

# EE 5885 - Geometry and Deep Computer Vision Project 1

Libao Jin

October 15, 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. Thoroughly 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 two corresponding points on Image 1 and Image 2, respectively. With the help of the calibration matrix, we can get the pixel coordinates of aforementioned two points  $\mathbf{x}_1$ ,  $\mathbf{x}_2$  as follows,

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

It is known that

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

where  $R_{2,1}$  is the rotation matrix from camera frame 1 to camera frame 2. Then we want to find  $\tilde{R}_{2,1}$  such that minimize the norm of the residual

$$\|\mathbf{x}_2 - K_2 \tilde{R}_{2,1} K_1^{-1} \mathbf{x}_1\|^2 + \|\mathbf{x}_1 - K_1 \tilde{R}_{2,1}^{-1} K_2^{-1} \mathbf{x}_2\|^2 = \|\mathbf{x}_2 - \tilde{\mathbf{x}}_1\|^2 + \|\mathbf{x}_1 - \tilde{\mathbf{x}}_2\|^2,$$

where  $\tilde{\mathbf{x}}_2 = K_2 \tilde{R}_{2,1} K_1^{-1} \mathbf{x}_1$  is the reprojection of  $\mathbf{x}_1$  from Image 1 to Image 2 while  $\tilde{\mathbf{x}}_1 = K_1 \tilde{R}_{2,1}^T K_2^{-1} \mathbf{x}_2$  is the reprojection of  $\mathbf{x}_2$  from Image 2 to Image 1. In general, given  $n$  corresponding points, say  $\mathbf{x}_1^i, \mathbf{x}_2^i, i = 1, 2, \dots, m$ , assume that the calibration matrices  $K_1 = K_2 = \dots = K_m = K$ , we want use these points to find a rotation matrix such that

$$\min J = \min \sum_{i=1}^m (\|\mathbf{x}_2^i - \tilde{\mathbf{x}}_1^i\|^2 + \|\mathbf{x}_1^i - \tilde{\mathbf{x}}_2^i\|^2) = \min_{R \in \mathbb{R}^{3 \times 3}} \sum_{i=1}^m (\|\mathbf{x}_2^i - KRK^{-1} \mathbf{x}_1^i\|^2 + \|\mathbf{x}_1^i - KR^T K^{-1} \mathbf{x}_2^i\|^2).$$

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  $\tilde{R} = \tilde{R}_{2,1}$  from Image 1 to Image 2.

1.  $\mathbf{X}_2^i = K^{-1} \mathbf{x}_2^i$  and  $\mathbf{X}_1 = K^{-1} \mathbf{x}_1^i$ .

$$2. D^i = \begin{bmatrix} 0 & (\hat{\mathbf{X}}_2^i \mathbf{X}_1^i)^T \\ \hat{\mathbf{X}}_2^i \mathbf{X}_1^i & \mathbf{X}_2^i \mathbf{X}_1^{iT} + \mathbf{X}_1^i \mathbf{X}_2^{iT} - 2\mathbf{X}_2^{iT} \mathbf{X}_1^i I \end{bmatrix}.$$

$$3. D = \sum_{i=1}^m D^i.$$

4. Find the eigenvector  $q = [q_0 \quad \mathbf{q}]^T$  of  $D$  which corresponds to the largest eigenvalue.

5.  $\tilde{R} = I + 2q_0 \hat{\mathbf{q}} + 2(\mathbf{q}\mathbf{q}^T - \mathbf{q}^T \mathbf{q}I)$ .

**end**

---

## 2.2 Finding Calibration Matrix $K$ By Solving A Optimization Problem

Suppose the rotation matrix  $R$  is given, then we want to find the calibration matrix such that  $J$  is minimized. 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,  $K$  is determined by five unknowns. Let  $\mathbf{k} = [k_1 \ k_2 \ k_3 \ k_4 \ k_5]^T$ , we can use  $\mathbf{k}$  to construct the calibration matrix  $K$  and the inverse of that  $K^{-1}$  as follows,

$$K = \begin{bmatrix} k_1 & k_2 & k_3 \\ 0 & k_4 & k_5 \\ 0 & 0 & 1 \end{bmatrix}, K^{-1} = \begin{bmatrix} \frac{1}{k_1} & -\frac{k_2}{k_1 k_4} & \frac{k_2 k_5}{k_1 k_4} - \frac{k_3}{k_1} \\ 0 & \frac{1}{k_4} & -\frac{k_5}{k_4} \\ 0 & 0 & 1 \end{bmatrix}. \quad (2.1)$$

