CSCE 643 Multi-View Geometry CV Project II

I. Calibrate Lens Distortion

A basic assumption in camera geometry study is that a linear model is an accurate model of the imaging process, i.e., world point, image point and optical center point are collinear and world lines are imaged as lines and so on. However, this might not be true for real lens in the world. Radial distortion is a typical deviation that real lens might have, we need to correct image measurements to those that should be obtained under a perfect linear camera action.

A. Problem Formulation

Suppose $(\tilde{x}, \tilde{y})$ is the ideal image position that obeys linear projection (under perfect lens settings), (x_d, y_d) to be the actual image position after the effects of radial distortion, $\tilde{r} = \sqrt{\tilde{x^2} + \tilde{y^2}}$ is the radial distance from the center of radial distortion, $L(\tilde{r})$ is a function of radius $\tilde{r}$ as the distortion factor.

For the coordinates of point under non-distorted pinhole projection $(\tilde{x}, \tilde{y})$ (in units of focal-length), we have:

$$(\tilde{x}, \tilde{y}, 1)^T = [I|\mathbf{0}]\mathbf{x}_{cam} \tag{1}$$

where $\mathbf{x}_{cam}$ is the 3D point in camera coordinates and it is related to world coordinates as follows:

$$\mathbf{x}_{cam} = \begin{bmatrix} R & -R\tilde{\mathbf{C}} \\ 0 & 1 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{bmatrix} R & -R\tilde{\mathbf{C}} \\ 0 & 1 \end{bmatrix} \mathbf{x} \tag{2}$$

Now we can easily model the radial distortion by introducing the function of $\tilde{r}$:

$$\begin{pmatrix} x_d \\ y_d \end{pmatrix} = L(\tilde{r}) \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} \tag{3}$$

If we dig into a single point coordinate (x, y), we have:

$$\begin{aligned} \tilde{x} &= x_c + L(r)(x - x_c) \\ \tilde{y} &= y_c + L(r)(y - y_c) \end{aligned} \tag{4}$$

where (x_c, y_c) is center of the image, $(\tilde{x}, \tilde{y})$ is the coordinates after correcting radial distortion. Also be noticed that r can is the radius from center of the image to the measured coordinates, which can be solved by:

$$r^2 = (x - x_c)^2 + (y - y_c)^2 \tag{5}$$

B. Distortion function and image center

Now let's talk about the choice of function $L(r)$ and center of image (x_c, y_c). According to the textbook, $L(r)$ is only defined for positive r and $L(0) = 1$. We can use Taylor expansion as an approximation of the distortion function:

$$L(r) = 1 + k_1 r + k_2 r^2 + k_3 r^3 + \dots \tag{6}$$

For image center (x_c, y_c), we often use principal point though it might not reside in the exact same location. On choosing the distortion function for this project, we did a tradeoff consideration. Apparently if we have higher order Taylor expansion for distortion function, we can get better correcting results, however, as the camera we are actually using is really good (though it is a relatively old model, modern cameras are generally good enough and generates very little distortion), we decided to choose $L(r) = 1 + k_1 r + k_2 r^2$ as our distortion function. The intuition behind this is that typically we choose higher order of distortion function as the degree of distortion increases, in our case the distortion is even ignorable thus we use a typical 2-order function to deal with our case.

C. Distortion function minimization

Similar as the minimization process we have done in the previous projects, here we need to minimize the geometric distance between predicted coordinates (by using $L(r)$) and measured coordinates. This can be done by exploiting the colinearity of points on the same line. In our project, we use the checker board image captured by our camera for lens distortion correction. The basic idea is that we can easily identify corner points on checker board and they naturally form parallel and perpendicular lines. If we determine a straight line using two corner points on the line, then all the other corner points along the line should be on the line if without radial distortion. Let's consider a simple scenario, suppose we have $\mathbf{A}, \mathbf{B}, \mathbf{D}$ on same line of a checker board, and the image center is $\mathbf{C}$, then we can do cross product to solve the crossing product of $\vec{AB}$ and $\vec{CD}$ to get $\mathbf{D'} = \vec{AB} \times \vec{CD}$, due to the radial distortion we have $\mathbf{D} \neq \mathbf{D'}$. If we apply $L(r)$ for the measured D to get corrected coordinates $\hat{\mathbf{D}}$, we want to find a function that minimizes $d(\hat{\mathbf{D}}, \mathbf{D'})$. This is now reduced to a simple minimization problem of geometric distance like we did before, we can construct the corresponding error function based on discussions above and use non linear solver in MATLAB to get function $L(r)$ that does the required minimization.

D. Experiment Results

To evaluate the performance of our approach, we started an experiment. We fixed the focal length of our camera by turning off the auto focus function and stick a checker board on the wall as the object we use for calibration, the original colored photo we get from the camera is shown in Figure 1.

Fig. 1. The original colored picture we used for lens undistortion

Then we grayize the photo and then use corner function provided by MATLAB to find out the corner points, the result we got from here is shown in Figure 2(a). We then do the following steps:

- Specify two perpendicular lines we want to use for this calibration.
- Identify all the corner points from the image.
- Compute the line through cross product of two end points.
- Compute the lines from image center to each point on those perpendicular lines, we call them lines to center.
- Use cross product to calculate the interceptions between two perpendicular lines and lines to center.
- Now we can calculate geometric distance between interceptions and measured points, we further assume $L(r) = 1 + k_1 r + k_2 r^2$ and use *lsqnonlin* with LM option to get the k that minimizes the geometric distance.
- After solving $L(r)$ function, we use a function for undistorting image which leverages the $L(r)$ function to remove the distortion of image.

The result after removing the distortion is also provided in Figure 2(b). Notice that though there is no significant different between two images due to the fact that there are few radial distortion in our camera lens, we refer readers to display the original undistorted image and compare it to the original one, we can still notice changes this way.

II. Camera Calibration

Recall from projects before, we have implemented DLT approach for finding the transformation homography from one image to another. Now in the camera calibration process, we can use similar methods, let's first adapt the original DLT approach and normalization process to camera calibration.

A. DLT Foundation

In the simplest DLT approach, we first transform the homogeneous equation given in equation 7 as follows:

$$\mathbf{x}_i \times \mathbf{P}\mathbf{X}_i = \mathbf{0} \tag{7}$$

Where $\mathbf{P}$ is the camera matrix and can be decomposed into combination of vectors like:

$$\mathbf{P} = \begin{pmatrix} \mathbf{p_1} \\ \mathbf{p_2} \\ \mathbf{p_3} \end{pmatrix} \tag{8}$$

Then we can rewrite equation 7 into the following form:

$$\mathbf{P}\mathbf{X}_i = \begin{pmatrix} \mathbf{p}_1\mathbf{X}_i \\ \mathbf{p}_2\mathbf{X}_i \\ \mathbf{p}_3\mathbf{X}_i \end{pmatrix} \tag{9}$$

Suppose the coordinates of the set of points we have are:

$$\mathbf{x}_i = (x_i, y_i, z_i) \tag{10}$$

As we have $\mathbf{p}_i\mathbf{X}_i = \mathbf{X}_i^T\mathbf{p}_j^T$, we can further get the following simultaneous linear equations set:

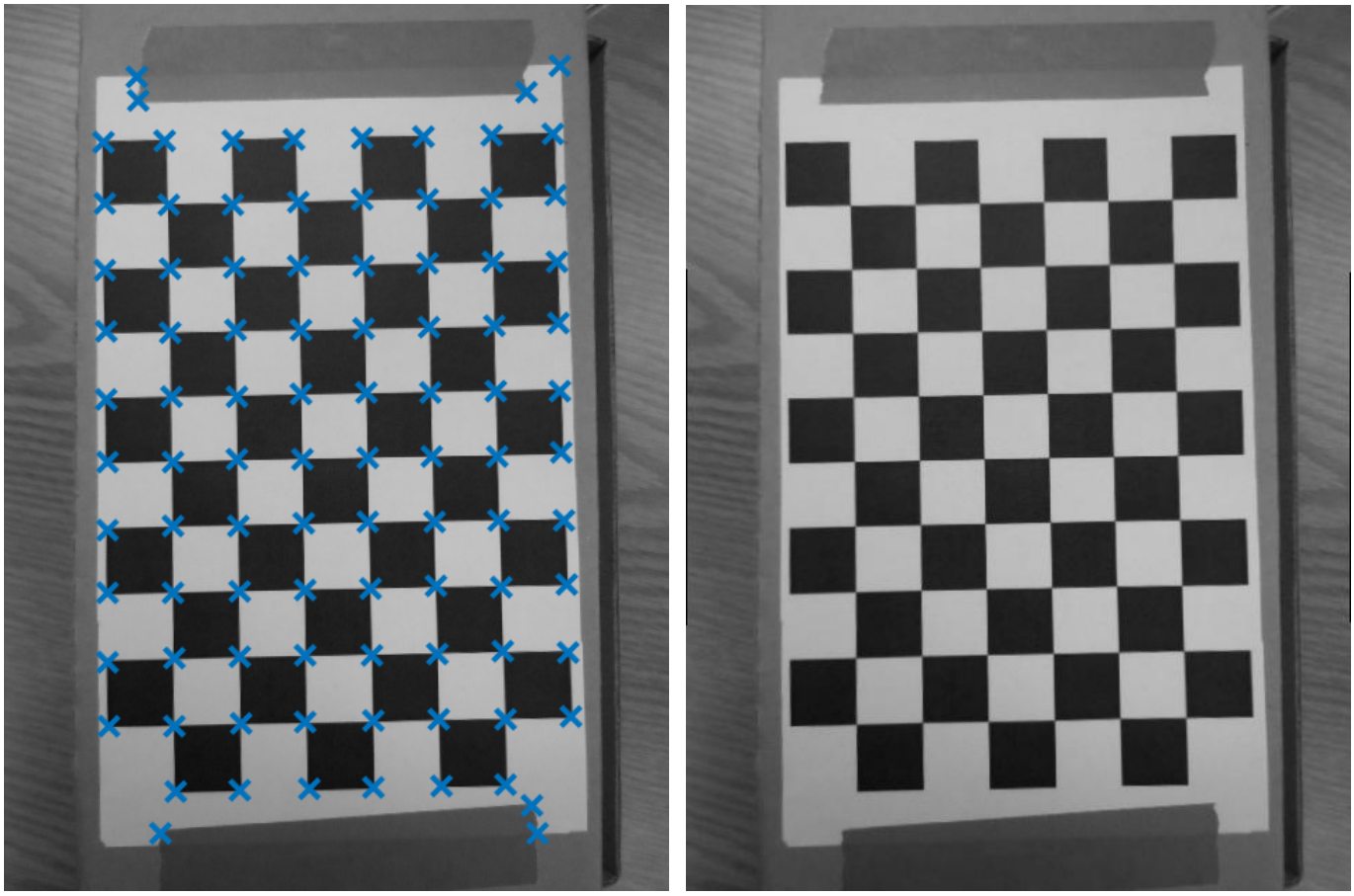
$$\begin{bmatrix} \mathbf{0}^T & -z_i\mathbf{X}_i^T & y_i\mathbf{X}_i^T \\ z_i\mathbf{X}_i^T & \mathbf{0}^T & -x_i\mathbf{X}_i^T \\ -y_i\mathbf{X}_i^T & x_i\mathbf{X}_i^T & \mathbf{0}^T \end{bmatrix} \begin{pmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \mathbf{p}_3^T \end{pmatrix} = \mathbf{0} \tag{11}$$

Be noticed that different from the math we did before in last project, here $\mathbf{X}_i$ is a 4D coordinates and p_i also contains 4 entries. Since we can obtain the 3rd row of the left matrix above using by combining first two rows up to scale, we can safely prune equation 7 to only preserve the linearly independent first two rows in that matrix and obtain:

$$\mathbf{A}\mathbf{p} = \mathbf{0} \tag{12}$$

where

$$\mathbf{A} = \begin{bmatrix} \mathbf{0}^T & -z_i\mathbf{X}_i^T & y_i\mathbf{X}_i^T \\ z_i\mathbf{X}_i^T & \mathbf{0}^T & -x_i\mathbf{X}_i^T \\ -y_i\mathbf{X}_i^T & x_i\mathbf{X}_i^T & \mathbf{0}^T \end{bmatrix}, \mathbf{p} = \begin{pmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \mathbf{p}_3^T \end{pmatrix} \tag{13}$$



(a) Grayized image with corner markers

(b) The image after removing lens distortion

Fig. 2. Remove lens distortion

As we know:

$$\mathbf{P} = \begin{pmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix} = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{bmatrix} \tag{14}$$

Now we have come to the point that is very similar to what we have done before in Project 1. After dehomogenization, the equation 12 we get is actually a equation with 11 unknowns in $\mathbf{p}$ (11 DoFs) which we can solve exactly if we have $5\frac{1}{2}$ point pairs (every point pair provide 2 equations), to get the homography we can do as the followings:

- If we have exact $5\frac{1}{2}$ point correspondences (which means the we can only know one of the coordinate for sixth points) in both images, we can solve equation 12 to get an exact solution, but as there might be some noises in both images, it could lead to very bad results.
- We can also solve this equation is we can find more point correspondence to get so-called overdetermined solution through singular value decomposition (SVD), that way we should get more accurate results since we are provided with more information in the image.

B. Normalized DLT

1) 2D Image Frame Normalization: Similar to the DLT part we did in finding transformation homography between two images, it's sometimes very important to carry out coordinates normalization to provide better homography (camera matrix in current case). $\mathbf{x}_i$ is in the same dimension as before, thus we can normalize it using previous method, i.e., the points should be translated so that centroid of all points is at the origin and we also need to scale them so that their RMS (root-mean-squared) distance from origin is $\sqrt{2}$. As we have talked and implemented in the project before, the normalization process for 2D space can be summarized as follows:

- Compute a similarity transformation $\mathbf{T}$, consisting of a translation and scaling that maps $\mathbf{x}_i$ to $\bar{\mathbf{x}}_i$ such that the centroid of all points $\bar{\mathbf{x}}_i$ is the coordinate origin $(0, 0)^T$ and their average distance from the origin is $\sqrt{2}$.
- Compute a similar transformation $\mathbf{T}'$ using the same process to map $\mathbf{x}'_i$ to $\bar{\mathbf{x}}'_i$.
- Apply simple DLT in Problem 1 using the new corresponding point pairs $\bar{\mathbf{x}}_i \leftrightarrow \bar{\mathbf{x}}'_i$ to obtain the normalized homography $\bar{\mathbf{H}}$
- Use $\mathbf{H} = \mathbf{T}'^{-1}\bar{\mathbf{H}}\mathbf{T}$ to denormalize the homography from last step and get the actual homography.

