EE 5885 - Geometry and Deep Computer Vision Project 1

Libao Jin

October 17, 2019

1 Problem Statement

This project is to calibrate a camera by using a sequence of images that are taken as the camera rotates without translating. A number of files to help with this task are provided in WyoCourses under the directory `project1`. First, it contains seven photos taken in Prexy's pasture as I rotated the camera. Second, it contains correspondence points for each adjoining set of images that are contained MATLAB using `load pureRotPrexyCorrespondencePoints`. It contains two data structures, `x1pMat` and `x2pMat`. These $3 \times 100 \times 6$ data structures contain the correspondence points between images I and $I + 1$. For instance, `x1pMat(:,:,1)` contains image 1 pixel correspondence points between images 1 and images 2, `x2pMat(:,:,1)` contains image 2 pixel correspondence points between photos 1 and 2, `x2pMat(:,:,6)` contains image 2 pixel correspondence points between image 6 and image 7, etc. Note: These correspondence points were automatically generated using the SIFT operator, which we will soon cover. MATLAB code for SIFT is available through VLFEAT. VLFEAT is not needed for this project, but is very useful. Your task is to implement and test the calibration algorithm that uses images obtained under pure rotation. The program `Proj1PureRotCalib5DsearchTemplate.m` is provided to help you load and plot the images, etc., so be sure to first load all the `project1` files into a single directory for you to work on., run this template, and look at its comments. It has several hints. A similar Python file, `proj1template.py` is also provided if you prefer to use Python. To do the optimization, you may find `fminsearch.m` useful in MATLAB, or `scipy.optimize.minimize` useful in Python.

1. Using the data provided, calibrate the camera. Make sure you use all six pairs of images to find a single calibration matrix that is common to all of them.

2. Estimate the rotation matrix between camera positions 2 and 3.

3. Evaluate how well your algorithm performs by plotting the reprojections of the data for images 2 and 3 along with the actual data. Comment on and try to explain any good or bad performance.

4. Extensively test your algorithm. You may want to produce simulated points with varying noise levels, use other correspondence points in the images, calibrate your own camera, etc.

5. Throughly document your algorithm, its performance, and the results of your tests.

High grades are given for keen observations, creative algorithm improvements, clear documentation, etc. Show me that you understand how to implement the algorithm, what its strengths and weaknesses are, ways to mitigate any problems, etc. A formal written report documenting the above is expected and will be graded for grammar and clearness of exposition.

2 Method

2.1 Finding Rotation Matrix R Using Optimal Quaternion Algorithm

Assume that the calibration matrices K_1 and K_2 are known, and suppose that $\mathbf{X}_1$ and $\mathbf{X}_2$ are the coordinates of the same point in camera frame 1 and camera frame 2, respectively. With the help of the calibration matrices, we can get the pixel coordinates of correspondence points $\mathbf{x}_1$, $\mathbf{x}_2$ as follows,

$$\lambda_1\mathbf{x}_1 = K_1\mathbf{X}_1, \lambda_2\mathbf{x}_2 = K_2\mathbf{X}_2.$$

It is known that

$$\mathbf{X}_2 = R_{2,1}\mathbf{X}_1 \implies \lambda_2 K_2^{-1}\mathbf{x}_2 = \lambda_1 R_{2,1}K_1^{-1}\mathbf{x}_1. \tag{2.1}$$

where $R_{2,1}$ is the rotation matrix from camera frame 1 to camera frame 2. Rotation preserves the magnitude of the vector, hence we can get

$$\frac{\lambda_2 K_2^{-1}\mathbf{x}_2}{\|\lambda_2 K_2^{-1}\mathbf{x}_2\|} = R_{2,1}\frac{\lambda_1 K_1^{-1}\mathbf{x}_1}{\|\lambda_1 K_1^{-1}\mathbf{x}_1\|} \implies \frac{K_2^{-1}\mathbf{x}_2}{\|K_2^{-1}\mathbf{x}_2\|} = R_{2,1}\frac{K_1^{-1}\mathbf{x}_1}{\|K_1^{-1}\mathbf{x}_1\|} \implies \mathbf{u}_2 = R_{2,1}\mathbf{u}_1, \tag{2.2}$$

where $\mathbf{u}_j = K_j^{-1}\mathbf{x}_j/\|K_j^{-1}\mathbf{x}_j\|, j = 1, 2$. We can rewrite (2.2) as follows,

$$\mathbf{u}_2 = R_{2,1}\mathbf{u}_1 \implies \mathbf{u}_2 - R_{2,1}\mathbf{u}_1 = 0.$$

In general, given m correspondence points, say $\mathbf{x}_1^i, \mathbf{x}_2^i, i = 1, 2, \ldots, m$, assume that the calibration matrices $K_1 = K_2 = K$, we want use these points to find a rotation matrix $\widetilde{R}_{2,1}$ such that the sum of the norm of the residuals is minimized,

$$\sum_{i=1}^{m}\|\mathbf{u}_2^i - \widetilde{R}_{2,1}\mathbf{u}_1^i\|_2^2 = \min_{R_{2,1}\in\mathbb{R}^{3\times3}}\sum_{i=1}^{m}\|\mathbf{u}_2^i - R_{2,1}\mathbf{u}_1^i\|_2^2 = \min_{R_{2,1}\in\mathbb{R}^{3\times3}}\sum_{i=1}^{m}\left\|\frac{K^{-1}\mathbf{x}_2^i}{\|K^{-1}\mathbf{x}_2^i\|} - R_{2,1}\frac{K^{-1}\mathbf{x}_1^i}{\|K^{-1}\mathbf{x}_1^i\|}\right\|_2^2.$$

It turns out that solving this optimization problem is equivalent to solving an eigenvalue problem, which gives us the Optimal Quaternion Algorithm (Algorithm 1).

Algorithm 1: Optimal Quaternion Algorithm

Function OptimalQuaternion($\{\mathbf{x}_1^i\}_{i=1}^m, \{\mathbf{x}_2^i\}_{i=1}^m, K$)**:**

Input: Pixel coordinates of m correspondence points $\{\mathbf{x}_1^i\}_{i=1}^m$, $\{\mathbf{x}_2^i\}_{i=1}^m$, and calibration matrix K.

Output: The rotation matrix $\widetilde{R} = \widetilde{R}_{2,1}$ from Image 1 to Image 2.