To find a common calibration matrix  $K$  to multiple pairs of images, we suppose that the rotation matrix  $R_{j+1,j}$ ,  $j = 1, 2, \dots, n$  from Image  $j$  to Image  $j + 1$  are known, the sum of the reprojection error of a sequence of the rotations ( $n + 1$  rotations) is

$$\begin{aligned} J &= \sum_{j=1}^n J_{j+1,j} + J_{j,j+1} \\ &= \sum_{j=1}^n \sum_{i=1}^{m_j} (\|\mathbf{x}_{j+1}^i - KR_{j+1,j}K^{-1}\mathbf{x}_j^i\|^2 + \|\mathbf{x}_j^i - KR_{j,j+1}K^{-1}\mathbf{x}_{j+1}^i\|^2) \\ &= \sum_{j=1}^n \sum_{i=1}^{m_j} (\|\mathbf{x}_{j+1}^i - KR_{j+1,j}K^{-1}\mathbf{x}_j^i\|^2 + \|\mathbf{x}_j^i - KR_{j+1,j}^T K^{-1}\mathbf{x}_{j+1}^i\|^2) \\ &= \sum_{j=1}^n \sum_{i=1}^{m_j} (\|\mathbf{x}_{j+1}^i - K\tilde{R}_j K^{-1}\mathbf{x}_j^i\|^2 + \|\mathbf{x}_j^i - K\tilde{R}_j^T K^{-1}\mathbf{x}_{j+1}^i\|^2), \end{aligned}$$

where  $\tilde{R}_j = R_{j+1,j}$ . Note that  $J = J(\mathbf{k})$ , then we solve the minimization problem to obtain  $\mathbf{k}$  as below

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

Then we can construct  $K$  and  $K^{-1}$  by (2.1).

## 2.3 Finding Rotation Matrix $R$ and Calibration Matrix $K$

In this project, the previous two sections are not directly applicable, because either the calibration matrix  $K$  or the rotation matrices  $R_{j+1,j}$ ,  $j = 1, 2, \dots, n$  are not given. In fact, while solving for the  $\mathbf{k}$  in the optimization problem, we use some numerical schemes which requires a 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:**  $\{\mathbf{x}_j^i\}_{i=1}^{m_j}, j = 1, 2, \dots, n$  are the pixel coordinates of  $m_j$  correspondence points,  
 $tol$  is the tolerance of the  $J$ .

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

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

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

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

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

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

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

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

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

9       **end**

end

---

### 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:

- `read_data(filename)`: reads MATLAB `.mat` file into Python.
- `read_csv(output_dir)`: reads stored calibration matrix and rotation matrices.
- `print(A, name, filename)`: formats the output of the calibration matrix and rotation matrices to `LATEX`.
- `print_matrix(output_dir)`: saves the rotation matrix  $R_j, j = 1, 2, \dots, 6$  and calibration matrix  $K$  to `.tex` files.

- `show_image(I, x, xr, filename)`: displays the image  $I$  with correspondence points and their reprojections using `matplotlib.image` and `matplotlib.pyplot`.
- `hat(v)`: works as the hat operator ( $\hat{\cdot}$ ), converting a vector to a skew-symmetric matrix.
- `D(z, v)`: 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)`: the implementation of Algorithm 1 with calibrated points.
- `K(k)`: constructs calibration matrix  $K$  from the vector  $k$ .
- `invert_K(k)`: constructs the inverse of the calibration matrix  $K^{-1}$  from the vector  $k$ .
- `find_k(x1, x2, tol, max_iter)`: calibrates the correspondence points and apply Algorithm 1 to find the rotation matrix  $R$ , and then repeatedly find the minimizer  $k$  using `scipy.optimize.minimize`.
- `find_R(x1, x2, k)`: the implementation of Algorithm 1.
- `find_R_K(x1, x2, tol, max_iter)`: puts `find_k` and `find_R` together, and reconstruct  $K$  from  $k$ .
- `find_all_R(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.
- `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).