2) 3D World Frame Normalization: However, with the introduction of points in world frame, we now have to consider the 3D points $\mathbf{X}_i$. To simplify the problem, here we consider the case where the variation in point depth from the camera is relatively slight we can do similar normalization for 3D points as we did for 2D points. Therefore, we translate centroid of all measured points to the origin and scale their coordinates so that RMS distance from the origin is $\sqrt{3}$. As mentioned in the textbook, such 2D-alike approach is actually suitable for a compact distribution of points. With few changes, we can build the 3D process of normalization as follows:

- Compute a similarity transformation $\mathbf{T}$, consisting of a translation and scaling that maps $\mathbf{X}_i$ to $\bar{\mathbf{X}}_i$ such that the centroid of all points $\bar{\mathbf{X}}_i$ is the coordinate origin $(0, 0, 0)^T$ and their average distance from the origin is $\sqrt{3}$, notice that here $\mathbf{X}_i$ and $\bar{\mathbf{X}}_i$ are in 3D space.
- Compute a similar transformation $\mathbf{T}'$ using the same process to map $\mathbf{X}'_i$ to $\bar{\mathbf{X}}'_i$.
- Apply simple DLT using the new corresponding point pairs $\bar{\mathbf{X}}_i \leftrightarrow \bar{\mathbf{X}}'_i$ to obtain the normalized homography $\bar{\mathbf{H}}$
- Use $\mathbf{H} = \mathbf{T}'^{-1}\bar{\mathbf{H}}\mathbf{T}$ to denormalize the homography from last step and get the actual homography.

C. Build World Coordinate System

As we showed in the above sections, the greatest different in applying DLT into camera calibration is that we are now mapping points from 3D world space to 2D image space, i.e., we will have $\mathbf{X}_i$ to be 4D (3 dimensions for coordinates, 1 for homogenization). In the previous DLT we implemented, both $\mathbf{x}_i$ and $\mathbf{x}'_i$ are selected from actual measurements and they are all in 2D (3D after homogenization) space. In our case for this project, we already have the 2D image space measurements while the real-world point locations cannot be obtained directly. Therefore, we introduce our definition of a world coordinate system here in order to obtain corresponding world coordinates for every points on the image. As shown in Figure 3, we use the baseline of the left checker board (lowest line that goes through edge of black ceils) as the x-axis, the baseline of right checker board as y-axis, and the intersection line between two checker boards as the z-axis.

We still need more information after the coordinate system is established, we need a well-defined distance metric for the world frame. Here we have measured the edge length of both black ceils and white ceils to be 24.5 mm, the distance between z-axis and right most black ceil edges on left checker board to be 21 mm, the distance between z-axis and left mots black ceil edges on right checker board to be 23.5 mm. Be noticed that we don't necessarily need a unified unit like mm when building the system, we just need to make sure the distance we assigned through the system is proportional to the actual distance. Now we have built a solid world frame coordinate system and apparently through the measurements of distance above, we can compute the real-world coordinates of every corner points in the image. To ensure the correspondence of real-world coordinates and coordinates in the image, we have to do some sorting algorithms on corner points and follow the same sequence while obtaining coordinates in both spaces, we skipped the details in doing this and we welcome readers to find out through codes attached in appendix.

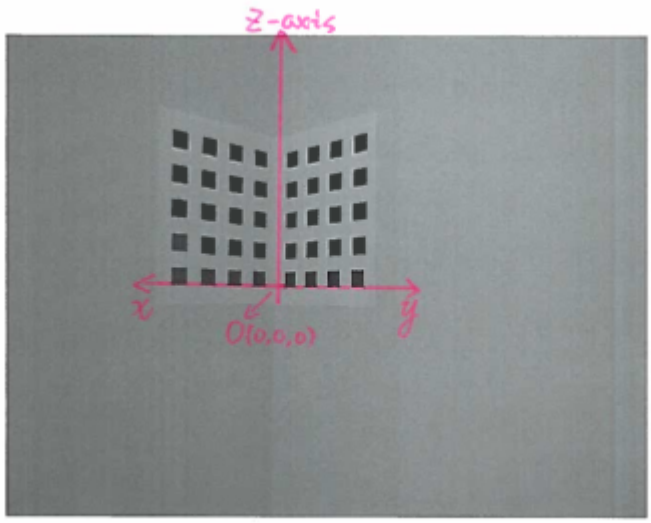


Fig. 3. Establishing the World Coordinate System

D. Camera Calibration Steps

According to the textbook, the camera calibration steps can be carried out as shown in Figure 4.

After performing all those steps we can get the calibration results. Since we have lots of noise points in the previous image we are using, here we apply another image for the camera calibration process, and the calibrated pictures are shown in Figure 5.

Be noticed that in the result figure, points that are marked blue in Figure 5(a) are all the corner points for calibration, red points in Figure 5(b) are the calibrated points. And we show the overlapped calibrated results with original point choices in Figure 5(c). If we zoom in the picture we can see the little geometric distance from calibrated result points to original point, however the variances between them are still trivial.

III. Test Calibration Result

The purpose of this section is to evaluate the result of our calibration. We followed the procedure in project requirements, that is, we used a coin and place it in the center of 10 different white ceils for taking the photo. Then we select the object positions in the image and record its coordinates (should be the center of white ceil that it was in). Through applying the $\mathbf{P}$ matrix, we can get the theoretical coordinates of the actual point into the camera frame $\mathbf{x} = \mathbf{XP}$ and compare it with the coordinates we recorded to compute errors and variances. Two examples of images we choose after placing the coins can be found in Figure 6. The calibration resulted errors and variances is presented in the section of Report Requirements. As the code for doing calibration is the same with section before, the only codes needed here is to read new points on new images, sum and average the errors and variances.

Objective

Given $n \geq 6$ world to image point correspondences $\{\mathbf{X}_i \leftrightarrow \mathbf{x}_i\}$, determine the Maximum Likelihood estimate of the camera projection matrix $\mathtt{P}$, i.e. the $\mathtt{P}$ which minimizes $\sum_i d(\mathbf{x}_i, \mathtt{P}\mathbf{X}_i)^2$.

Algorithm

(i) **Linear solution.** Compute an initial estimate of $\mathtt{P}$ using a linear method such as algorithm 4.2(p109):

(a) **Normalization:** Use a similarity transformation $\mathtt{T}$ to normalize the image points, and a second similarity transformation $\mathtt{U}$ to normalize the space points. Suppose the normalized image points are $\tilde{\mathbf{x}}_i = \mathtt{T}\mathbf{x}_i$, and the normalized space points are $\tilde{\mathbf{X}}_i = \mathtt{U}\mathbf{X}_i$.

(b) **DLT:** Form the $2n \times 12$ matrix $\mathtt{A}$ by stacking the equations (7.2) generated by each correspondence $\tilde{\mathbf{X}}_i \leftrightarrow \tilde{\mathbf{x}}_i$. Write $\mathbf{p}$ for the vector containing the entries of the matrix $\tilde{\mathtt{P}}$. A solution of $\mathtt{A}\mathbf{p} = \mathbf{0}$, subject to $\|\mathbf{p}\| = 1$, is obtained from the unit singular vector of $\mathtt{A}$ corresponding to the smallest singular value.