1. $\mathbf{u}_1^i = K^{-1}\mathbf{x}_1^i/\|K^{-1}\mathbf{x}_1^i\|$ and $\mathbf{u}_2^i = K^{-1}\mathbf{x}_2^i/\|K^{-1}\mathbf{x}_2^i\|$ for $i = 1, 2, \ldots, m$.
2. $D^i = \begin{bmatrix} 0 & (\widehat{\mathbf{u}}_1^i\mathbf{u}_2^i)^T \\ \widehat{\mathbf{u}}_1^i\mathbf{u}_2^i & \mathbf{u}_1^i\mathbf{u}_2^{iT} + \mathbf{u}_2^i\mathbf{u}_1^{iT} - 2\mathbf{u}_1^{iT}\mathbf{u}_2^i I \end{bmatrix}$.
3. $D = \sum_{i=1}^{m} D^i$.
4. Find the eigenvector $q = \begin{bmatrix} q_0 & \mathbf{q} \end{bmatrix}^T$ of D which corresponds to the largest eigenvalue.
5. $\widetilde{R} = I + 2q_0\widehat{\mathbf{q}} + 2(\mathbf{q}\mathbf{q}^T - \mathbf{q}^T\mathbf{q}I)$.

end

2.2 Finding Calibration Matrix K By Solving A Optimization Problem

Next, let us find the calibration matrix K when R is given. Rewriting (2.1) gives

$$\lambda_2 K_2^{-1}\mathbf{x}_2 = \lambda_1 R_{2,1} K_1^{-1}\mathbf{x}_1 \implies \lambda_2\mathbf{x}_2 = \lambda_1 K_2 R_{2,1} K_1^{-1}\mathbf{x}_1.$$

Suppose the approximated rotation matrix $\widetilde{R}_{2,1}$ is found using Algorithm 1, then we have

$$\lambda_2\mathbf{x}_2 = \lambda_1 K_2 R_{2,1} K_1^{-1}\mathbf{x}_1 \approx \lambda_1 K_2 \widetilde{R}_{2,1} K_1^{-1}\mathbf{x}_1 \implies \mathbf{x}_2 \approx \frac{\lambda_1}{\lambda_2} K_2 \widetilde{R}_{2,1} K_1^{-1}\mathbf{x}_1.$$

Note that in the provided photos, the depths of correspondence points are almost the same, then we can use the weak orthographic projection to approximate the perspective projection. Therefore, the scaling factor λ_1/λ_2 is a constant which can be combined into the calibration matrix K_1 and K_2, we can obtain

$$\mathbf{x}_2 = K_2 R_{2,1} K_1^{-1}\mathbf{x}_1 \approx K_2 \widetilde{R}_{2,1} K_1^{-1}\mathbf{x}_1 = \widetilde{\mathbf{x}}_2,$$

where $\widetilde{\mathbf{x}}_2 = K_2 \widetilde{R}_{2,1} K_1^{-1}\mathbf{x}_1$ is the reprojection of $\mathbf{x}_1$ from Image 1 to Image 2. Now given m correspondence points $\mathbf{x}_1^i, \mathbf{x}_2^i, i = 1, 2, \ldots, m$, and assume that $K_1 = K_2 = K$, we want to find the calibration matrix $\widetilde{K}$ so as to minimize the sum of the norms of reprojection residuals from Image 1 to Image 2 $J_{2,1}$ and that from Image 2 to Image 1 $J_{1,2}$.

$$\begin{aligned}
\min J_{1,2} + J_{2,1} &= \min \sum_{i=1}^{m} \|\mathbf{x}_1^i - \widetilde{\mathbf{x}}_1^i\| + \sum_{i=1}^{m} \|\mathbf{x}_2^i - \widetilde{\mathbf{x}}_2^i\| \\
&= \min_{K\in\mathbb{R}^{3\times 3}} \sum_{i=1}^{m} (\|\mathbf{x}_1^i - K\widetilde{R}_{1,2}K^{-1}\mathbf{x}_2^i\| + \|\mathbf{x}_2^i - K\widetilde{R}_{2,1}K^{-1}\mathbf{x}_1^i\|) \\
&= \min_{K\in\mathbb{R}^{3\times 3}} \sum_{i=1}^{m} (\|\mathbf{x}_1^i - K\widetilde{R}_{2,1}^T K^{-1}\mathbf{x}_2^i\| + \|\mathbf{x}_2^i - K\widetilde{R}_{2,1}K^{-1}\mathbf{x}_1^i\|) \\
&= \sum_{i=1}^{m} (\|\mathbf{x}_1^i - \widetilde{K}\widetilde{R}_{2,1}^T\widetilde{K}^{-1}\mathbf{x}_2^i\| + \|\mathbf{x}_2^i - \widetilde{K}\widetilde{R}_{2,1}\widetilde{K}^{-1}\mathbf{x}_1^i\|).
\end{aligned}$$

Observe that the calibration matrix K has the form

$$K = \begin{bmatrix} fs_x & fs_\theta & o_x \\ 0 & fs_y & o_y \\ 0 & 0 & 1 \end{bmatrix}.$$

That is to say, K can be uniquely determined by five unknowns. Let $\mathbf{k} = \begin{bmatrix} k_1 & k_2 & k_3 & k_4 & k_5 \end{bmatrix}^T$, we can use $\mathbf{k}$ to construct the calibration matrix K and the inverse of that K^{-1} as follows,

$$K = \begin{bmatrix} k_1 & k_2 & k_3 \\ 0 & k_4 & k_5 \\ 0 & 0 & 1 \end{bmatrix}, K^{-1} = \begin{bmatrix} \frac{1}{k_1} & -\frac{k_2}{k_1 k_4} & \frac{k_2 k_5}{k_1 k_4} - \frac{k_3}{k_1} \\ 0 & \frac{1}{k_4} & -\frac{k_5}{k_4} \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.3}$$

Therefore, $J_{1,2} + J_{2,1} = J_{1,2}(\mathbf{k}) + J_{2,1}(\mathbf{k})$, we can solve for $\widetilde{\mathbf{k}}$ using some numerical schemes such that

$$\widetilde{\mathbf{k}} = \underset{\mathbf{k}\in\mathbb{R}^5}{\operatorname{argmin}}\, J_{1,2}(\mathbf{k}) + J_{2,1}(\mathbf{k}) = \{\widetilde{\mathbf{k}} | J_{1,2}(\widetilde{\mathbf{k}}) + J_{2,1}(\widetilde{\mathbf{k}}) = \min_{\mathbf{k}\in\mathbb{R}^5} J_{1,2}(\mathbf{k}) + J_{2,1}(\mathbf{k})\}.$$

Once such $\mathbf{k}$ is found, we can construct K and K^{-1} by (2.3). Furthermore, we can use the same principle to find a common calibration matrix K to multiple pairs of images. Suppose that $\mathbf{x}_j^i, \mathbf{x}_{j+1}^i, i = 1, 2, \ldots, m_j, j = 1, 2, \ldots, n$ are the m_j pairs correspondence points in Image j and Image $j+1$, $R_{j+1,j}$ is the rotation matrices from Image j to Image $j+1$, then our objective function is the sum of norms of reprojection residuals of all the images:

$$J(\mathbf{k}) = \sum_{j=1}^{n} J_{j,j+1}(\mathbf{k}) + J_{j+1,j}(\mathbf{k}) = \sum_{j=1}^{n}\sum_{i=1}^{m_j}(\|\mathbf{x}_j^i - KR_{j+1,j}^T K^{-1}\mathbf{x}_{j+1}^i\| + \|\mathbf{x}_{j+1}^i - KR_{j+1,j}K^{-1}\mathbf{x}_j^i\|).$$