## 4 Result and Discussion

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

$$K = \begin{bmatrix} 36.8035 & 26.0249 & -0.5228 \\ 0.0000 & 19.2506 & 6.0843 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}.$$

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

$$\tilde{R}_2 = \begin{bmatrix} 0.9995 & 0.0000 & 0.0328 \\ 0.0000 & 1.0000 & -0.0003 \\ -0.0328 & 0.0003 & 0.9995 \end{bmatrix}.$$

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 3, we can see that Algorithm 2 is very satisfactory: the reprojections of the points coincide with the actual points with very small error. When it comes to image 1 and 2 (Figure 4) or image 4 and 5 (Figure 5), the reprojections of the correspondence points are slightly off. However, Algorithm 2 failed to reproject points to its correspondence points between Image 5 and 6 (Figure 6). 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:

$$\begin{aligned}\tilde{R}_1 &= \begin{bmatrix} 0.9988 & -0.0396 & -0.0294 \\ 0.0396 & 0.9992 & 0.0008 \\ 0.0294 & -0.0020 & 0.9996 \end{bmatrix}. \\ \tilde{R}_3 &= \begin{bmatrix} 1.0000 & 0.0001 & -0.0000 \\ -0.0001 & 1.0000 & -0.0033 \\ 0.0000 & 0.0033 & 1.0000 \end{bmatrix}. \\ \tilde{R}_4 &= \begin{bmatrix} 0.9956 & -0.0037 & -0.0941 \\ -0.0020 & 0.9982 & -0.0600 \\ 0.0942 & 0.0599 & 0.9938 \end{bmatrix}. \\ \tilde{R}_5 &= \begin{bmatrix} 0.9591 & 0.0340 & -0.2811 \\ 0.1258 & 0.8381 & 0.5309 \\ 0.2536 & -0.5445 & 0.7995 \end{bmatrix}. \\ \tilde{R}_6 &= \begin{bmatrix} 1.0000 & 0.0005 & 0.0062 \\ -0.0006 & 1.0000 & 0.0092 \\ -0.0062 & -0.0092 & 0.9999 \end{bmatrix}.\end{aligned}$$

