use this to convert the line from rho theta to homogeneous coordinates
    # we operate most of the time with homogeneous as it is easier to use
    rho = cv2_line[0][0]
    theta = cv2_line[0][1]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a * rho
    y0 = b * rho
    pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
    pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))

    return get_point_or_line(pt1, pt2)

def separate_lines(lines):
    # we separate lines by horizontal lines (theta < pi/4 and rho positive, or theta>3pi/4 and negative rho),
    # and if they are not horizontal then they are vertical
    horizontal = []
    vertical = []
    for line in lines:
        rho = line[0][0]
        theta = line[0][1]
        if (theta <= np.pi/4 and rho >= 0) or (theta > 3*np.pi/4 and rho < 0):
            vertical.append(line)
        else:
            horizontal.append(line)
    return vertical, horizontal

def group_lines(lines, tol=30, image_shape=(600,600), horizontal=False, grid_shape=(8,10)):
    # in here we group lines together with depending on the spot they intersect the edges of the image
    # for horizonal lines we check where it hits the left edge of the image (y=0)
    # for vertical lines we check where it hits the top of the image (x=0)
    # we group them by checking which intersects are within a distance threshold of another interect
    # after we have grouped them together, we
    if horizontal:
        ref_line = get_point_or_line([0,10],[0,15])
        ref_line_n = get_point_or_line([image_shape[1],10],[image_shape[1],15])

    else:
        ref_line = get_point_or_line([10,0],[15,0])
        ref_line_n = get_point_or_line([10,image_shape[0]],[15,image_shape[0]])
    tot = []
    added = [False] * len(lines)
    for idx,iline in enumerate(lines):
        aux = []
```

```
            iline_rep = cvl_to_homogeneous(iline)
            ref_inter = get_point_or_line(iline_rep, ref_line)
            if not added[idx]:
                aux.append(iline_rep)
                added[idx] = True
                for jdx,jline in enumerate(lines):
                    if (jline != iline).any():
                        jline_rep = cvl_to_homogeneous(jline)
                        p_inter = get_point_or_line(jline_rep, ref_line)
                        if horizontal:
                            dist = np.abs(ref_inter[1] - p_inter[1])
                        else:
                            dist = np.abs(ref_inter[0] - p_inter[0])
                        if dist < tol and added[jdx] == False:
                            aux.append(jline_rep)
                            added[jdx] = True
                tot.append(aux)
    # now that we have them separated by groups, we "average" those which have 2 or more lines
    a_lines = []
    for lines in tot:
        if len(lines) > 1:
            inter_point1 = np.array([0,0],dtype=float)
            inter_point2 = np.array([0,0],dtype=float)
            count = 0
            for line in lines:
                inter_point1 += get_point_or_line(ref_line,line)[:2]
                inter_point2 += get_point_or_line(ref_line_n,line)[:2]
                count += 1
            inter_point1 = inter_point1/count
            inter_point2 = inter_point2/count
            n_line = get_point_or_line(inter_point1, inter_point2)
        else:
            n_line = lines[0]
        a_lines.append(n_line)


    # now that we have grouped and averaged the lines, we will order them with their x or y intercept depending on the orientation
    n_intercept = []
    for line in a_lines:
        intercept = get_point_or_line(line, ref_line)
        if horizontal:
            n_intercept.append(intercept[1])
        else:
            n_intercept.append(intercept[0])
    # we sort them with their intercepts
    n_intercept = np.array(n_intercept)
    inter_idx_sorted = np.argsort(n_intercept)
    t_lines = []
    for idx in inter_idx_sorted:
        t_lines.append(a_lines[idx])

```

```
    # this is the very last step, it will only filter the excess ones, this is mainly used for our dataset 2
    if horizontal:
        if len(t_lines) > grid_shape[1]:
            t_lines = t_lines[:grid_shape[1]]
    else:
        if len(t_lines) > grid_shape[0]:
            t_lines = t_lines[:grid_shape[0]]
    return t_lines

def get_corners(image, tol=15, save_canny=None, save_lines=None, save_hough=None, canny_p1=370, canny_p2=300, hough_p=50):
    # this function runs canny, houghlines, groups the lines, plots all of this,
    # also will loop through all the lines and get the intersects, those are our corners.
    img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    edges = cv2.Canny(img,canny_p1,canny_p2)
    lines = cv2.HoughLines(edges, 1, np.pi / 180, hough_p, None, 0, 0)
    if save_hough != None:
        plot_hough_lines(image, lines, save_hough[:-4]+"_houghlines.jpg")
    if save_canny != None:
        cv2.imwrite(save_canny[:-4]+"_canny.jpg", edges)