Then we solve the minimization problem to obtain $\widetilde{\mathbf{k}}$ as below

$$\widetilde{\mathbf{k}} = \underset{\mathbf{k}\in\mathbb{R}^5}{\operatorname{argmin}}\, J(\mathbf{k}) = \{\widetilde{\mathbf{k}} | J(\widetilde{\mathbf{k}}) = \min_{\mathbf{k}\in\mathbb{R}^5} J(\mathbf{k})\}.$$

2.3 Finding Rotation Matrix R and Calibration Matrix K

In this project, the methods in the previous two subsections are not directly applicable, because either the calibration matrix K or the rotation matrices $R_{j+1,j}, j = 1, 2, \ldots, n$ are not given. In fact, while solving for the $\mathbf{k}$ in the optimization problem, we use some numerical schemes which requires an initial guess. Then the solution we get might be a local minimizer instead of a global minimizer. Hence we need to adjust our initial guess to get a better solution. Based on this, and due to the unknown specification of the camera, we can only pick a random guess for $\mathbf{k}$. Then construct K and K^{-1} accordingly, and then use the Optimal Quaternion Algorithm to find the rotation matrix R, which is likely to be far from the actual rotation matrix. But we can now find $\mathbf{k}$ such that the J calculated using R is minimized, again, we can use the obtained $\mathbf{k}$ to adjust R. Hence, we have the Algorithm 2.

Algorithm 2:

Function Calibrate($\{\mathbf{x}_j^i\}_{i=1}^{m_j}, tol$)**:**

Input: $\{\mathbf{x}_j^i\}_{i=1}^{m_j}, j = 1, 2, \ldots, n$ are the pixel coordinates of m_j correspondence points, and tol is the tolerance of the J.

Output: Rotation matrices $R_j, j = 1, 2, \ldots, n$, and calibration matrix K.

$\mathbf{k} = \text{rand}(5, 1)$

$$K = \begin{bmatrix} k_1 & k_2 & k_3 \\ 0 & k_4 & k_5 \\ 0 & 0 & 1 \end{bmatrix}, K^{-1} = \begin{bmatrix} \frac{1}{k_1} & -\frac{k_2}{k_1 k_4} & \frac{k_2 k_5}{k_1 k_4} - \frac{k_3}{k_1} \\ 0 & \frac{1}{k_4} & -\frac{k_5}{k_4} \\ 0 & 0 & 1 \end{bmatrix}$$

$R_j = \text{OptimalQuaternion}(\{\mathbf{x}_j^i\}_{i=1}^{m_j}, \{\mathbf{x}_{j+1}^i\}_{i=1}^{m_{j+1}}, K), j = 1, 2, \ldots, n$

$\mathbf{k} = \text{argmin}_{\mathbf{k}}\, J(\mathbf{k})$

while $J(\mathbf{k}) > tol$ **do**

$$K = \begin{bmatrix} k_1 & k_2 & k_3 \\ 0 & k_4 & k_5 \\ 0 & 0 & 1 \end{bmatrix}, K^{-1} = \begin{bmatrix} \frac{1}{k_1} & -\frac{k_2}{k_1 k_4} & \frac{k_2 k_5}{k_1 k_4} - \frac{k_3}{k_1} \\ 0 & \frac{1}{k_4} & -\frac{k_5}{k_4} \\ 0 & 0 & 1 \end{bmatrix}$$

$R_j = \text{OptimalQuaternion}(\{\mathbf{x}_j^i\}_{i=1}^{m_j}, \{\mathbf{x}_{j+1}^i\}_{i=1}^{m_{j+1}}, K), j = 1, 2, \ldots, n$

$\mathbf{k} = \text{argmin}_{\mathbf{k}}\, J(\mathbf{k})$

end

end

2.4 Simulation to Validate the Effectiveness of the Algorithms

Due to the lack of exact rotation matrices and calibration matrix, it is hard to directly evaluate the performance of Algorithm 1 and Algorithm 2. Hence some simulation points are generated with noises of different levels using some fixed rotation matrix R and calibration matrix K.

3 Implementation

The above algorithms (Algorithm 1 and Algorithm 2) in Python (the program is attached at the end). In Algorithm 2, we need to determine the stopping criterion *tol*. However, the number of correspondence points varies from images to images. Even if all images have the same number of correspondence points, it is still hard to determine the stopping criterion: for a different set of images which have more correspondence points, we need to change the stopping criterion accordingly. To make the algorithm more applicable with a consistent criterion, we modify J as follows,

$$J = \frac{1}{n}\sum_{j=1}^{n}\frac{J_{j+1,j} + J_{j,j+1}}{m_j},$$

where m_j is the number of correspondence points between image j and $j+1$. Then J is the average of the norm of the residuals of all points in each image.
We define a class called `CameraCalibration`, which includes methods to complete each task in the algorithms. Here is a list of all the methods:

- `hat(v)`: works as the hat operator ($\widehat{\ }$), converting a vector to a skew-symmetric matrix.
- `D(u1, u2)`: constructs the matrix D needed in Algorithm 1.
- `eigvec(A)`: finds the eigenvector which corresponds the largest eigenvalue of matrix A.
- `q2R(q4)`: converts a quaternion to the rotation matrix R.
- `optimal_quaternion(x1, x2, K)`: the implementation of Algorithm 1 with calibrated points.
- `K(k)`: constructs calibration matrix K from the vector $\mathbf{k}$.
- `invert_K(k)`: constructs the inverse of the calibration matrix K^{-1} from the vector $\mathbf{k}$.
- `find_R_list_pair(x1_list, x2_list, tol, max_iter)`: finds the rotation matrices for all the images with their own calibration matrix.
- `calibrate(x1_list, x2_list, tol, max_iter, output_dir)`: finds the common calibration matrix of all images.
- `read_data(filename)`: reads MATLAB `.mat` file into Python.
- `write_csv(R_list, K, output_dir)`: saves rotation matrices and calibration matrix to a `.csv` file.
- `read_csv(output_dir)`: reads stored calibration matrix and rotation matrices.
- `disp(A, name, filename)`: formats the output of the calibration matrix and rotation matrices to LaTeX.

- `print_matrix(output_dir)`: saves the rotation matrix $R_j, j = 1, 2, \ldots, 6$ and calibration matrix K to `.tex` files.
- `show_iamge(I, x, xr, output_dir, name)`: displays the image `I` with correspondence points and their reprojections using `matplotlib.image` and `matplotlib.pyplot`.
- `reproject(x, R, K)`: reprojects one correspondence points to another image.
- `solve(mat_file, images, output_dir, tol, max_iter)`: completes tasks in the project and use `matplotlib.image` and `matplotlib.pyplot` for visualization (annotation).
- `solve_cv2(mat_file, images, output_dir, tol, max_iter)`: completes tasks in the project and use OpenCV Python module `cv2` for visualization (annotation).
- `test_optimal_quaternion(output_dir)`: tests the Optimal Quaternion Algorithm using simulation points.