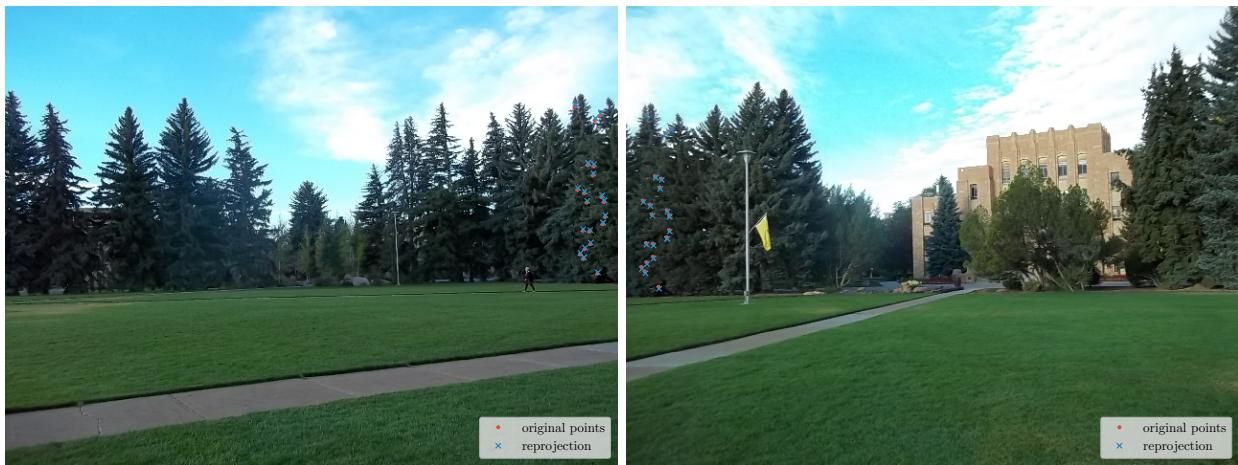


Figure 1: Reprojections of Correspondence Points Between Image 2 and Image 3

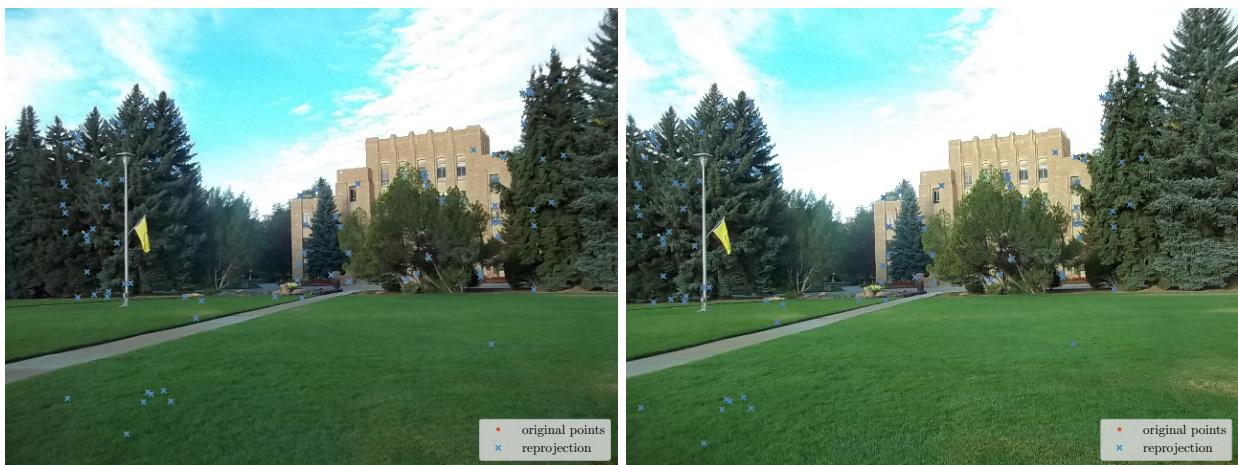


Figure 2: Reprojections of Correspondence Points Between Image 3 and Image 4



Figure 3: Reprojections of Correspondence Points Between Image 6 and Image 7



Figure 4: Reprojections of Correspondence Points Between Image 1 and Image 2



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



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

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

```
1 #! /usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 #
4 # camera_calibration.py - 2019-10-10 15:07
5 #
6 # Copyright © 2019 Libao Jin <jinlibao@outlook.com>
7 # Distributed under terms of the MIT license.
8 #
9 '''
10 Camera Calibration
11 '''
12
13 import os
14 import numpy as np
15 import scipy as sp
16 from scipy.io import loadmat
17 from scipy.optimize import minimize
18 import matplotlib.pyplot as plt
19 import matplotlib.image as img
20 from matplotlib.backends.backend_pdf import PdfPages
21 import cv2
22
23
24 class CameraCalibration():
25     '''Camera Calibration Using Pure Rotation'''
26
27     def __init__(self):
28         plt.style.use('ggplot')
29         plt.rc('text', usetex=True)
30         plt.rc('font', family='serif')
31         np.random.seed(seed=888)
32
33     def read_data(self, filename):
34         mat = loadmat(filename)
35         return (mat['x1pMat'], mat['x2pMat'])
36
37     def read_csv(self, output_dir):
38         R_list = []
39         for i in range(6):
40             R = np.loadtxt('{:s}/R_{:d}.csv'.format(output_dir, i + 1),
41                           delimiter=',')
42             R_list.append(R)
43         K = np.loadtxt('{:s}/K.csv'.format(output_dir), delimiter=',')
44         return (R_list, K)
45
46     def print(self, A, name='A', filename=''):
47         nrow, ncol = A.shape
48         if len(filename) == 0:
49             print(r'\begin{equation*}')
50             print('{:s} = \n'.format(name))
51             print(r'\begin{bmatrix}')
52             for i in range(nrow):
53                 print(r'{:8.4f} & {:8.4f} & {:8.4f} \\'.format(
54                     A[i, 0], A[i, 1], A[i, 2]))
55             print(r'\end{bmatrix}')
56             print(r'\end{equation*}')
57         return
```