    vert_lines, hor_lines = separate_lines(lines)
    v_lines = group_lines(vert_lines,tol=tol, image_shape=(image.shape[1], image.shape[0]), horizontal=False)
    h_lines = group_lines(hor_lines,tol=tol, image_shape=(image.shape[1], image.shape[0]), horizontal=True)
    if save_lines != None:
        line_plot = image.copy()
        for vline in v_lines:
            p1, p2 = plot_line_rep(vline, line_plot, horizontal=False)
            p1_ = (int(p1[0]), int(p1[1]))
            p2_ = (int(p2[0]), int(p2[1]))
            line_plot = cv2.line(line_plot,p1_, p2_, (255,0,0), 2);
        for hline in h_lines:
            p1, p2 = plot_line_rep(hline, line_plot, horizontal=True)
            p1_ = (int(p1[0]), int(p1[1]))
            p2_ = (int(p2[0]), int(p2[1]))
            line_plot = cv2.line(line_plot,p1_, p2_, (0,255,0), 2);
        cv2.imwrite(save_lines[:-4]+"_fillines.jpg", line_plot)

    corners = []
    #this is how we get the corners
    for h_line in h_lines:
        for v_line in v_lines:
            corner = get_point_or_line(h_line, v_line)[:2]
            corners.append(corner)
    return corners

# reusing these from my hw5
def point_to_point_system(domain, range_,num_points):
    # we changed this function to accept an arbitrary number of points,
```

```
    # this is because most of the systems we will solve have more than 4 data points
    mat1 = np.empty((0,8),dtype=float)
    mat2 = np.empty((0,0),dtype=float)
    for i in range(num_points):
        x = domain[i,0]
        x_prime = range_[i,0]
        y = domain[i,1]
        y_prime = range_[i,1]

        mat1 = np.append(mat1,np.array([[x, y, 1, 0, 0, 0, -x*x_prime, -y*x_prime]]),axis=0)
        mat1 = np.append(mat1,np.array([[0, 0, 0, x, y, 1, -x*y_prime, -y*y_prime]]),axis=0)
        mat2 = np.append(mat2,x_prime)
        mat2 = np.append(mat2,y_prime)

    return mat1,mat2.reshape(num_points*2,1)

def linear_least_squares(mat1, mat2):
    # we make a distinction because this is for inhomogeneous least squares
    return np.linalg.inv(mat1.T @ mat1) @ mat1.T @ mat2

def linear_least_squares_homogeneous(mat1):
    # and this is for homogeneous least squares
    _, _, v_t = np.linalg.svd(mat1.T @ mat1)
    return v_t[-1]

def get_H_matrix(domain,range_):
    # I reuse this from hw5
    n_points = len(domain)
    mat1, mat2 = point_to_point_system(domain, range_,n_points)
    # we use this function to solve the equation described in the logic,
    # I replaced np.dot with @ as that is what they suggest in the numpy website
    # in this case we use the pseudo inverse (ATA)^-1 AT
    sol = linear_least_squares(mat1,mat2)
    # we append the 1 since this will only have 8 values, the 1 is missing
    sol = np.append(sol,np.array([[1]]),axis=0)
    return sol.reshape((3,3)).astype(float)

def add_ones(array):
    # this function takes in an array and adds ones
    # this is mainly for applying homographies
    ones_np = np.ones(array.shape[:-1])[...,None]
    # need to expand the dimensions by 1 to be able to concatenate it
    return np.append(array,ones_np,axis=-1)
def add_number(array,n):
    # this function takes in an array and adds any number
    # this is mainly for adding a 0, for z=0 in the real life plane coordinates
    ones_np = np.ones(array.shape[:-1])[...,None]
    ones_np *= n
```