4 Result and Discussion

1. The single calibration matrix that is common to all of the six pairs of images is

$$K = \begin{bmatrix} 9.3247 & -0.8647 & -160.3975 \\ 0.0000 & 2.5373 & 0.0664 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}.$$

2. The rotation matrix between camera position 2 and 3 is

$$\widetilde{R}_2 = \begin{bmatrix} 0.9986 & -0.0520 & -0.0006 \\ 0.0520 & 0.9986 & 0.0007 \\ 0.0005 & -0.0007 & 1.0000 \end{bmatrix}.$$

3. The reprojection of images 2 and 3 is shown in Figure 1. As we can see, the reprojections almost overlap with the actual points. That is to say, Algorithm 2 we proposed is valid to find the approximation of the calibration matrix K and rotation matrix $R_{3,2}$. The reason why our algorithm is effective might be that the Image 2 and Image 3 were taken with the same camera calibration and it is indeed pure rotation between camera position 2 and camera position 3.

4. We also used other correspondence points in the images. As shown in Figure 2 and 4, we can see that Algorithm 2 is satisfactory: the reprojections of the points coincide with the actual points with very small error. When it comes to image 1 and 2 (Figure 5) or image 6 and 7 (Figure 4), the reprojections of the correspondence points are slightly off. However, Algorithm 2 failed to reproject points to its correspondence points between Image 4 and 5 (Figure **??**). The possible causes of the above result are that

   (a) When all these photos were taken, different camera calibrations were used.

   (b) The rotations between some camera positions may not be pure rotations.

Also, the Algorithm 2 may be more robust when the correspondence points are more spread out than those which are not. Here are the rotation matrices:

$$\widetilde{R}_1 = \begin{bmatrix} 0.8970 & -0.4421 & -0.0000 \\ 0.4421 & 0.8970 & -0.0000 \\ 0.0000 & -0.0000 & 1.0000 \end{bmatrix}.$$

$$\widetilde{R}_3 = \begin{bmatrix} 0.9995 & -0.0315 & -0.0006 \\ 0.0315 & 0.9995 & 0.0001 \\ 0.0006 & -0.0001 & 1.0000 \end{bmatrix}.$$

$$\widetilde{R}_4 = \begin{bmatrix} 0.9989 & -0.0339 & -0.0321 \\ 0.0339 & 0.9994 & -0.0005 \\ 0.0321 & -0.0006 & 0.9995 \end{bmatrix}.$$

$$\widetilde{R}_5 = \begin{bmatrix} 0.9994 & -0.0347 & -0.0000 \\ 0.0347 & 0.9994 & 0.0000 \\ 0.0000 & -0.0000 & 1.0000 \end{bmatrix}.$$

$$\widetilde{R}_6 = \begin{bmatrix} 0.9419 & -0.3359 & -0.0000 \\ 0.3359 & 0.9419 & -0.0000 \\ 0.0000 & -0.0000 & 1.0000 \end{bmatrix}.$$

To justify the validity of Algorithm 1, we provide a simulation (see `test_optimal_quaternion`). Assume that

$$K = \begin{bmatrix} 2.0000 & 0.0000 & 1.0000 \\ 0.0000 & 2.0000 & 1.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}.$$

$$R = \begin{bmatrix} 0.5000 & 0.0000 & 0.8660 \\ 0.0000 & 1.0000 & 0.0000 \\ -0.8660 & 0.0000 & 0.5000 \end{bmatrix}.$$

Then we randomly generate 20 points $\mathbf{x}_1^i, i = 1, 2, \ldots, n$, and use K and R to get $\mathbf{x}_2^i, i = 1, 2, \ldots, n$, where $\mathbf{x}_2 = KRK^{-1}\mathbf{x}_1$. Then we call `optimal_quaternion(x1, x2, K)` and obtain R' and we calculate the error $R_{error} = R - R'$ as follows,

$$R' = \begin{bmatrix} 0.5000 & -0.0000 & 0.8660 \\ -0.0000 & 1.0000 & 0.0000 \\ -0.8660 & -0.0000 & 0.5000 \end{bmatrix}.$$

$$R_{error} = \begin{bmatrix} 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & -0.0000 \\ 0.0000 & 0.0000 & 0.0000 \end{bmatrix}.$$

As shown above, Algorithm 1 has a very good performance. And we can also check the norm of reprojection residual

$$\sum_{i=1}^{20} \|\widetilde{\mathbf{x}}_1^i - \mathbf{x}_2^i\| = 3.5789 \times 10^{-12}.$$