```

58
59     with open(filename, 'w') as f:
60         f.write('\\begin{equation*}\n')
61         f.write('{:s} = \n'.format(name))
62         f.write('\\begin{bmatrix}\n')
63         for i in range(nrow):
64             f.write('{:8.4f} & {:8.4f} & {:8.4f} \\\\\n'.format(
65                 A[i, 0], A[i, 1], A[i, 2]))
66         f.write('\\end{bmatrix}.\n')
67         f.write('\\end{equation*}\n')
68
69     def print_matrix(self, output_dir):
70         R_list, K = self.read_csv(output_dir)
71         for i in range(len(R_list)):
72             self.print(R_list[i], '\\widetilde{{R}}_{\\{{:d}\\}}'.format(i + 1),
73                         '{:s}/R_{:d}.tex'.format(output_dir, i + 1))
74         self.print(K, 'K', '{:s}/K.tex'.format(output_dir))
75
76     def show_image(self, I, x, xr, filename):
77         with PdfPages(filename) as pdf:
78             fig = plt.figure(frameon=False)
79             ax = plt.Axes(fig, [0., 0., 1., 1.])
80             ax.set_axis_off()
81             fig.add_axes(ax)
82             if I.ndim == 2:
83                 plt.imshow(I, cmap='gray')
84             else:
85                 plt.imshow(I)
86             h1, = plt.plot(x[0, :],
87                            x[1, :],
88                            '.', 
89                            markersize=4,
90                            label='original points')
91             h2, = plt.plot(xr[0, :],
92                            xr[1, :],
93                            'x',
94                            markersize=4,
95                            label='reprojection')
96             plt.legend(handles=[h1, h2], loc='lower right')
97             plt.grid()
98             plt.axis([-0.5, 1279.5, 959.5, -0.5])
99             plt.axis('off')
100            plt.show(block=False)
101            plt.savefig(filename.replace('pdf', 'png'), format='png', dpi=300)
102            pdf.savefig(fig)
103            plt.close()
104
105    def hat(self, v):
106        return np.array([[0, -v[2], v[1]], [v[2], 0, -v[0]], [-v[1], v[0], 0]])
107
108    def D(self, z, v):
109        D = np.zeros((4, 4))
110        D[0, 1:] = self.hat(z).dot(v).T
111        D[1:, 0] = self.hat(z).dot(v).reshape(3)
112        D[1:, 1:] = z.dot(v.T) + v.dot(z.T) - 2 * z.T.dot(v) * np.eye(3)
113        return D
114
115    def eigvec(self, A):
116        a, w = np.linalg.eig(A)