(ii) **Minimize geometric error.** Using the linear estimate as a starting point minimize the geometric error (7.4):

$$\sum_i d(\tilde{\mathbf{x}}_i, \tilde{\mathtt{P}}\tilde{\mathbf{X}}_i)^2$$

over $\tilde{\mathtt{P}}$, using an iterative algorithm such as Levenberg–Marquardt.

(iii) **Denormalization.** The camera matrix for the original (unnormalized) coordinates is obtained from $\tilde{\mathtt{P}}$ as

$$\mathtt{P} = \mathtt{T}^{-1}\tilde{\mathtt{P}}\mathtt{U}.$$

Fig. 4. Algorithm of Camera Calibration using Gold Standard

IV. Toolbox result comparison

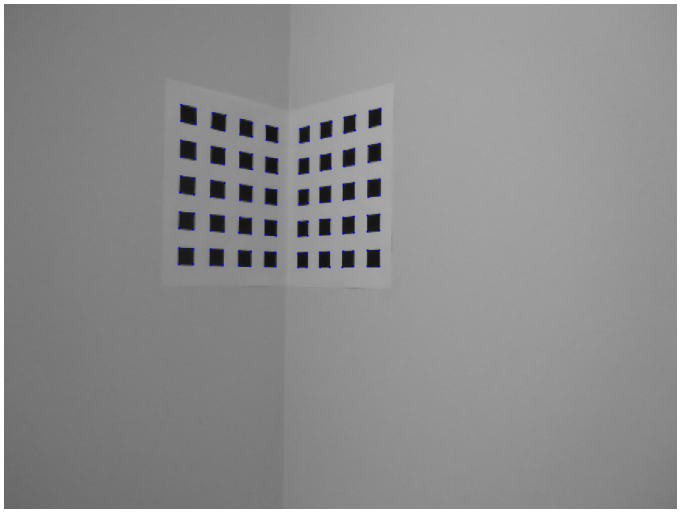
The toolbox usage for camera calibration can be summarized as follows:

- Installation
  - Download all files from the website.
  - Extract the compressed folder to certain directory and add this directory to MATLAB PATH variable.
  - Input command *calib_gui* in MATLAB to open the toolbox.
- Corner Detection
  - Select four corners of the checker board.
  - Input the number of corners we have in each axis (notice that the actual numbers we need to input is the total squares we have minus 1) defined by the toolbox in the figure.
  - The tool will automatically generate markers on all corner points and we can check if they are correct for use, as showin in Figure 7.
- Calibration
  - Input the edge of squares we have in figures (in our case I used 245).
  - We can choose if we need a initial guess for lens distortion (if the points marked in figure are relatively far away from actual corners), this is actually the same as we do in the project.
  - Do calibration and get the results through the command line. Be noticed that the results we are getting are fc (focal length), cc (principal point), alpha_c (skew), kc (k for removing lens distortion) and err (pixel error). We can further compose the K matrix for calibration using fc, cc and alpha_c.

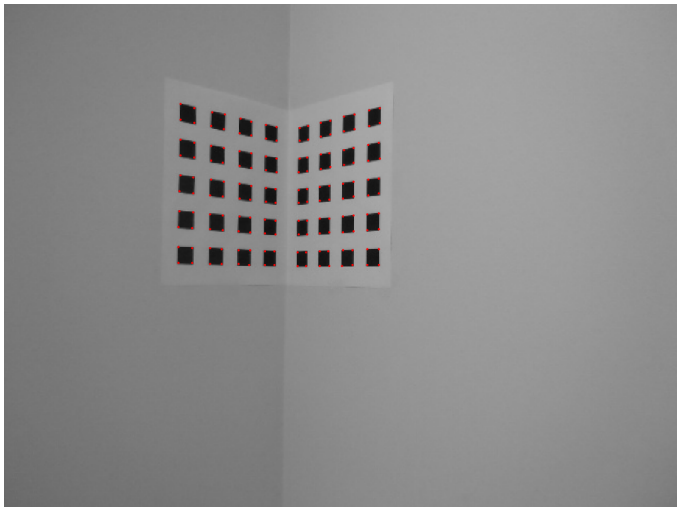
An example of results we get from the toolbox (after removing lens distortion) is shown in Figure 8.

V. Report requirement

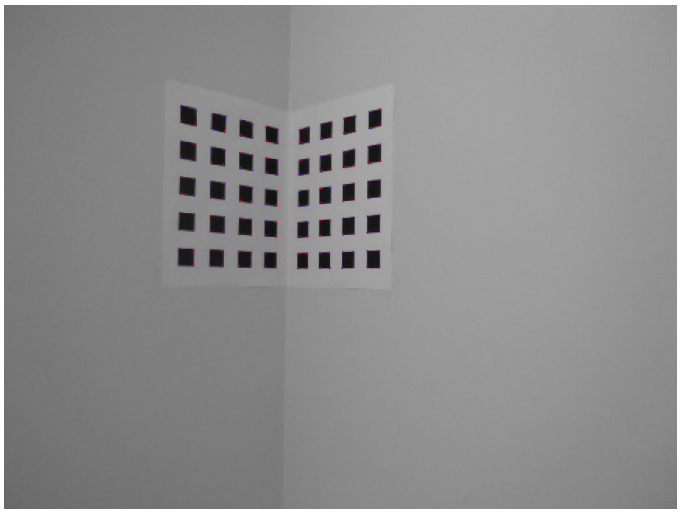
- Set of points used for lens distortion calibration (superimpose your points on top of your calibration object.):