```
    # need to expand the dimensions by 1 to be able to concatenate it
    return np.append(array,ones_np,axis=-1)

def apply_homography(positions, H):
    # reused from hw5
    # this gets the homography transformation for all the coordinates in
    the image that we get from the get_positions function
    # we do it in a way that exploits broadcasting, so we don't need to
    use for loops
    temp_pos = add_ones(positions)
    new_pos = (H @ temp_pos.T).astype(float)
    new_pos /= new_pos[2,:]
    return new_pos[:2,:].T

def get_coordinates(shape=(8,10),length=5):
    # this is how we create our 2d grid that is on Z=0,
    # for both datasets we end up using length =1 since the squares are of
    1 inch length
    x = np.linspace(0, length*(shape[0] - 1), shape[0])
    y = np.linspace(0, length*(shape[1] - 1), shape[1])
    xv, yv = np.meshgrid(x,y)
    xv = xv.ravel()[:, None]
    yv = yv.ravel()[:, None]
    return np.hstack((xv,yv))

def get_vij(H, i,j):
    # we change ij to go from 0 to 2 rather than 1 to 3, to adjust for
    python indexes
    # we follow the equations from the scrolls
    v_ij = np.array([H[0,i]*H[0,j],
                     H[0,i]*H[1,j]+H[0,j]*H[1,i],
                     H[1,i]*H[1,j],
                     H[0,i]*H[2,j]+H[0,j]*H[2,i],
                     H[1,i]*H[2,j]+H[1,j]*H[2,i],
                     H[2,i]*H[2,j]])
    return v_ij

def get_V(H):
    # this is to get the V matrix
    v_11 = get_vij(H,0,0)[..., None]
    v_12 = get_vij(H,0,1)[..., None]
    v_22 = get_vij(H,1,1)[..., None]
    V = np.vstack((v_12.T,(v_11 - v_22).T))
    return V

def get_b(V):
    # we get b from the stack of Vs
    b = linear_least_squares_homogeneous(V)
    return b

def get_omega(V):
    # this is the entire stack of Vs,
    # we get omega from b which we get from the stack of Vs
    b = get_b(V)
```

```
    omega = np.array([[b[0], b[1], b[3]],
                      [b[1], b[2], b[4]],
                      [b[3], b[4], b[5]]])
    return omega

def get_K_matrix(omega):
    # we follow the instructions from the scrolls and build our K

    x_0 = (omega[0,1]*omega[0,2] - omega[0,0]*omega[1,2])/(omega[0,0]*omega[1,1] - omega[0,1]**2)
    lambda_ = omega[2,2] - (omega[0,2]**2 + x_0*(omega[0,1]*omega[0,2] - omega[0,0]*omega[1,2]))/(omega[0,0])
    alpha_x = np.sqrt(lambda_/omega[0,0])
    alpha_y = np.sqrt(lambda_*omega[0,0]/(omega[0,0]*omega[1,1] - omega[0,1]**2))
    s = -1 * (omega[0,1] * alpha_x**2 * alpha_y)/(lambda_)
    y_0 = (s * x_0 / alpha_y) - (omega[0,2]*alpha_x**2)/(lambda_)
    K = np.array([[alpha_x, s, x_0],
                  [0, alpha_y, y_0],
                  [0, 0, 1]])
    return K

def get_extrinsics(K, H):
    # just following the instructions from the scrolls
    scale = 1 / np.linalg.norm(np.linalg.inv(K) @ H[:,0])
    r_1 = scale * np.linalg.inv(K) @ H[:,0]
    r_2 = scale * np.linalg.inv(K) @ H[:,1]
    r_3 = np.cross(r_1, r_2)
    t = scale * np.linalg.inv(K) @ H[:,2]
    R = np.column_stack((r_1,r_2,r_3))
    # now we condition R, very important step
    u, _, v = np.linalg.svd(R)
    R_conditioned = u @ v
    return R_conditioned, t

def get_rodrigues_rep_from_R(R):
    # we get the rodrigues vector from the rotation matrix, again following the scrolls
    phi = np.arccos((np.trace(R)-1)/2)
    w = np.array([R[2,1] - R[1,2],
                  R[0,2] - R[2,0],
                  R[1,0] - R[0,1]])

    return w * (phi/(2*np.sin(phi)))

def get_R_from_rodrigues(w_0, w_1, w_2):
    # this function is to help us reconstruct our matrices during LM optimization,
    # more on that in the following functions
    # again we follow the scrolls
    W_X = np.array([[0, -w_2, w_1],
                    [w_2, 0, -w_0],
                    [-w_1, w_0, 0]])
    phi = np.sqrt(w_0**2 + w_1**2 + w_2**2)
```