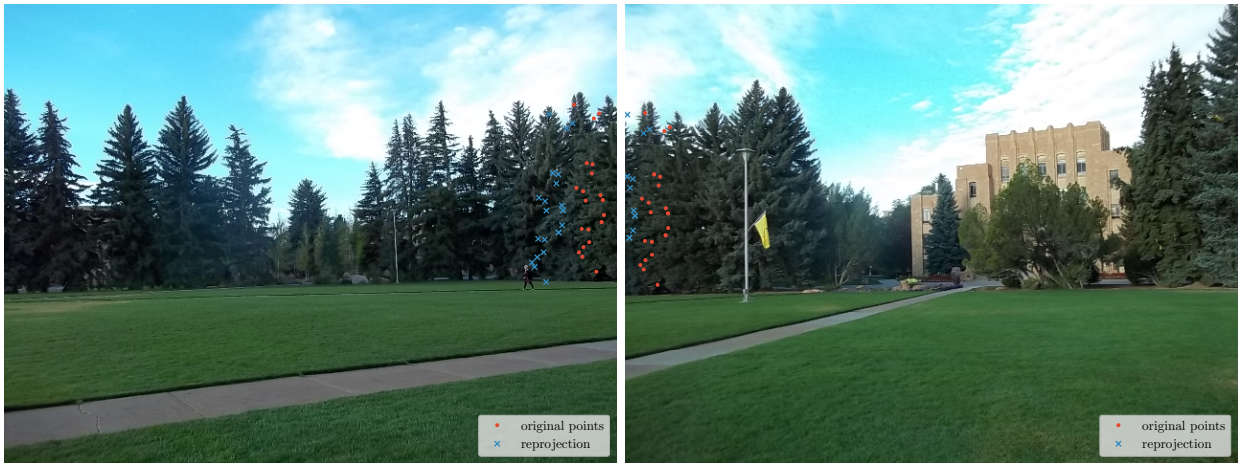

Figure 1: Reprojections of Correspondence Points Between Image 2 and Image 3

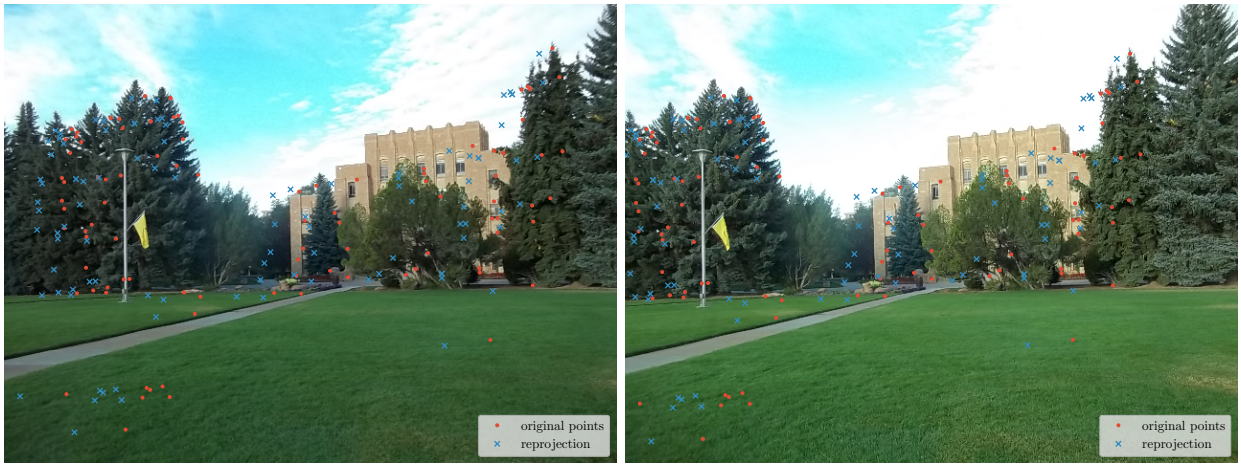

Figure 2: Reprojections of Correspondence Points Between Image 3 and Image 4

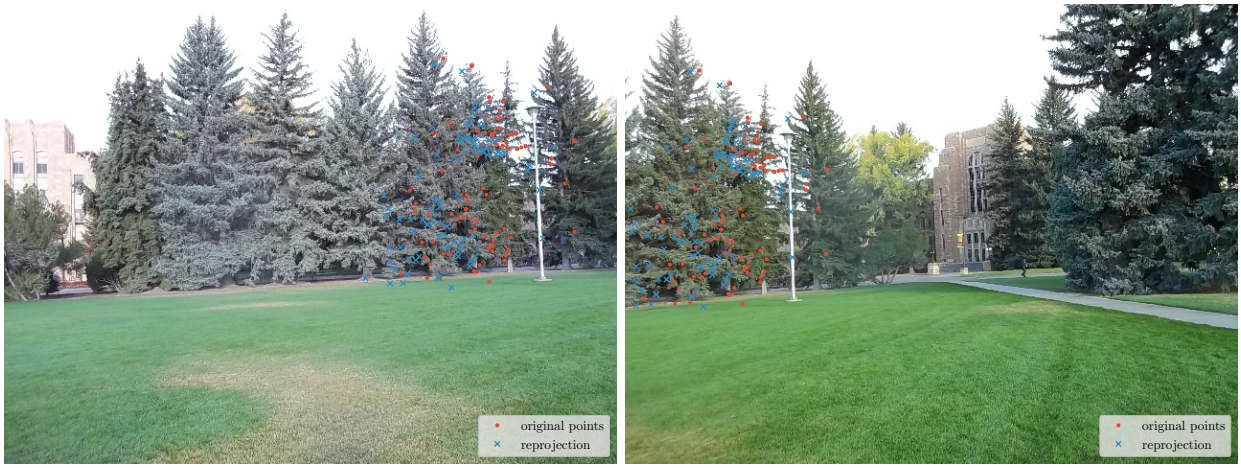

Figure 3: Reprojections of Correspondence Points Between Image 5 and Image 6

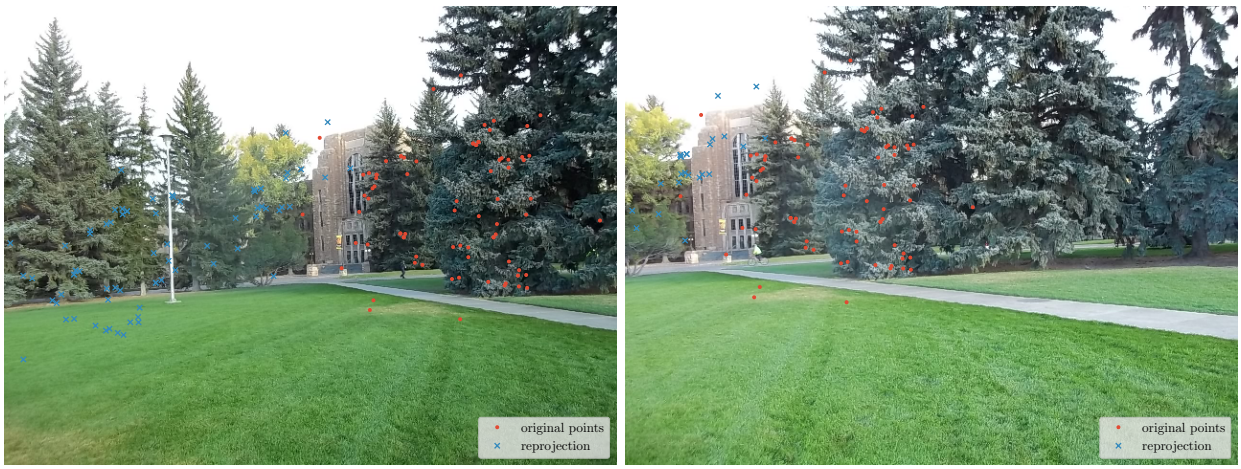

Figure 4: Reprojections of Correspondence Points Between Image 6 and Image 7

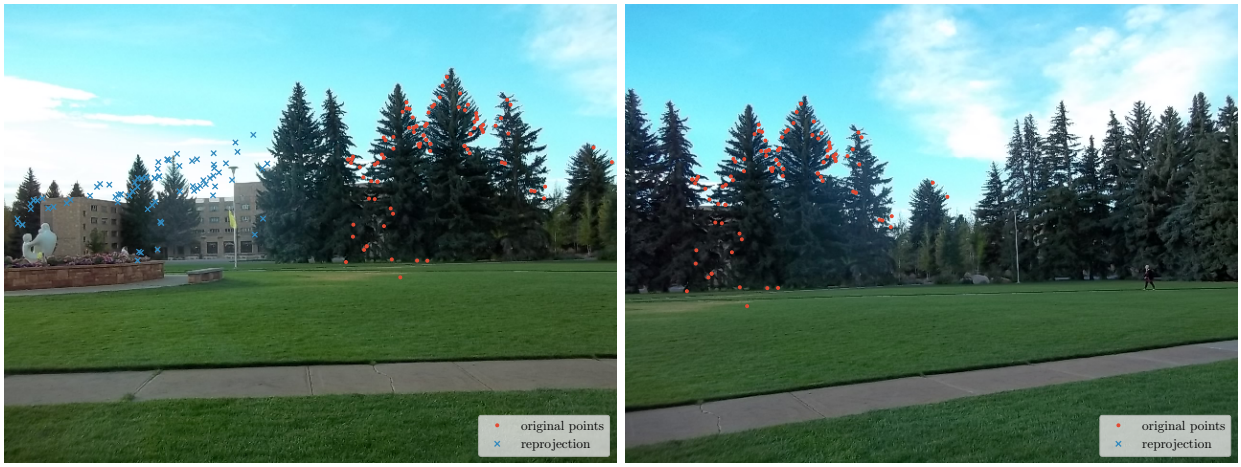

Figure 5: Reprojections of Correspondence Points Between Image 1 and Image 2

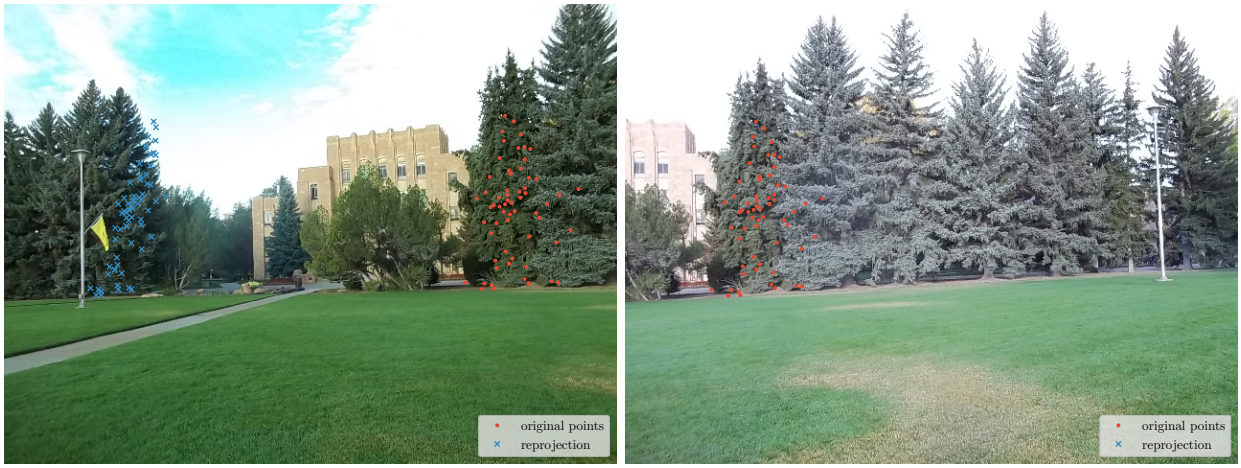

Figure 6: Reprojections of Correspondence Points Between Image 4 and Image 5

Download code at https://libaoj.in/Computer-Vision/1-Camera-Calibration-Using-Pure-Rotation/.

```python
#! /usr/bin/env python3
# -*- coding: utf-8 -*-
#
# camera_calibration.py - 2019-10-10 15:07
#
# Copyright © 2019 Libao Jin <jinlibao@outlook.com>
# Distributed under terms of the MIT license.
#
'''
Camera Calibration
'''

import os
import numpy as np
import scipy as sp
import scipy.linalg as LA
from scipy.io import loadmat
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import matplotlib.image as img
from matplotlib.backends.backend_pdf import PdfPages
import cv2


class CameraCalibration():
    '''Camera Calibratoin Using Pure Rotation'''

    def __init__(self):
        plt.style.use('ggplot')
        plt.rc('text', usetex=True)
        plt.rc('font', family='serif')
        np.random.seed(seed=666)

    def hat(self, v):
        return np.array([[0, -v[2], v[1]], [v[2], 0, -v[0]], [-v[1], v[0], 0]])

    def D(self, u1, u2):
        D = np.zeros((4, 4))
        D[0, 1:] = self.hat(u1).dot(u2).T
        D[1:, 0] = self.hat(u1).dot(u2).reshape(3)
        D[1:, 1:] = u1.dot(u2.T) + u2.dot(u1.T) - 2 * u1.T.dot(u2) * np.eye(3)
        return D

    def eigvec(self, A):
        a, w = np.linalg.eig(A)
        return w[:, 0].reshape((4, 1))

    def q2R(self, q4):
        q0 = q4[0, 0]
        q = q4[1:, 0]
        R = np.eye(3) + 2 * q0 * self.hat(q) + 2 * self.hat(q).dot(self.hat(q))
        return R

    def optimal_quaternion(self, x1, x2, K):
        K_inv = np.linalg.inv(K)
        X1 = [K_inv.dot(x1[:, i]).reshape(3, 1) for i in range(x1.shape[1])]
        X2 = [K_inv.dot(x2[:, i]).reshape(3, 1) for i in range(x2.shape[1])]
```

```
        D = np.zeros((4, 4))