(a) Point choice in the new image

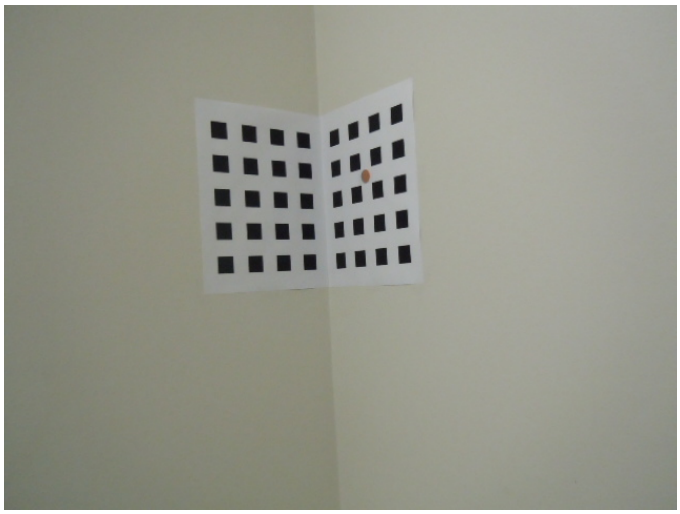


(b) Point after calibration

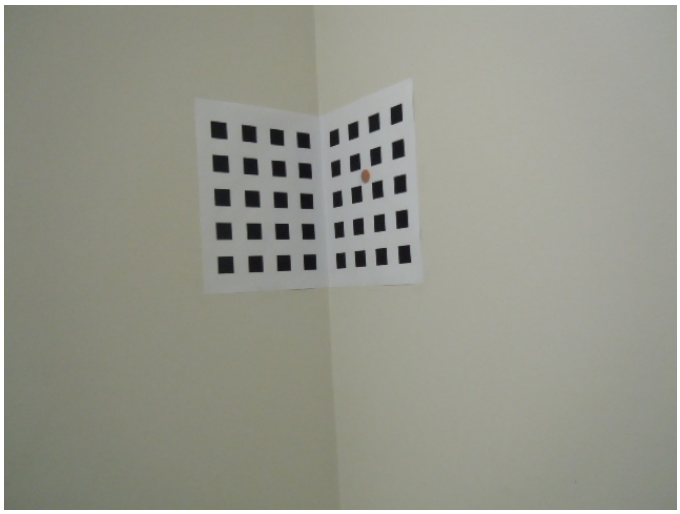


(c) Correspondences of point after calibration and before

Fig. 5. Camera Calibration Results



(a) Sample 1



(b) Sample 2

Fig. 6. Sample images we took for testing calibration

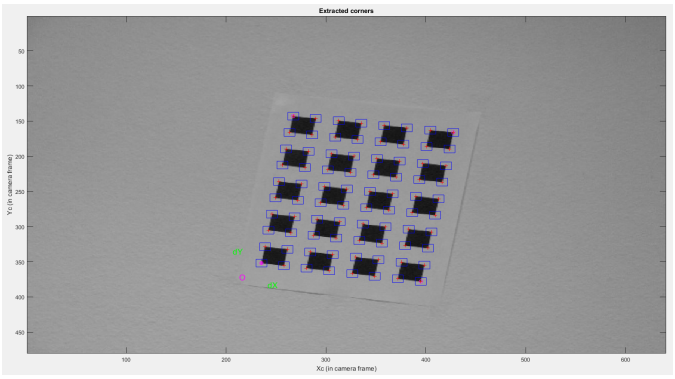


Fig. 7. Toolbox Corner Detection

```
Calibration results after optimization (with uncertainties):

Focal Length:          fc = [ 2373.88011   2398.58830 ] +/- [ 2698.81159   2654.44721 ]
Principal point:       cc = [ 319.50000   239.50000 ] +/- [ 0.00000   0.00000 ]
Skew:             alpha_c = [ 0.00000 ] +/- [ 0.00000  ]  => angle of pixel axes = 90.00000 +/- 0.00000 degrees
Distortion:            kc = [ 0.21334   80.60062   -0.01321   0.01094  0.00000 ] +/- [ 2.47790   644.89213   0.02083   0.03147  0.00000 ]
Pixel error:          err = [ 0.14815   0.15589 ]

Note: The numerical errors are approximately three times the standard deviations (for reference).


Recommendation: Some distortion coefficients are found equal to zero (within their uncertainties).
               To reject them from the optimization set est_dist=[0;0;1;1;0] and run Calibration


Calibration results (with uncertainties):

Focal Length:          fc = [ 2373.88011   2398.58830 ] ± [ 2698.81159   2654.44721 ]
Principal point:       cc = [ 319.50000   239.50000 ] ± [ 0.00000   0.00000 ]
Skew:             alpha_c = [ 0.00000 ] ± [ 0.00000  ]  => angle of pixel axes = 90.00000 ± 0.00000 degrees
Distortion:            kc = [ 0.21334   80.60062   -0.01321   0.01094  0.00000 ] ± [ 2.47790   644.89213   0.02083   0.03147  0.00000 ]
Pixel error:          err = [ 0.14815   0.15589 ]

Note: The numerical errors are approximately three times the standard deviations (for reference).
```

Fig. 8. Result From Calibration Toolbox after Removing Lens Distortion

the set of points used for lens distortion is provided in Appendix.

- Lens distortion parameter sets and model (what is the order of the polynomial? how do you decide to omit high order terms?): as mentioned before in Section I, typically we choose the distortion param and model with consideration of the degree of radial distortion generated by lens. With more severe distortion, we need to use higher order Tyler expansion as $L(r)$ function. However, in our case, the radial distortion of our camera is very slight, thus we choose to use the typical 2 order expansion. We also provided results of using higher order (3 and 4 order) Tyler expansion function in the appendix, however the final images using those functions with different orders look the same.
- Images before and after correcting the lens distortion: all the required images are provided in Section I.
- Set of points used for camera calibration (superimpose your points on top of your calibration object.): Please refer to Figure 5(a) for the super imposed point choice, however since we have too many point choices we skipped all the coordinates of those points here.
- Math about calibration process: shown in Section II.
- Matrix P(K) from both the your calibration algorithm and matlab toolbox: The result we got from tool box is shown in Appendix.
- Results from testing objects, point positions (measured vs. those computed from P), mean error and variance in image coordinate system: the mean error we get in testing objects is: 8.003623 while the variance we got is 13.145783. (This is the actual best results we got, and the results really do vary a lot with different image input in different lighting conditions).
- Discussion about what you learn from the process.

VI. Discussion

As we mentioned before, typically we choose the distortion param and model with consideration of the degree of radial distortion generated by lens. With more severe distortion, we need to use higher order Tyler expansion as $L(r)$ function. However, in our case, the radial distortion of our camera is very slight, thus we choose to use the typical 2 order expansion. To achieve a better accuracy in corner detection (especially in terms of the sequence for our program to recognize corners), we have to ensure that the photo is taken under good illumination and we'd better hold the camera horizotally parallel to ground so that the x/y coordinates of points on the same line remain the same to avoid further processing in the code. In the first time when I took photos, I choosed a relatively complex environment, and the other useless stuffs showed in the images are recognized as corners, which also poses increasing difficulty in processing image. We also learnt that the modern camera can already provide good photos as they have few lens distortions.

I would like to thank every classmates for their kindly help for me in this project, and several friends of mine who set up the environment for taking photos with me. Also, I will not to be able to finish the last problem without the kindly extension from Prof. Song.

VII. Intermediate Results

A. Lens Distortion

The intercept points we solved by cross product lines to center and row/column lines are shown as follows:

$$\mathbf{I} = \begin{pmatrix} 73.0 & 101.0 & 1.0 \\ 73.41 & 144.4 & 1.0 \\ 73.88 & 192.9 & 1.0 \\ 74.29 & 236.6 & 1.0 \\ 74.76 & 286.0 & 1.0 \\ 75.19 & 331.1 & 1.0 \\ 75.66 & 380.1 & 1.0 \\ 76.09 & 424.9 & 1.0 \\ 76.56 & 474.5 & 1.0 \\ 77.0 & 521.0 & 1.0 \\ 396.0 & 95.0 & 1.0 \\ 352.1 & 95.82 & 1.0 \\ 303.1 & 96.73 & 1.0 \\ 259.0 & 97.54 & 1.0 \\ 209.9 & 98.46 & 1.0 \\ 117.1 & 100.2 & 1.0 \\ 165.8 & 99.28 & 1.0 \\ 73.0 & 101.0 & 1.0 \end{pmatrix} \tag{15}$$

The actual measurements we get directly from the image (Set of points used for lens distortion calibration), they are also superimposed in Figure 2(a):

$$\mathbf{M} = \begin{pmatrix} 73.0 & 101.0 & 1.0 \\ 74.0 & 145.0 & 1.0 \\ 74.0 & 193.0 & 1.0 \\ 75.0 & 237.0 & 1.0 \\ 75.0 & 286.0 & 1.0 \\ 76.0 & 331.0 & 1.0 \\ 76.0 & 380.0 & 1.0 \\ 76.0 & 425.0 & 1.0 \\ 76.0 & 475.0 & 1.0 \\ 77.0 & 521.0 & 1.0 \\ 396.0 & 95.0 & 1.0 \\ 352.0 & 96.0 & 1.0 \\ 303.0 & 97.0 & 1.0 \\ 259.0 & 98.0 & 1.0 \\ 210.0 & 99.0 & 1.0 \\ 117.0 & 100.0 & 1.0 \\ 166.0 & 100.0 & 1.0 \\ 73.0 & 101.0 & 1.0 \end{pmatrix} \tag{16}$$