```
    R = np.eye(3) + (np.sin(phi)/phi) * W_X + ((1 - np.cos(phi))/(phi**2))
    * W_X @ W_X
    return R

def get_params_from_matrices(K, R_list, t_list):
    params = []
    # get the parameters needed from K, R, and t
    # R and t are per image,
    # since K is shared between all images, we make sure to optimize it in
    this way, that is why it is only appended once
    params.append(K[0,0])
    params.append(K[0,1])
    params.append(K[0,2])
    params.append(K[1,1])
    params.append(K[1,2])
    for idx in range(len(R_list)):
        R = R_list[idx]
        t = t_list[idx]
        #we need to use rodrigues instead of the rotation matrix
        w = get_rodrigues_rep_from_R(R)
        params.append(w[0])
        params.append(w[1])
        params.append(w[2])
        params.append(t[0])
        params.append(t[1])
        params.append(t[2])
    return np.array(params)

def get_matrices_from_params(params):
    # this is how we reconstruct our matrices using the parameters
    alpha_x = params[0]
    alpha_y = params[1]
    s = params[2]
    x_0 = params[3]
    y_0 = params[4]
    new_R_list = []
    new_t_list = []
    for idx in range(5,len(params),6):
        w_0 = params[idx]
        w_1 = params[idx + 1]
        w_2 = params[idx + 2]
        # we get our R back
        R_ = get_R_from_rodrigues(w_0, w_1, w_2)

        t_0 = params[idx + 3]
        t_1 = params[idx + 4]
        t_2 = params[idx + 5]
        t_ = np.array([t_0, t_1, t_2])

        new_R_list.append(R_)
        new_t_list.append(t_)

    K = np.array([[alpha_x, s, x_0],
```

```
                    [0, alpha_y, y_0],
                    [0, 0, 1]])

    return K, new_R_list, new_t_list

def get_projection_matrix(K, R, t):
    # this is just following the scrolls, this is the definition of
    projection matrix
    R_t = np.hstack((R, t[:,None]))
    P = K @ R_t
    return P

def error_f(params,pts_3d, pts_img_list):
    # this is the function we use to optimize,
    # first we get every parameter in a numpy array
    K, R_list, t_list = get_matrices_from_params(params)
    error = np.empty((0),dtype=np.float32)
    # get the error X - f, per image and append it to an error list
    for idx in range(len(R_list)):
        P = get_projection_matrix(K, R_list[idx], t_list[idx])
        reprojected_pts = apply_homography(pts_3d, P)
        # always the geometric error here
        err = np.abs((pts_img_list[idx] - reprojected_pts)).ravel()

        error = np.append(error, err, axis=0)

    return error

def reprojection_error(pts_3d, pts_img, P):
    # this is just another function to use to give us the same thing but
    this is just for one image
    reprojected_pts = apply_homography(pts_3d, P)
    return np.abs((pts_img - reprojected_pts)).ravel()

def lm_optim_P(pts_3d, pts_img_list, K, R_list, t_list):
    # this is our optimization function, we call scipy least squares here
    after getting all the parameters
    params = get_params_from_matrices(K, R_list, t_list)
    op_params = least_squares(error_f, params, method='lm', args=(pts_3d,
    pts_img_list),verbose=False).x
    # we reconstruct them and this is what we return
    K_op, R_op_list, t_op_list = get_matrices_from_params(op_params)

    return K_op, R_op_list, t_op_list

def get_camera_poses(R, t):
    # this is how we get our camera poses,
    # there is one modification we did, we are no longer adding the C into
    the directions,
    # as the function we call to draw the vectors in 3d gets the direction
    not the endpoint
    C = - R.T @ t

    X_xcam = np.array([1,0,0])
```

```
    X_ycam = np.array([0,1,0])
    X_zcam = np.array([0,0,1])

    X_x = R.T @ X_xcam
    X_y = R.T @ X_ycam
    X_z = R.T @ X_zcam


    return C, X_x, X_y, X_z

def plot_corners(image, corners_, name):
    # this is just to plot corners and the label
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    img = image.copy()
    cont = 0
    ms = 2
    plt.figure()
    plt.imshow(img);
    for corner in corners_:
        x = corner[0]
        y = corner[1]
        plt.plot(x,y, 'g.', markersize=ms)
        plt.text(x, y, str(cont), fontsize=8, color='g')
        cont += 1
    plt.axis('off');
    plt.savefig(name[:-4] + ".png",bbox_inches='tight',pad_inches=None);
    plt.close()
    plt.cla()
    plt.clf()

def plot_corners_improvement(image, corners_, corners_improved, name):
    # we use this to plot the detected corners and the reprojected corners
    # projected corners always in red and detected corners in green
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    img = image.copy()
    cont = 0
    ms = 2
    plt.figure()
    plt.imshow(img);
    for corner in corners_:
        x = corner[0]
        y = corner[1]
        plt.plot(x,y, 'r.', markersize=ms)
        #plt.text(x, y, str(cont), fontsize=8, color='g')
        #cont += 1
    for corner in corners_improved:
        x = corner[0]
        y = corner[1]
        plt.plot(x,y, 'g.', markersize=ms)
        plt.text(x, y, str(cont), fontsize=8, color='g')
        cont += 1
    plt.axis('off');
    plt.savefig(name[:-4] + ".png",bbox_inches='tight',pad_inches=None);
    plt.close()
```