        for i in range(len(X1)):
            u1 = X1[i] / np.linalg.norm(X1[i])
            u2 = X2[i] / np.linalg.norm(X2[i])
            D += self.D(u1, u2)
        q = self.eigvec(D)
        R = self.q2R(q)
        return R

    def K(self, k):
        k = k.reshape(5, 1)
        return np.array([[k[0, 0], k[1, 0], k[2, 0]], [0, k[3, 0], k[4, 0]],
                         [0, 0, 1]])

    def invert_K(self, k):
        k = k.reshape(5, 1)
        return np.array([[
            1 / k[0, 0], -k[1, 0] / (k[0, 0] * k[3, 0]),
            k[1, 0] * k[4, 0] / (k[0, 0] * k[3, 0]) - k[2, 0] / k[0, 0]
        ], [0, 1 / k[3, 0], -k[4, 0] / k[3, 0]], [0, 0, 1]])

    def find_R_list_pair(self, x1_list, x2_list, tol, max_iter):
        R_list = []
        k_list = []
        for m in range(len(x1_list)):
            print('Finding R of Image {:d} and Image {:d}'.format(m + 1, m + 2))
            x1, x2 = x1_list[m], x2_list[m]
            k = np.random.rand(5, 1)
            R = self.optimal_quaternion(x1, x2, self.K(k))
            K = lambda k: self.K(k)
            K_inv = lambda k: self.invert_K(k)
            x1_reproj = lambda R, k: np.sum(
                np.array([
                    np.linalg.norm(x1[:, i] - K(k).dot(
                        R.T.dot(K_inv(k).dot(x2[:, i])))))
                    for i in range(x2.shape[1])
                ])) / (2 * x2.shape[1])
            x2_reproj = lambda R, k: np.sum(
                np.array([
                    np.linalg.norm(x2[:, i] - K(k).dot(
                        R.dot(K_inv(k).dot(x1[:, i])))))
                    for i in range(x1.shape[1])
                ])) / (2 * x1.shape[1])
            J = lambda k: x1_reproj(R, k) + x2_reproj(R, k)
            j = 0
            k_best, J_best = k, J(k)
            while J(k) > tol and j < max_iter:
                if J_best > J(k):
                    k_best, J_best = k, J(k)
                print('{:3d}: {:.4f}'.format(j, J(k)))
                R = self.optimal_quaternion(x1, x2, self.K(k))
                J = lambda k: np.sum(x1_reproj(R, k)) + np.sum(x2_reproj(R, k))
                k = minimize(J, k.T)['x'].reshape(5, 1)
                j += 1
            print('{:3d}: {:.4f}'.format(j, J(k)))
            if J_best > J(k):
                k_best, J_best = k, J(k)
            #  R = self.optimal_quaternion(x1, x2, self.K(k_best))
            R_list.append(R)
```

```
            k_list.append(k_best)