We init k in function $L(r)$ as follows and use minimization by LM to get k_{final} as follows:

$$\begin{aligned} k_{init} &= \begin{pmatrix} 0.05 & 0.025 \end{pmatrix} \\ k_{final} &= \begin{pmatrix} 3.579 \cdot 10^{-5} & -1.32 \cdot 10^{-7} \end{pmatrix} \end{aligned} \tag{17}$$

The k we obtained through 3 order $L(r)$ function:

$$\begin{aligned} k_{init} &= \begin{pmatrix} 0.05 & 0.025 & 0.005 \end{pmatrix} \\ k_{final} &= \begin{pmatrix} 7.351 \cdot 10^{-5} & -4.636 \cdot 10^{-7} & 7.15 \cdot 10^{-10} \end{pmatrix} \end{aligned} \tag{18}$$

B. Camera Calibration

In the camera calibration experiments, we tested our algorithm both with and without lens distortion to see the influence of lens distortion:

1) Results With Lens Distortion: The $\mathbf{P}$ matrix of our camera is calculated to be:

$$\mathbf{P} = \begin{pmatrix} -677.0 & 222.3 & 13.15 & 2.786 \cdot 10^5 \\ -202.0 & -178.7 & -611.0 & 2.603 \cdot 10^5 \\ -0.7337 & -0.6792 & 0.0192 & 1033.0 \end{pmatrix} \tag{19}$$

The matrix $\mathbf{K}$ we obtained is as follows:

$$\mathbf{K} = \begin{pmatrix} 623.1 & -0.3841 & 346.0 \\ 0 & 616.1 & 257.9 \\ 0 & 0 & 1.0 \end{pmatrix} \tag{20}$$

2) Results after Removing Lens Distortion: The $\mathbf{P}$ matrix of our camera is calculated to be:

$$\mathbf{P} = \begin{pmatrix} 681.6 & -225.8 & -14.15 & -2.803 \cdot 10^5 \\ 201.8 & 179.7 & 617.2 & -2.622 \cdot 10^5 \\ 0.7306 & 0.6824 & -0.02201 & -1040.0 \end{pmatrix} \tag{21}$$

The matrix $\mathbf{K}$ we obtained is as follows:

$$\mathbf{K} = \begin{pmatrix} 630.3 & -0.08575 & 344.2 \\ 0 & 623.0 & 256.5 \\ 0 & 0 & 1.0 \end{pmatrix} \tag{22}$$

3) Results from MATLAB Calibration Toolbox: The matrix $\mathbf{K}$ we obtained is as follows:

$$\mathbf{K} = \begin{pmatrix} 2373.88011 \pm 2698.81159 & 0 & 319.50000 \\ 0 & 2398.58830 \pm 2654.44721 & 239.50000 \\ 0 & 0 & 1.0 \end{pmatrix} \tag{23}$$

APPENDIX

A. Remove Lens Distortion

```
clear all; close all; clc;
single_fig = imread('pics/single.jpg');
single_fig = rgb2gray(single_fig);
figure(1); imshow(single_fig);
hold on;

% get all corner points in the single checker board figure
raw_corners = get_corners(single_fig);

% draw all corner points on the figure
% h = plot(raw_corners(:, 1), raw_corners(:, 2), 'x', 'Color', 'r', 'MarkerSize', 15)
    ;
% set(h,'linewidth',3);

% corners sorted by x, 1st row
s_corners_x = sortrows(raw_corners, 1);
s_corners_y = sortrows(raw_corners, 2);

% change this for new image
% get desired horizotal points and vertical points
vPts = s_corners_x(1:10, :);
hPts = s_corners_y(5:12, :);
% show all corner points on img
%h = plot(hPts(:, 1), hPts(:, 2), 'x', 'Color', 'r', 'MarkerSize', 15);
%set(h,'linewidth',3);
%h = plot(vPts(:, 1), vPts(:, 2), 'x', 'Color', 'b', 'MarkerSize', 15);
%set(h,'linewidth',3);

% change this for new image
% compute count of vertical and horizontal points
global vPtsCount hPtsCount;
vPtsCount = 10 - 1 + 1;
hPtsCount = 12 - 5 + 1;

% compute horizontal line and vertical line
% use the first point and the last point
vLine = cross(vPts(1, :), vPts(vPtsCount, :));
hLine = cross(hPts(1, :), hPts(hPtsCount, :));
% normalize
vLine = [vLine(1)/vLine(3), vLine(2)/vLine(3), 1];
hLine = [hLine(1)/hLine(3), hLine(2)/hLine(3), 1];

% test if the line is correct
%tmp = vPts(10, :) * (vLine');
%tmp = hPts(1, :) * (hLine');
global x_c y_c;
[y_c, x_c] = size(single_fig);
x_c = x_c / 2;
y_c = y_c / 2;
img_center = [x_c, y_c, 1];

center2V = [];
center2H = [];
```

```
% compute lines from center to points on vertical or horizontal line
for i = 1 : vPtsCount
    tmpLine = cross(img_center, vPts(i, :));
    % draw line between img center and vertical points
    %line([img_center(1), vPts(i, 1)], [img_center(2), vPts(i, 2)]);
    % normalize
    tmpLine = [tmpLine(1)/tmpLine(3), tmpLine(2)/tmpLine(3), 1];
    center2V = [center2V; tmpLine];
end
for i = 1 : hPtsCount
    tmpLine = cross(img_center, hPts(i, :));
    % draw line between img center and horizontal points
    %line([img_center(1), hPts(i, 1)], [img_center(2), hPts(i, 2)]);
    % normalie
    tmpLine = [tmpLine(1)/tmpLine(3), tmpLine(2)/tmpLine(3), 1];
    center2H = [center2H; tmpLine];
end
% compute interceptions
v_inter = [];
h_inter = [];
for i = 1 : vPtsCount
    tmpPt = cross(center2V(i, :), vLine(1, :));
    tmpPt = [tmpPt(1)/tmpPt(3), tmpPt(2)/tmpPt(3), 1];
    v_inter = [v_inter; tmpPt];
end
for i = 1 : hPtsCount
    tmpPt = cross(center2H(i, :), hLine(1, :));
    tmpPt = [tmpPt(1)/tmpPt(3), tmpPt(2)/tmpPt(3), 1];
    h_inter = [h_inter; tmpPt];
end
% construct matrix containing all measured point x-y coordinates, from
% vertical to horizontal
global measuredPts;
measuredPts = [];
% geometric distance from measurements to img center
global dist2center;
dist2center = [];
for i = 1 : vPtsCount
    measuredPts = [measuredPts; vPts(i, 1:2)];
    tmpDist = sqrt((vPts(i, 1) - x_c)^2 + (vPts(i, 2) - y_c)^2);
    dist2center = [dist2center; tmpDist];
end
for i = 1 : hPtsCount
    measuredPts = [measuredPts; hPts(i, 1:2)];
    tmpDist = sqrt((hPts(i, 1) - x_c)^2 + (hPts(i, 2) - y_c)^2);
    dist2center = [dist2center; tmpDist];
end
% construct matrix containing all interception correspondence
global interPts;
interPts = [];
for i = 1 : vPtsCount
    interPts = [interPts; v_inter(i, 1:2)];
end
for i = 1 : hPtsCount
    interPts = [interPts; h_inter(i, 1:2)];
end
```

```
global err;
err = 0;
k = [0.05, 0.025, 0.005];
dev_init = err_func(k);

options = optimset('Algorithm', 'levenberg-marquardt', 'Tolfun', 1e-8);
kfinal = lsqnonlin(@err_func, k, [], [], options);
dev_final = err_func(kfinal);

% lens distortion correction
undistorted_img = undistortimage(single_fig, 1, x_c, y_c, kfinal(1), kfinal(2),
    kfinal(3), 0, 0, 0);
figure(2); imshow(undistorted_img); hold on;

kfinal = [kfinal 0];
undisPts = undistortPts(measuredPts, x_c, y_c, kfinal);
```