```python
    plt.cla()
    plt.clf()

def reproject_to_fixed_image(fixed_img, fixed_image_data, other_image_list):
    #this is the function we use to call the reprojection into the fixed image,
    # we are only interested in the first, second and last column of the projection matrix,
    # so we take just that from the projection matrix and use the resulting 3 by 3 matrix to project the points
    P_fixed = fixed_image_data["projection_matrix"]
    H_fixed = P_fixed[:,[0,1,3]]
    corners_fixed = fixed_image_data["corners"]
    P_fixed_op = fixed_image_data["projection_matrix_new"]
    H_fixed_op = P_fixed_op[:,[0,1,3]]
    save_path = "improvement"
    for image_data in other_image_list:
        print("-"*10)
        print(image_data["filename"])
        corners = image_data["corners"]
        P_old = image_data["projection_matrix"]
        H_old = P_old[:,[0,1,3]]

        projection_3d_old = apply_homography(corners, np.linalg.inv(H_old))
        reprojected_fixedim_corners_old = apply_homography(projection_3d_old, H_fixed)

        plot_corners_improvement(fixed_img, reprojected_fixedim_corners_old, corners_fixed, os.path.join(save_path, image_data["filename"][:-4]+"_beforeLM.jpg"))

        P_op = image_data["projection_matrix_new"]
        H_op = P_op[:,[0,1,3]]

        projection_3d_op = apply_homography(corners, np.linalg.inv(H_op))
        reprojected_fixedim_corners_new = apply_homography(projection_3d_op, H_fixed_op)
        plot_corners_improvement(fixed_img, reprojected_fixedim_corners_new, corners_fixed, os.path.join(save_path, image_data["filename"][:-4]+"_afterLM.jpg"))
        err_old = np.linalg.norm(reprojected_fixedim_corners_old - fixed_image_data["corners"],axis=1)

        err_new = np.linalg.norm(reprojected_fixedim_corners_new - fixed_image_data["corners"],axis=1)

        print("Mean: ", err_old.mean())
        print("Mean after LM: ", err_new.mean())
        print("-"*4)
        print("Std: ", err_old.std())
        print("Std after LM: ", err_new.std())

```

```
def plot_camera_poses(data_dict_list,pattern_shape=(8,10),pattern_size=1., name=None):
    # this is how we plot the camera poses, it takes all the data at once and loops through it
    plt.close()
    plt.cla()
    plt.clf()
    fig = plt.figure(figsize=(11, 11))
    ax = fig.add_subplot(projection='3d')
    # this is for the calibration pattern
    for j in range(pattern_shape[1]-1):
        for i in range(pattern_shape[0]-1):
            # need to reflect the fact that it "skips" a line
            if j % 2 == 0:
                # alternate black and white
                color = 'black' if (i + j) % 2 == 0 else 'white'
                x = i * pattern_size
                y = j * pattern_size
                rect = Rectangle((x, y), pattern_size, pattern_size, color=color, alpha=1, edgecolor=None)
                ax.add_patch(rect)
                # Set the 3D position of each rectangle at Z=0
                pathpatch_2d_to_3d(rect, z=0, zdir="z")
            else:
                # in this case, it is all white squares
                color = 'white'
                x = i * pattern_size
                y = j * pattern_size
                rect = Rectangle((x, y), pattern_size, pattern_size, color=color, alpha=1, edgecolor=None)
                ax.add_patch(rect)
                # Set the 3D position of each rectangle at Z=0
                pathpatch_2d_to_3d(rect, z=0, zdir="z")
    # nowe we draw our poses
    for img_data_dict in data_dict_list:
        # loop through the data, choose a random color, the way matplotlib does it is with 0 to 1
        color_random = np.random.random(3)
        center = img_data_dict["camera_pose"]["center"]
        c_x, c_y, c_z = center
        x_dir = img_data_dict["camera_pose"]["x_vector"]
        x_x, x_y, x_z = x_dir
        y_dir = img_data_dict["camera_pose"]["y_vector"]
        y_x, y_y, y_z = y_dir
        z_dir = img_data_dict["camera_pose"]["z_vector"]
        z_x, z_y, z_z = z_dir