        # for i in range(len(R_list)):
        #     print('{:d}: {:.4f}'.format(i, J(k_list[i])))
        #     print(R_list[i])
        #     print(k_list[i])
        return (R_list, k_list)

    def calibrate(self, x1_list, x2_list, tol, max_iter, output_dir):
        max_iter_1, max_iter_2, max_iter_3 = max_iter
        k = np.random.rand(5, 1)
        R_list, _ = self.find_R_list_pair(x1_list, x2_list, tol, max_iter_1)
        K = lambda k: self.K(k)
        K_inv = lambda k: self.invert_K(k)
        x1_reproj = lambda R, k, j: np.sum(
            np.array([
                np.linalg.norm(x1_list[j][:, i] - K(k).dot(
                    R.T.dot(K_inv(k).dot(x2_list[j][:, i])))))
                for i in range(x1_list[j].shape[1])
            ])) / (2 * x1_list[j].shape[1] * len(x1_list))
        x2_reproj = lambda R, k, j: np.sum(np.array([
            np.linalg.norm(x2_list[j][:, i] - K(k).dot(
                R.dot(K_inv(k).dot(x1_list[j][:, i])))))
            for i in range(x2_list[j].shape[1])
        ])) / (2 * x2_list[j].shape[1] * len(x2_list))
        J = lambda k: np.sum(np.array([
            x1_reproj(R_list[j], k, j) + x2_reproj(R_list[j], k, j)
            for j in range(len(x1_list))
        ]))
        print('Finding the calibration matrix K for all rotations')
        k = np.random.rand(5, 1)
        k_best, J_best = k, J(k)
        m = 0
        while J_best > tol and m < max_iter_2:
            n = 0
            while J(k) > tol and n < max_iter_3:
                k = np.random.rand(5, 1)
                if J(k) < J_best:
                    k_best, J_best = k, J(k)
                print('{:3d}, {:3d}: {:.4f}'.format(m, n, J(k)))
                J = lambda k: np.sum([
                    x1_reproj(R_list[j], k, j) + x2_reproj(R_list[j], k, j)
                    for j in range(len(x1_list))
                ])
                k = minimize(J, k.T)['x'].reshape(5, 1)
                n += 1
            if J(k) < J_best:
                k_best, J_best = k, J(k)
            k = k_best
            # for i in range(len(x1_list)):
            #     R = self.optimal_quaternion(x1_list[i], x2_list[i], self.K(k))
            #     R_list[i] = R
            print('{:3d}, {:3d}: {:.4f}'.format(m, n, J(k)))
            m += 1
        return (R_list, self.K(k))

    def read_data(self, filename):
        mat = loadmat(filename)
        return (mat['x1pMat'], mat['x2pMat'])