B. Corner Function

```
% use corner function to get corner points, do homogenization
function corners = get_corners(fig)
    corners = corner(fig);
    [corner_m corner_n] = size(corners);
    corners(:, 3:3) = ones(corner_m, 1);
end
```

C. Error Function for Distorted Image

```
% function for computing geometric distance between pts after removing
% distortion and those in actual measurements
% to do this function, make sure:
% 1. ptsCount is total points we have
% 2. dist2center contains the radius from image center for all pts_count points
% 3. measuredPts contains all measured coordinates of pts_count points
% 4. interPts the interception points of two lines we calculated by cross
% product
% 5. (x_c, y_c): center coordinates for img
function [deviation] = err_func(k)
    global x_c y_c measuredPts interPts dist2center vPtsCount hPtsCount err;
    ptsCount = vPtsCount + hPtsCount;
    deviation = zeros(ptsCount, 1);
    err = 0;
    for i = 1: ptsCount
        L = 1 + k(1) * dist2center(i) + k(2) * dist2center(i)^2 + k(3) * dist2center(
            i)^3;% + k(4) * dist2center(i)^4;
        x_hat = x_c + L * (measuredPts(i, 1) - x_c);
        y_hat = y_c + L * (measuredPts(i, 2) - y_c);

        x_err = x_hat - interPts(i, 1);
        y_err = y_hat - interPts(i, 2);

        err = err + sqrt(x_err^2 + y_err^2);
        deviation(2*i - 1) = x_err;
        deviation(2*i) = y_err;
    end
end
```

D. Undistort Points using K

```
% input params:
% @x: the param that carries all measured coordinates of corners in figure
% @k: the param for k values
function result = undistortPts(x, x_c, y_c, k)
    result = [];
    [m, n] = size(x);
    for i = 1 : m
        x_0 = x(i, 1);
        y_0 = x(i, 2);
        r = sqrt((x_0 - x_c)^2 + (y_0 - y_c)^2);
        L = 1 + k(1) * r + k(2) * r^2 + k(3) * r^3 + k(4) * r^4;
        x_new = x_c + L * (x_0 - x_c);
        y_new = y_c + L * (y_0 - y_c);
        result = [result; [x_new, y_new, 1]];
    end
end
```

E. Undistort Image using K

```
% UNDISTORTIMAGE - Removes lens distortion from an image
%
% Usage:  nim = undistortimage(im, f, ppx, ppy, k1, k2, k3, p1, p2)
%
% Arguments:
%          im - Image to be corrected.
%           f - Focal length in terms of pixel units
%               (focal_length_mm/pixel_size_mm)
%     ppx, ppy - Principal point location in pixels.
%   k1, k2, k3 - Radial lens distortion parameters.
%       p1, p2 - Tangential lens distortion parameters.
%
% Returns:
%         nim - Corrected image.
%
%  It is assumed that radial and tangential distortion parameters are
%  computed/defined with respect to normalised image coordinates corresponding
%  to an image plane 1 unit from the projection centre.  This is why the
%  focal length is required.

% Copyright (c) 2010 Peter Kovesi
% Centre for Exploration Targeting
% The University of Western Australia
% peter.kovesi at uwa edu au
%
% Permission is hereby granted, free of charge, to any person obtaining a copy
% of this software and associated documentation files (the "Software"), to deal
% in the Software without restriction, subject to the following conditions:
%
% The above copyright notice and this permission notice shall be included in
% all copies or substantial portions of the Software.
%
% The Software is provided "as is", without warranty of any kind.

% October   2010  Original version
% November  2010  Bilinear interpolation + corrections
% April     2015  Cleaned up and speeded up via use of interp2
% September 2015  Incorporated k3 + tangential distortion parameters
```

```
function nim = undistortimage(im, f, ppx, ppy, k1, k2, k3, k4, p1, p2)

    % Strategy: Generate a grid of coordinate values corresponding to an ideal
    % undistorted image.  We then apply the imaging process to these
    % coordinates, including lens distortion, to obtain the actual distorted
    % image locations.  In this process these distorted image coordinates end up
    % being stored in a matrix that is indexed via the original ideal,
    % undistorted coords.  Thus for every undistorted pixel location we can
    % determine the location in the distorted image that we should map the grey
    % value from.

    % Start off generating a grid of ideal values in the undistorted image.
    [rows,cols,chan] = size(im);
    [xu,yu] = meshgrid(1:cols, 1:rows);

    % Convert grid values to normalised values with the origin at the principal
    % point.  Dividing pixel coordinates by the focal length (defined in pixels)
    % gives us normalised coords corresponding to z = 1
    x = (xu-ppx)/f;
    y = (yu-ppy)/f;

    % Radial lens distortion component
    r2 = sqrt(x.^2+y.^2);                    % Squared normalized radius.
    dr = k1*r2 + k2*r2.^2 + k3*r2.^3 + k4*r2.^4;  % Distortion scaling factor.

    % Tangential distortion component (Beware of different p1,p2
    % orderings used in the literature)
    dtx =    2*p1*x.*y      +  p2*(r2 + 2*x.^2);
    dty = p1*(r2 + 2*y.^2)  +    2*p2*x.*y;

    % Apply the radial and tangential distortion components to x and y
    x = x + dr.*x + dtx;
    y = y + dr.*y + dty;

    % Now rescale by f and add the principal point back to get distorted x
    % and y coordinates
    xd = x*f + ppx;
    yd = y*f + ppy;

    % Interpolate values from distorted image to their ideal locations
    if ndims(im) == 2   % Greyscale
        nim = interp2(xu,yu,double(im),xd,yd);
    else % Colour
        nim = zeros(size(im));
        for n = 1:chan
            nim(:,:,n) = interp2(xu,yu,double(im(:,:,n)),xd,yd);
        end
    end

    if isa(im, 'uint8')      % Cast back to uint8 if needed
        nim = uint8(nim);
    end
end
```

F. Vec2mat Function

```
function [ mat ] = vec2mat( vec, N )
    %VEC2MAT builds res-matrix from res-vector
    %   mat:  ceil(M) columns, N rows
    %   vec:  has to be row vector
    if ~isrow(vec)
        warning('Vektoren sollten ZeilenVektoren (1 x n) sein');
        vec=vec';
    end

    segs=length(vec);
    M=segs/N;
    Mc=ceil(M);
    vec=[vec nan(  1, Mc*N-segs)];

    mat=reshape(vec,N,Mc).';
end
```