        # this is for our direction vectors
        ax.quiver(c_x, c_y, c_z, x_x, x_y, x_z, color='r', linewidth=1)
        ax.quiver(c_x, c_y, c_z, y_x, y_y, y_z, color='g', linewidth=1)
        ax.quiver(c_x, c_y, c_z, z_x, z_y, z_z, color='b', linewidth=1)

        # now the camera plane, in here we notice that the corners are made up from addition and substractions of the camera center and the
```

```
    directions,
        # so we make use to that
        points = [center + x_dir + y_dir,
                  center + x_dir - y_dir,
                  center - x_dir - y_dir,
                  center - x_dir + y_dir]

        poly = Poly3DCollection([points], color=color_random, alpha=0.4,
    edgecolor=None)
        ax.add_collection3d(poly)

    #ax.view_init(45,-90)
    # we set reasonable limits and plot it
    ax.set_xlim([-15, 15])
    ax.set_ylim([-15, 15])
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    if name:
        plt.savefig("camera_poses_.pdf",bbox_inches='tight',pad_inches=
    None);
    else:
        plt.savefig("camera_poses.pdf",bbox_inches='tight',pad_inches=None
    );
    plt.close()
    plt.cla()
    plt.clf()

img_path = "HW8-Files/Dataset1"
save_path = "imgs"
save_canny = "canny"
save_lines = "lines"
# we get all the files in the dataset to loop through
file_list = [x for x in os.listdir(img_path) if x.endswith(".jpg")]
imgs_data = []
H_list = []
corners_list = []
Vs = []
# again, 1 for length as we are using inches
RL_coords = get_coordinates(shape=(8,10),length=1)
# this is now our 3d coordinate list
RL_3d = add_number(RL_coords,0)

for file in file_list:
    img_data = {}
    img_data["filename"] = file
    img = cv2.imread(os.path.join(img_path,file))
    sname = os.path.join(save_path, file)
    canny_name = os.path.join(save_canny, file)
    line_name = os.path.join(save_lines, file)
    hough_name = os.path.join('houghlines', file)
    corners = get_corners(img, tol=15, save_canny=canny_name, save_lines=
    line_name, save_hough=hough_name)
    corners_list.append(corners)
```

```
    img_data["corners"] = np.array(corners)


    plot_corners(img, corners, sname);

    H = get_H_matrix(RL_coords, np.array(corners))
    img_data["homography"] = H
    #print(H)
    if len(corners) == 80:
        # need to make sure that the number of corners is 80
        H_list.append(H)

        V_h = get_V(H)
        Vs.append(V_h[0])
        Vs.append(V_h[1])

        imgs_data.append(img_data)
    else:
        print("Incorrect number of lines for ", file)

Vs = np.array(Vs)
omega = get_omega(Vs)
K = get_K_matrix(omega)

R_list = []
t_list = []

for idx in range(len(imgs_data)):
    R, t = get_extrinsics(K, imgs_data[idx]["homography"])
    imgs_data[idx]["rotation_matrix"] = R
    imgs_data[idx]["translation_vector"] = t
    imgs_data[idx]["projection_matrix"] = get_projection_matrix(K, R, t)
    R_list.append(R)
    t_list.append(t)
# now the optimization
K_op, R_op_list, t_op_list = lm_optim_P(RL_3d, corners_list, K, R_list,
    t_list)

for idx in range(len(imgs_data)):
    # loop through the data and populate what is missing
    imgs_data[idx]["rotation_matrix"] = R_list[idx]
    imgs_data[idx]["translation_vector"] = t_list[idx]
    imgs_data[idx]["projection_matrix"] = get_projection_matrix(K, R_list[
    idx], t_list[idx])

    imgs_data[idx]["K_op"] = K_op
    imgs_data[idx]["R_op"] = R_op_list[idx]
    imgs_data[idx]["t_op"] = t_op_list[idx]

    P_new = get_projection_matrix(K_op, R_op_list[idx], t_op_list[idx])
    P_old = get_projection_matrix(K, R_list[idx], t_list[idx])
    imgs_data[idx]["projection_matrix_new"] = P_new

```