```

```
176     def write_csv(self, R_list, K, output_dir):
177         if not os.path.exists(output_dir):
178             os.mkdir(output_dir)
179         for i in range(len(R_list)):
180             R = R_list[i]
181             np.savetxt('{:s}/R_{:d}.csv'.format(output_dir, i + 1),
182                        R,
183                        delimiter=',',
184                        fmt='%15.8f',
185                        newline='\n')
186         np.savetxt('{:s}/K.csv'.format(output_dir),
187                    K,
188                    delimiter=',',
189                    fmt='%15.8f',
190                    newline='\n')
191 
192     def read_csv(self, output_dir):
193         if not os.path.exists(output_dir):
194             os.mkdir(output_dir)
195         R_list = []
196         for i in range(6):
197             R = np.loadtxt('{:s}/R_{:d}.csv'.format(output_dir, i + 1),
198                            delimiter=',')
199             R_list.append(R)
200         K = np.loadtxt('{:s}/K.csv'.format(output_dir), delimiter=',')
201         return (R_list, K)
202 
203     def disp(self, A, name='A', filename=''):
204         nrow, ncol = A.shape
205         if len(filename) == 0:
206             print(r'\begin{equation*}')
207             print('{:s} = \n'.format(name))
208             print(r'\begin{bmatrix}')
209             for i in range(nrow):
210                 print(r'{:8.4f} & {:8.4f} & {:8.4f} \\'.format(
211                     A[i, 0], A[i, 1], A[i, 2]))
212             print(r'\end{bmatrix}')
213             print(r'\end{equation*}')
214 
215         with open(filename, 'w') as f:
216             f.write('\\begin{equation*}\n')
217             f.write('{:s} = \n'.format(name))
218             f.write('\\begin{bmatrix}\n')
219             for i in range(nrow):
220                 f.write('{:8.4f} & {:8.4f} & {:8.4f} \\\\\n'.format(
221                     A[i, 0], A[i, 1], A[i, 2]))
222             f.write('\\end{bmatrix}.\n')
223             f.write('\\end{equation*}\n')
224 
225     def print_matrix(self, output_dir):
226         if not os.path.exists(output_dir):
227             os.mkdirG(output_dir)
228         R_list, K = self.read_csv(output_dir)
229         for i in range(len(R_list)):
230             self.disp(R_list[i], '\\widetilde{{R}}_{{{:d}}}'.format(i + 1),
231                       '{:s}/R_{:d}.tex'.format(output_dir, i + 1))
232         self.disp(K, 'K', '{:s}/K.tex'.format(output_dir))
233 
234     def show_image(self, I, x, xr, output_dir, name):
```

```
235         if not os.path.exists(output_dir):
236             os.mkdir(output_dir)
237         filename = '{:s}/{:s}'.format(output_dir, name)
238         with PdfPages(filename) as pdf:
239             fig = plt.figure(frameon=False)
240             ax = plt.Axes(fig, [0., 0., 1., 1.])
241             ax.set_axis_off()
242             fig.add_axes(ax)
243             if I.ndim == 2:
244                 plt.imshow(I, cmap='gray')
245             else:
246                 plt.imshow(I)
247             h1, = plt.plot(x[0, :],
248                            x[1, :],
249                            '.',
250                            markersize=4,
251                            label='original points')
252             h2, = plt.plot(xr[0, :],
253                            xr[1, :],
254                            'x',
255                            markersize=4,
256                            label='reprojection')
257             plt.legend(handles=[h1, h2], loc='lower right')
258             plt.grid()
259             plt.axis([-0.5, 1279.5, 959.5, -0.5])
260             plt.axis('off')
261             plt.show(block=False)
262             plt.savefig(filename.replace('pdf', 'png'), format='png', dpi=300)
263             pdf.savefig(fig)
264             plt.close()
265 
266     def reproject(self, x, R=np.eye(3), K=np.eye(3)):
267         K_inv = np.linalg.inv(K)
268         x_reproj = K.dot(R.dot(K_inv.dot(x)))
269         return x_reproj
270 
271     def solve(self, mat_file, images, output_dir, tol=1, max_iter=(50, 10, 5)):
272         I = [img.imread(image) for image in images]
273         x1pMat, x2pMat = self.read_data(mat_file)
274         ncols = []
275         for i in range(len(I) - 1):
276             for j in range(x1pMat[:, :, i].shape[1]):
277                 if sp.linalg.norm(x1pMat[:, j, i]
278                                   ) == 0 or j == x1pMat[:, :, i].shape[1] - 1:
279                     ncols.append(j)
280                     break
281         x1_list = [x1pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
282         x2_list = [x2pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
283         if (os.path.exists('{:s}/K.csv'.format(output_dir))):
284             R_list, K = self.read_csv(output_dir)
285         else:
286             R_list, K = self.calibrate(x1_list, x2_list, tol, max_iter,
287                                        output_dir)
288         for i in range(len(R_list)):
289             print('R from Image {:d} to Image {:d}:'.format(i + 1, i + 2))
290             print(R_list[i])
291         print('K:')
292         print(K)
293 
```

```
294         for i in range(len(I) - 1):
295             I1, I2 = I[i].copy(), I[i + 1].copy()
296             x1pmat = x1pMat[:, 0:ncols[i], i]
297             x2pmat = x2pMat[:, 0:ncols[i], i]
298             x1pReproMat = self.reproject(x1pmat, R_list[i], K)
299             x2pReproMat = self.reproject(x2pmat, R_list[i], K)
300             filename_1 = 'prexy{:d}_2.pdf'.format(i + 1)
301             filename_2 = 'prexy{:d}_1.pdf'.format(i + 2)
302             self.show_image(I1, x1pmat, x1pReproMat, output_dir, filename_1)
303             self.show_image(I2, x2pmat, x2pReproMat, output_dir, filename_2)
304         self.write_csv(R_list, K, output_dir)
305         self.print_matrix(output_dir)
306 
307     def solve_cv2(self, mat_file, images, output_dir, tol=1, max_iter=(50, 10, 5)):
308         I = [cv2.imread(image) for image in images]
309         x1pMat, x2pMat = self.read_data(mat_file)
310         ncols = []
311         for i in range(len(I) - 1):
312             for j in range(x1pMat[:, :, i].shape[1]):
313                 if sp.linalg.norm(x1pMat[:, j, i]
314                                   ) == 0 or j == x1pMat[:, :, i].shape[1] - 1:
315                     ncols.append(j)
316                     break
317         x1_list = [x1pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
318         x2_list = [x2pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
319         if (os.path.exists('{:s}/K.csv'.format(output_dir))):
320             R_list, K = self.read_csv(output_dir)
321         else:
322             R_list, K = self.calibrate(x1_list, x2_list, tol, max_iter,
323                                        output_dir)
324         for i in range(len(R_list)):
325             print('Rotation from Image {:d} to Image {:d}:'.format(
326                 i + 1, i + 2))
327             print(R_list[i])
328         print('Calibration matrix:')
329         print(K)
330 
331         for i in range(len(I) - 1):
332             I1, I2 = I[i].copy(), I[i + 1].copy()
333             x1pmat = x1pMat[:, 0:ncols[i], i]
334             x2pmat = x2pMat[:, 0:ncols[i], i]
335             x1pReproMat = self.reproject(x1pmat, R_list[i], K)
336             x2pReproMat = self.reproject(x2pmat, R_list[i], K)
337             for j in range(0, x1pmat.shape[1]):
338                 cv2.putText(I1, "{}".format("X"),
339                             (int(x1pmat[0, j]), int(x1pmat[1, j])),
340                             cv2.FONT_HERSHEY_SIMPLEX, 0.3, (0, 0, 255), 2)
341                 cv2.putText(I2, "{}".format("X"),
342                             (int(x2pmat[0, j]), int(x2pmat[1, j])),
343                             cv2.FONT_HERSHEY_SIMPLEX, 0.3, (0, 0, 255), 2)
344                 cv2.putText(I1, "{}".format("O"),
345                             (int(x1pReproMat[0, j]), int(x1pReproMat[1, j])),
346                             cv2.FONT_HERSHEY_SIMPLEX, 0.3, (255, 0, 0), 2)
347                 cv2.putText(I2, "{}".format("O"),
348                             (int(x2pReproMat[0, j]), int(x2pReproMat[1, j])),
349                             cv2.FONT_HERSHEY_SIMPLEX, 0.3, (255, 0, 0), 2)
350             cv2.imshow("Image 1 reproj (prexy{:d}.jpg)".format(i + 1), I1)
351             cv2.imshow("Image 2 reproj (prexy{:d}.jpg)".format(i + 2), I2)
352             cv2.imwrite('{:s}/prexy{:d}_2.jpg'.format(output_dir, i + 1), I1)
```

```
            cv2.imwrite('{:s}/prexy{:d}_1.jpg'.format(output_dir, i + 2), I2)
            cv2.waitKey(300)
            cv2.destroyAllWindows()
        self.write_csv(R_list, K, output_dir)
        self.print_matrix(output_dir)

    def test_optimal_quaternion(self, output_dir):
        pi = np.arctan(1) * 4
        k = np.array([2, 0, 1, 2, 1])
        w = np.array([0, 2, 0])
        w = w / np.linalg.norm(w)
        t = pi / 3
        R = LA.expm(self.hat(w) * t)
        K = self.K(k)
        K_inv = self.invert_K(k)
        x1 = np.random.rand(3, 20) * 10000
        x1[2, :] = np.random.rand(1, 20) * 10 + 1
        x11 = x1.copy()
        for i in range(x11.shape[1]):
            x11[:, i] /= x11[2, i]
        x2 = K.dot(R.dot(K_inv.dot(x11)))
        x22 = x2.copy()
        for i in range(x22.shape[1]):
            x22[:, i] /= x22[2, i]
        R2 = self.optimal_quaternion(x11, x22, K)
        x2p = self.reproject(x11, R2, K)
        print("K:"); print(K)
        print(x2 - x2p)
        print(np.linalg.norm(x2p[0:2, :] - x2[0:2, :]))
        print(R)
        print(R2)
        print(R - R2)

        if not os.path.exists(output_dir):
            os.mkdir(output_dir)
        self.disp(K, 'K', '{:s}/K_test.tex'.format(output_dir))
        self.disp(R, 'R', '{:s}/R_test.tex'.format(output_dir))
        self.disp(R2,'R\'', '{:s}/R_test_prime.tex'.format(output_dir))
        self.disp(R - R2, 'R_{error}', '{:s}/R_test_error.tex'.format(output_dir))

    def test_calibrate(self):
        pass


if __name__ == '__main__':
    c = CameraCalibration()
    mat_file = './data/pureRotPrexyCorrespondencePoints.mat'
    output_dir = './output'
    images = ['./data/prexy{:d}.jpg'.format(i + 1) for i in range(7)]
    c.solve(mat_file, images, output_dir, 5, (50, 1, 20))
    c.test_optimal_quaternion('test')
    # c.solve_cv2(mat_file, images, output_dir, 1, 50)
'''
End of file
'''
```