G. Geometric Distance from World to Image

```
% compute the geometric distance between points we got on applying P for
% normalized world frame and those directly from normalization
function [ F ] = geoDist(p,X)
    P = [p(1) p(2) p(3) p(4);
        p(5) p(6) p(7) p(8);
        p(9) p(10) p(11) p(12)]; % Intrinsic parameter
    [M,N] = size(X);
    F = zeros(M,N-1);
    for k = 1:M
        Y = P * X(k,:)';
        F(k,1) = Y(1)/Y(3);
        F(k,2) = Y(2)/Y(3);
        F(k,3) = 1;
    end
end
```

H. Camera Calibration

```
clear all; close all; clc;
two_img = imread('pics/two_new.jpg');
two_img = rgb2gray(two_img);
figure(1); imshow(two_img);
hold on;

% get all corner points in the single checker board figure
raw_corners = get_corners(two_img);

% draw all corner points on the figure
%h = plot(raw_corners(:, 1), raw_corners(:, 2), 'x', 'Color', 'r', 'MarkerSize', 15);
%set(h,'linewidth',3);

% corners sorted by x, 1st row
% sort by first coordinate (x) we get points by vertical lines
rawVCorners = sortrows(raw_corners, 1);
%hCorners = sortrows(raw_corners, 2);

% count of all corner pts on the image
corners_size = size(raw_corners);
```

```
ptsCount = corners_size(1);
% count of pts per vertical/horizontal line
vLinesCount = 10;
hLinesCount = 16;

sortedVCorners = [];
% sort vertical points top-down (ascending y)
for i = 1 : hLinesCount
    sIndex = (i - 1) * vLinesCount + 1;
    endIndex = i * vLinesCount;
    tmpVPtsSet = rawVCorners(sIndex:endIndex, :);
    tmpVPtsSet = sortrows(tmpVPtsSet, 2);
    sortedVCorners = [sortedVCorners; tmpVPtsSet];
end
% get the first two row (2d coordinate for img frame)
%imgPts = sortedVCorners(:, 1:3);
% remove lens distortion
% legacy from lens distortion
k = [7.35094247828436e-05,-4.63575177452332e-07,7.15015120029531e-10,0];
[y_c, x_c] = size(two_img);
x_c = x_c / 2;
y_c = y_c / 2;
imgPts = undistortPts(sortedVCorners, x_c, y_c, k);

% test the sequence of rawVCorners(partially sorted) points
% for i = 1 : ptsCount
%     h = plot(rawVCorners(i, 1), rawVCorners(i, 2), 'x', 'Color', 'r', 'MarkerSize',
    6);
%     set(h, 'linewidth', 3);
% end
% test the sequence of sortedVCorners(totally sorted) points
for i = 1 : ptsCount
    h = plot(sortedVCorners(i, 1), sortedVCorners(i, 2), 'x', 'Color', 'r', '
        MarkerSize', 6);
    set(h, 'linewidth', 3);
end

% compute the world coordinate based on actual measurements
xzPts = [];
yzPts = [];

% processing xz plane
% first determine the coordinate of a reference point
leftTop1 = [192.5, 0, 220.5];
% based on the same sequence of our corner seeking function, determine the
% corresponding world coordinates of those corners
for i = 1 : (hLinesCount / 2)
    tmpX = leftTop1(1) - (i - 1) * 24.5;
    for j = 1 : (vLinesCount)
        tmpZ = leftTop1(3) - (j - 1) * 24.5;
        xzPts = [xzPts; [tmpX, 0, tmpZ, 1]];
    end
end

% processing yz plane
leftTop2 = [0, 23.5, 220.5];
```

```
for i = 1 : (hLinesCount / 2)
    tmpY = leftTop2(2) + (i - 1) * 24.5;
    for j = 1 : (vLinesCount)
        tmpZ = leftTop1(3) - (j - 1) * 24.5;
        yzPts = [yzPts; [0, tmpY, tmpZ, 1]];
    end
end
% integrate xz and yz plane points into the same mat
worldPts = [xzPts; yzPts];

% do normalization for both world frame and image frame
[worldHomo, worldNorm] = normalize(worldPts, 3);
[imgHomo, imgNorm] = normalize(imgPts, 2);

% do DLT
% compute A matrix, ptsCount*2 rows and 3*4 cols
A = zeros(ptsCount * 2, 12);
for i = 1 : ptsCount
   A(2*i-1, :) = [zeros(1, 4) -worldNorm(i, :) imgNorm(i, 2)*worldNorm(i, :)];
   A(2*i, :) = [worldNorm(i, :) zeros(1, 4) -imgNorm(i, 1)*worldNorm(i, :)];
end
% do singular value decomposition
[U, D, V] = svd(A);
% basically aliase V as p
p = V(:, end);

% compute errors before applying LM minimization
tmpP = [p(1) p(2) p(3) p(4); p(5) p(6) p(7) p(8); p(9) p(10) p(11) p(12)];
tmpP = imgHomo\(tmpP * worldHomo);
lastSumSqrt = sqrt(sum(tmpP(3,1:3).^2));
tmpP = tmpP / lastSumSqrt;
xcenter = worldPts * tmpP';
xnew = xcenter(:,:) ./ xcenter(:,3);
error_init = sum(sum((imgPts - xnew).^2))/320;
fprintf('Error before LM is: %d\n',error_init);

% LM
% options for using LM
opt = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt');
[q,res] = lsqcurvefit(@geoDist,p,worldNorm,imgNorm,[],[],opt); % Refined parameter

P = vec2mat(q,4);
P = imgHomo \ P * worldHomo;
lastSumSqrt = sqrt(sum(P(3,1:3).^2));
P = P / lastSumSqrt;
% check error after LM
xcupdate = worldPts * P';
xnew = xcupdate(:,:)./xcupdate(:,3);

error_lm = sum(sum((imgPts - xnew).^2))/320;
fprintf('Errors after LM: %d\n', error_lm);

% final result
[Q,R] = qr(P(1:3,1:3)^(-1));
K = R^(-1);
K = K / K(3,3);
```

```
C = -P(1:3,1:3)\P(:,4);

% for testing the sequence of world frame point generation
% figure(2);
% for i = 1 : (ptsCount / 2)
%       scatter3(xzPts(i, 1), xzPts(i, 2), xzPts(i, 3));
%       hold on;
% end
% for i = 1 : (ptsCount / 2)
%       scatter3(yzPts(i, 1), yzPts(i, 2), yzPts(i, 3));
%       hold on;
% end
% scatter3(xzPts(:, 1), xzPts(:, 2), xzPts(:, 3));
% hold on;
%scatter3(yzPts(:, 1), yzPts(:, 2), yzPts(:, 3));
%hold on;
```

I. Normalization

```
% normalization for 2d and 3d space
function [homo,result] = normalize(A,ndim)
    if (ndim == 2)
        nordis = sqrt(2);
    elseif (ndim == 3)
        nordis = sqrt(3);
    else
        printf('we cant handle cases other than 2/3 dimensions');
    end
    [m,n] = size(A);
    % get the centroid of all points
    avg = mean(A);
    % get the sum of distances between pts and centroid
    total = sum(sqrt(sum((A - avg).^2,2)));
    % calculate diagonal element s in similarity homography
    s = m*nordis/total;
    lc = [-s*avg(1) -s*avg(2) 1];
    if (ndim == 3)
        lc = [-s*avg(1) -s*avg(2) -s*avg(3) 1];
    end
    homo = s*eye(n);
    homo(:,n) = lc';
    result = A*homo';
end
```