```

```
117     return w[:, 0].reshape((4, 1))
118
119     def q2R(self, q4):
120         q0 = q4[0, 0]
121         q = q4[1:, 0]
122         R = np.eye(3) + 2 * q0 * self.hat(q) + 2 * self.hat(q).dot(self.hat(q))
123         return R
124
125     def optimal_quaternion(self, x1, x2):
126         D = np.zeros((4, 4))
127         for i in range(len(x1)):
128             z, v = x1[i].reshape(3, 1), x2[i].reshape(3, 1)
129             D += self.D(z, v)
130         q = self.eigvec(D)
131         R = self.q2R(q)
132         return R
133
134     def K(self, k):
135         k = k.reshape(5, 1)
136         return np.array([[k[0, 0], k[1, 0], k[2, 0]], [0, k[3, 0], k[4, 0]],
137                         [0, 0, 1]])
138
139     def invert_K(self, k):
140         k = k.reshape(5, 1)
141         return np.array([
142             1 / k[0, 0], -k[1, 0] / (k[0, 0] * k[3, 0]),
143             k[1, 0] * k[4, 0] / (k[0, 0] * k[3, 0]) - k[2, 0] / k[0, 0],
144         ], [0, 1 / k[3, 0], -k[4, 0] / k[3, 0]], [0, 0, 1])
145
146     def find_k(self, x1, x2, tol, max_iter):
147         R = self.find_R(x1, x2, np.random.rand(5, 1))
148         K = lambda k: self.K(k)
149         K_inv = lambda k: self.invert_K(k)
150         x1_reproj = lambda R, k: np.array([
151             np.linalg.norm(x2[:, i] - K(k).dot(R.T.dot(K_inv(k)).dot(x2[:, i])))])
152             for i in range(x2.shape[1])
153         ]) / x2.shape[1]
154         x2_reproj = lambda R, k: np.array([
155             np.linalg.norm(x1[:, i] - K(k).dot(R.dot(K_inv(k)).dot(x1[:, i])))])
156             for i in range(x1.shape[1])
157         ]) / x1.shape[1]
158         J = lambda k: np.sum(x1_reproj(R, k)) + np.sum(x2_reproj(R, k))
159         k = np.random.rand(5, 1)
160         i = 0
161         while J(k) > tol and i < max_iter:
162             print('{:3d}: {:.4f}'.format(i, J(k)))
163             R = self.find_R(x1, x2, k)
164             J = lambda k: np.sum(x1_reproj(R, k)) + np.sum(x2_reproj(R, k))
165             y = minimize(J, k.T)
166             k = y['x'].reshape(5, 1)
167             i += 1
168         print('{:3d}: {:.4f}'.format(i, J(k)))
169         return k
170
171     def find_R(self, x1, x2, k):
172         K_guess = self.K(k)
173         K_inv_guess = np.linalg.inv(K_guess)
174         X1 = [K_inv_guess.dot(x1[:, i]) for i in range(x1.shape[1])]
175         X2 = [K_inv_guess.dot(x2[:, i]) for i in range(x2.shape[1])]
```

```

176     R = self.optimal_quaternion(X1, X2)
177     return R
178
179     def find_R_K(self, x1, x2, tol, max_iter):
180         k = self.find_k(x1, x2, tol, max_iter)
181         R = self.find_R(x1, x2, k)
182         K = self.K(k)
183         return (R, K)
184
185     def find_all_R(self, x1_list, x2_list, tol, max_iter):
186         R_list = []
187         for i in range(len(x1_list)):
188             print('Finding R between Image {:d} and Image {:d}...'.format(
189                 i, i + 1))
190             R, K = self.find_R_K(x1_list[i], x2_list[i], tol, max_iter)
191             R_list.append(R)
192         return R_list
193
194     def calibrate(self, x1_list, x2_list, tol, max_iter, output_dir):
195         R_list = self.find_all_R(x1_list, x2_list, tol, max_iter)
196         K = lambda k: self.K(k)
197         K_inv = lambda k: self.invert_K(k)
198         x1_reproj = lambda R, k, j: np.array([
199             np.linalg.norm(x2_list[j][:, i] - K(k).dot(
200                 R.T.dot(K_inv(k).dot(x2_list[j][:, i]))))
201             for i in range(x2_list[j].shape[1])
202         ]) / x2_list[j].shape[1]
203         x2_reproj = lambda R, k, j: np.array([
204             np.linalg.norm(x1_list[j][:, i] - K(k).dot(
205                 R.T.dot(K_inv(k).dot(x1_list[j][:, i]))))
206             for i in range(x1_list[j].shape[1])
207         ]) / x1_list[j].shape[1]
208         J = lambda k: np.sum([
209             np.sum(x1_reproj(R_list[j], k, j)) + np.sum(
210                 x2_reproj(R_list[j], k, j)) for j in range(len(x1_list))
211         ])
212         k = np.random.rand(5, 1)
213         print('Finding the calibration matrix K for all rotations...')
214         y = minimize(J, k)
215         k = y['x']
216         for i in range(len(R_list)):
217             R = R_list[i]
218             np.savetxt('{:s}/R_{:d}.csv'.format(output_dir, i + 1),
219                         R,
220                         delimiter=',',
221                         fmt='%.15.8f',
222                         newline='\n')
223             np.savetxt('{:s}/K.csv'.format(output_dir, i + 1),
224                         self.K(k),
225                         delimiter=',',
226                         fmt='%.15.8f',
227                         newline='\n')
228         return (R_list, self.K(k))
229
230     def reproject(self, x, R=np.eye(3), K=np.eye(3)):
231         K_inv = np.linalg.inv(K)
232         x_reproj = K.dot(R.dot(K_inv.dot(x)))
233         return x_reproj
234