```python
    imgs_data[idx]["old_reprojection_error"] = reprojection_error(RL_3d, imgs_data[idx]["corners"], P_old)
    imgs_data[idx]["new_reprojection_error"] = reprojection_error(RL_3d, imgs_data[idx]["corners"], P_new)
    imgs_data[idx]["err_mean_old"] = imgs_data[idx]["old_reprojection_error"].mean()
    imgs_data[idx]["err_mean_new"] = imgs_data[idx]["new_reprojection_error"].mean()
    imgs_data[idx]["err_std_old"] = imgs_data[idx]["old_reprojection_error"].std()
    imgs_data[idx]["err_std_new"] = imgs_data[idx]["new_reprojection_error"].std()

    C, X_x, X_y, X_z = get_camera_poses(R_op_list[idx], t_op_list[idx])
    camera_pose = {}
    camera_pose["center"] = C
    camera_pose["x_vector"] = X_x
    camera_pose["y_vector"] = X_y
    camera_pose["z_vector"] = X_z
    imgs_data[idx]["camera_pose"] = camera_pose

print("K before LM: ", K)
print("K after LM: ", K_op)

imgs_to_show = ["Pic_19.jpg","Pic_31.jpg"]
fixed_img = "Pic_11.jpg"
fixed_image = cv2.imread(os.path.join(img_path, fixed_img))
imgs_show_data = []
for img_data in imgs_data:
    if img_data["filename"] == fixed_img:
        fixed_img_data = img_data
    if img_data["filename"] in imgs_to_show:
        imgs_show_data.append(img_data)

reproject_to_fixed_image(fixed_image, fixed_img_data, imgs_show_data)

for img_show_data in imgs_show_data:
    print("-"*10)
    print(img_show_data["filename"])
    print("Before LM, rotation matrix: ",img_show_data["rotation_matrix"])
    print("Before LM, translation vector: ",img_show_data["translation_vector"])
    print("After LM, rotation matrix: ",img_show_data["R_op"])
    print("After LM, translation vector: ",img_show_data["t_op"])

    img = cv2.imread(os.path.join(img_path, img_show_data['filename']))
    new_points = apply_homography(RL_3d, img_show_data["projection_matrix"])
    new_points_improved = apply_homography(RL_3d, img_show_data["projection_matrix_new"])
    plot_corners_improvement(img, new_points, img_show_data["corners"], os.path.join("improvement",img_show_data["filename"][:-4]+"_corners_beforeLM.jpg"))
```

```
    plot_corners_improvement(img, new_points_improved, img_show_data["corners"], os.path.join("improvement",img_show_data["filename"][:-4]+"_corners_afterLM.jpg"))
    plot_corners_improvement(img, new_points, new_points_improved, os.path.join("improvement",img_show_data["filename"][:-4]+"_corners_comparison.jpg"))

plot_camera_poses(imgs_data)

# for the second dataset we follow exactly the same steps
img_path_ = "dataset2/small"
save_path = "imgs"
save_canny = "canny"
save_lines = "lines"
file_list_ = [x for x in os.listdir(img_path_) if x.endswith(".jpeg")]
imgs_data_ = []
H_list_ = []
corners_list_ = []
Vs_ = []
RL_coords = get_coordinates(shape=(8,10),length=1)
RL_3d = add_number(RL_coords,0)

for file in file_list_:
    img_data_ = {}
    img_data_["filename"] = file
    img = cv2.imread(os.path.join(img_path_,file))
    sname = os.path.join(save_path, file)
    canny_name = os.path.join(save_canny, file)
    line_name = os.path.join(save_lines, file)
    hough_name = os.path.join('houghlines', file)
    corners = get_corners(img, tol=20, save_canny=canny_name, save_lines=line_name, save_hough=hough_name, canny_p1=370, canny_p2=300, hough_p=50)
    corners_list_.append(corners)
    img_data_["corners"] = np.array(corners)

    if len(corners) == 80:

        plot_corners(img, corners, sname);

        H = get_H_matrix(RL_coords, np.array(corners))
        img_data_["homography"] = H
        #print(H)
        H_list_.append(H)

        V_h = get_V(H)
        Vs_.append(V_h[0])
        Vs_.append(V_h[1])

        imgs_data_.append(img_data_)
    else:
        print(file)
        print(len(corners))

```

```
Vs_ = np.array(Vs_)
omega = get_omega(Vs_)
K_ = get_K_matrix(omega)

R_list_ = []
t_list_ = []

for idx in range(len(imgs_data_)):
    R, t = get_extrinsics(K_, imgs_data_[idx]["homography"])
    imgs_data_[idx]["rotation_matrix"] = R
    imgs_data_[idx]["translation_vector"] = t
    imgs_data_[idx]["projection_matrix"] = get_projection_matrix(K_, R, t)
    R_list_.append(R)
    t_list_.append(t)

K_op_, R_op_list_, t_op_list_ = lm_optim_P(RL_3d, corners_list_, K_,
    R_list_, t_list_)

for idx in range(len(imgs_data_)):
    imgs_data_[idx]["rotation_matrix"] = R_list_[idx]
    imgs_data_[idx]["translation_vector"] = t_list_[idx]
    imgs_data_[idx]["projection_matrix"] = get_projection_matrix(K_,
    R_list_[idx], t_list_[idx])

    imgs_data_[idx]["K_op"] = K_op_
    imgs_data_[idx]["R_op"] = R_op_list_[idx]
    imgs_data_[idx]["t_op"] = t_op_list_[idx]

    P_new = get_projection_matrix(K_op_, R_op_list_[idx], t_op_list_[idx])
    P_old = get_projection_matrix(K_, R_list_[idx], t_list_[idx])
    imgs_data_[idx]["projection_matrix_new"] = P_new

    imgs_data_[idx]["old_reprojection_error"] = reprojection_error(RL_3d,
    imgs_data_[idx]["corners"], P_old)
    imgs_data_[idx]["new_reprojection_error"] = reprojection_error(RL_3d,
    imgs_data_[idx]["corners"], P_new)
    imgs_data_[idx]["err_mean_old"] = imgs_data_[idx]["
    old_reprojection_error"].mean()
    imgs_data_[idx]["err_mean_new"] = imgs_data_[idx]["
    new_reprojection_error"].mean()
    imgs_data_[idx]["err_std_old"] = imgs_data_[idx]["
    old_reprojection_error"].std()
    imgs_data_[idx]["err_std_new"] = imgs_data_[idx]["
    new_reprojection_error"].std()

    C, X_x, X_y, X_z = get_camera_poses(R_op_list_[idx], t_op_list_[idx])
    camera_pose = {}
    camera_pose["center"] = C
    camera_pose["x_vector"] = X_x
    camera_pose["y_vector"] = X_y
    camera_pose["z_vector"] = X_z
    imgs_data_[idx]["camera_pose"] = camera_pose

print("K before LM: ", K_)
```

```
print("K after LM: ", K_op_)

imgs_to_show = ["pose9.jpeg","pose33.jpeg"]
fixed_img = "fixed_img.jpeg"
fixed_image = cv2.imread(os.path.join(img_path_, fixed_img))
imgs_show_data = []
for img_data in imgs_data_:
    if img_data["filename"] == fixed_img:
        fixed_img_data = img_data
    if img_data["filename"] in imgs_to_show:
        imgs_show_data.append(img_data)

reproject_to_fixed_image(fixed_image, fixed_img_data, imgs_show_data)

print(fixed_img_data["rotation_matrix"])
print(fixed_img_data["translation_vector"])
print(fixed_img_data["R_op"])
print(fixed_img_data["t_op"])

for img_show_data in imgs_show_data:
    print("-"*10)
    print(img_show_data["filename"])
    print("Before LM, rotation matrix: ",img_show_data["rotation_matrix"])
    print("Before LM, translation vector: ",img_show_data["translation_vector"])
    print("After LM, rotation matrix: ",img_show_data["R_op"])
    print("After LM, translation vector: ",img_show_data["t_op"])

    img = cv2.imread(os.path.join(img_path_, img_show_data['filename']))
    new_points = apply_homography(RL_3d, img_show_data["projection_matrix"])
    new_points_improved = apply_homography(RL_3d, img_show_data["projection_matrix_new"])
    plot_corners_improvement(img, new_points, img_show_data["corners"], os.path.join("improvement",img_show_data["filename"][:-4]+"_corners_beforeLM.jpg"))
    plot_corners_improvement(img, new_points_improved, img_show_data["corners"], os.path.join("improvement",img_show_data["filename"][:-4]+"_corners_afterLM.jpg"))
    plot_corners_improvement(img, new_points, new_points_improved, os.path.join("improvement",img_show_data["filename"][:-4]+"_corners_comparison.jpg"))

plot_camera_poses(imgs_data_,name=True)
```

Listing 1: Source code