```

```
235     def solve(self, mat_file, images, output_dir, tol=1, max_iter=200):
236         I = [img.imread(image) for image in images]
237         x1pMat, x2pMat = self.read_data(mat_file)
238         ncols = []
239         for i in range(len(I) - 1):
240             for j in range(x1pMat[:, :, i].shape[1]):
241                 if sp.linalg.norm(x1pMat[:, j, i])
242                     ) == 0 or j == x1pMat[:, :, i].shape[1] - 1:
243                     ncols.append(j)
244                     break
245         x1_list = [x1pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
246         x2_list = [x2pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
247         if os.path.exists('{:s}/K.csv'.format(output_dir)):
248             R_list, K = self.read_csv(output_dir)
249         else:
250             R_list, K = self.calibrate(x1_list, x2_list, tol, max_iter,
251                                         output_dir)
252         for i in range(len(R_list)):
253             print('Rotation from Image {:d} to Image {:d}:'.format(
254                 i + 1, i + 2))
255             print(R_list[i])
256         print('Calibration matrix:')
257         print(K)
258         self.print_matrix(output_dir)
259
260         for i in range(len(I) - 1):
261             I1, I2 = I[i].copy(), I[i + 1].copy()
262             x1pmat = x1pMat[:, 0:ncols[i], i]
263             x2pmat = x2pMat[:, 0:ncols[i], i]
264             x1pReprojMat = self.reproject(x1pmat, R_list[i], K)
265             x2pReprojMat = self.reproject(x2pmat, R_list[i], K)
266             filename_1 = '{:s}/prexy{:d}.pdf'.format(output_dir, i + 1)
267             filename_2 = '{:s}/prey{:d}.pdf'.format(output_dir, i + 2)
268             self.show_image(I1, x1pmat, x1pReprojMat, filename_1)
269             self.show_image(I2, x2pmat, x2pReprojMat, filename_2)
270
271     def solve_cv2(self, mat_file, images, output_dir, tol=16, max_iter=100):
272         I = [cv2.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('Rotation from Image {:d} to Image {:d}:'.format(
290                 i + 1, i + 2))
291             print(R_list[i])
292         print('Calibration matrix:')
293         print(K)
```

```
294     self.print_matrix(output_dir)
295
296     for i in range(len(I) - 1):
297         I1, I2 = I[i].copy(), I[i + 1].copy()
298         x1pmat = x1pMat[:, 0:ncols[i], i]
299         x2pmat = x2pMat[:, 0:ncols[i], i]
300         x1pReprojMat = self.reproject(x1pmat, R_list[i], K)
301         x2pReprojMat = self.reproject(x2pmat, R_list[i], K)
302         for j in range(0, x1pmat.shape[1]):
303             cv2.putText(I1, "{}".format("X"),
304                         (int(x1pmat[0, j]), int(x1pmat[1, j])),
305                         cv2.FONT_HERSHEY_SIMPLEX, 0.3, (0, 0, 255), 2)
306             cv2.putText(I2, "{}".format("X"),
307                         (int(x2pmat[0, j]), int(x2pmat[1, j])),
308                         cv2.FONT_HERSHEY_SIMPLEX, 0.3, (0, 0, 255), 2)
309             cv2.putText(I1, "{}".format("0"),
310                         (int(x1pReprojMat[0, j]), int(x1pReprojMat[1, j])),
311                         cv2.FONT_HERSHEY_SIMPLEX, 0.3, (255, 0, 0), 2)
312             cv2.putText(I2, "{}".format("0"),
313                         (int(x2pReprojMat[0, j]), int(x2pReprojMat[1, j])),
314                         cv2.FONT_HERSHEY_SIMPLEX, 0.3, (255, 0, 0), 2)
315             cv2.imshow("Image 1 reproj (prexy{:d}.jpg)".format(i + 1), I1)
316             cv2.imshow("Image 2 reproj (prexy{:d}.jpg)".format(i + 2), I2)
317             cv2.imwrite('{:s}/prexy{:d}_2.jpg'.format(output_dir, i + 1), I1)
318             cv2.imwrite('{:s}/prexy{:d}_1.jpg'.format(output_dir, i + 2), I2)
319             cv2.waitKey(300)
320             cv2.destroyAllWindows()
321
322
323 if __name__ == '__main__':
324     c = CameraCalibration()
325     mat_file = './data/pureRotPrexyCorrespondencePoints.mat'
326     output_dir = './output'
327     images = ['./data/prexy{:d}.jpg'.format(i + 1) for i in range(7)]
328     c.solve(mat_file, images, output_dir, 1, 200)
329     # c.solve_cv2(mat_file, images, output_dir, 1, 200)
330     '''
331 End of file
332'''
```