EE 5885 - Geometry and Deep Computer Vision Project 1

Libao Jin

October 14, 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 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

$$\mathbf{X}_2 = R_{2,1}\mathbf{X}_1 \implies \mathbf{x}_2 = K_2R_{2,1}K_1^{-1}\mathbf{x}_1,$$

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

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

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 while $\widetilde{\mathbf{x}}_1 = K_1\widetilde{R}_{2,1}^TK_2^{-1}\mathbf{x}_2$ is the reprojection of $\mathbf{x}_2$ from Image 2 to Image 1. In general, given n corresponding points, say $\mathbf{x}_1^i, \mathbf{x}_2^i, i = 1, 2, \ldots, m$, assume that the calibration matrices $K_1 = K_2 = \cdots K_m = K$, we want use these points to find a rotation matrix such that

$$\min J = \min \sum_{i=1}^{m} (\|\mathbf{x}_2^i - \widetilde{\mathbf{x}}_1^i\|^2 + \|\mathbf{x}_1^i - \widetilde{\mathbf{x}}_2^i\|^2) = \min_{R\in\mathbb{R}^{3\times3}} \sum_{i=1}^{m} (\|\mathbf{x}_2^i - KRK^{-1}\mathbf{x}_1^i\|^2 + \|\mathbf{x}_1^i - KR^TK^{-1}\mathbf{x}_2^i\|^2).$$

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 $\widetilde{R} = \widetilde{R}_{2,1}$ 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. $D^i = \begin{bmatrix} 0 & (\widehat{\mathbf{X}}_2^i\mathbf{X}_1^i)^T \\ \widehat{\mathbf{X}}_2^i\mathbf{X}_1^i & \mathbf{X}_2^i\mathbf{X}_1^{iT} + \mathbf{X}_1^i\mathbf{X}_2^{iT} - 2\mathbf{X}_2^{iT}\mathbf{X}_1^iI \end{bmatrix}$.
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

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 $\mathbf{k} = \begin{bmatrix} k_1 & k_2 & k_3 & k_4 & k_5 \end{bmatrix}^T$, we can use $\mathbf{k}$ to construct the calibration matrix K and the inverse of that K^{-1} as follows,

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

To find a common calibration matrix K to multiple pairs of images, we suppose that the rotation matrix $R_{j+1,j}, j = 1, 2, \ldots, n$ 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

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

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

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

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

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

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

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

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

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

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

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

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, filename)`: formats the output of the calibration matrix and rotation matrices to LaTeX.
- `show_iamge(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 ($\widehat{\ }$), 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 $\mathbf{k}$.
- `invert_K(k)`: constructs the inverse of the calibration matrix K^{-1} from the vector $\mathbf{k}$.
- `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 $\mathbf{k}$ 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.
- `print_matrix(output_dir)`: saves the rotation matrix $R_j, j = 1, 2, \ldots, 6$ and calibration matrix K to `.tex` files.
- `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

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

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

$$\begin{bmatrix} 0.9995 & 0.0000 & 0.0328 \\ 0.0000 & 1.0000 & -0.0003 \\ -0.0328 & 0.0003 & 0.9995 \end{bmatrix}.$$

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 or image 4 and 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. 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.

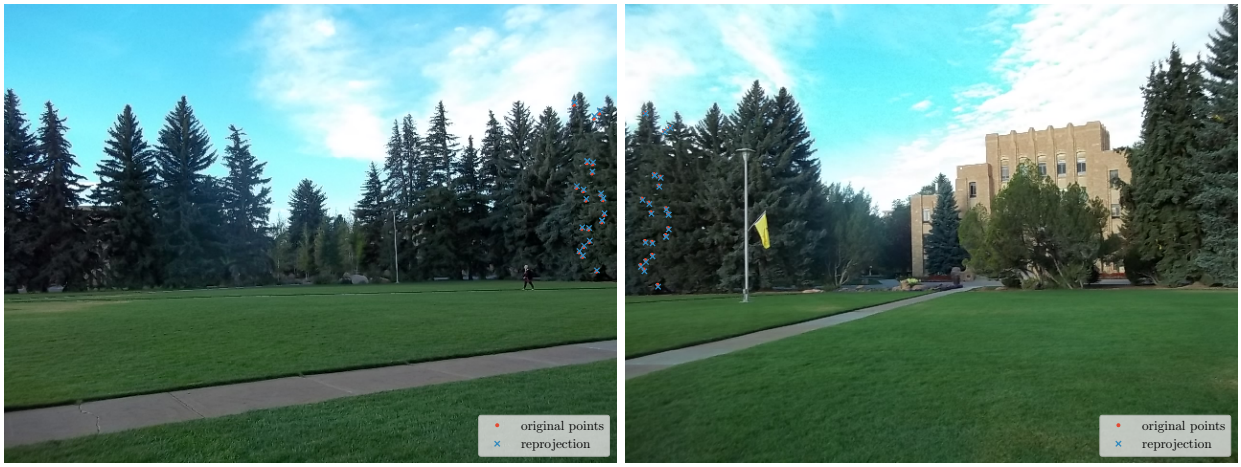

Figure 1: Reprojections of Correspondence Points Between Image 2 and Image 3

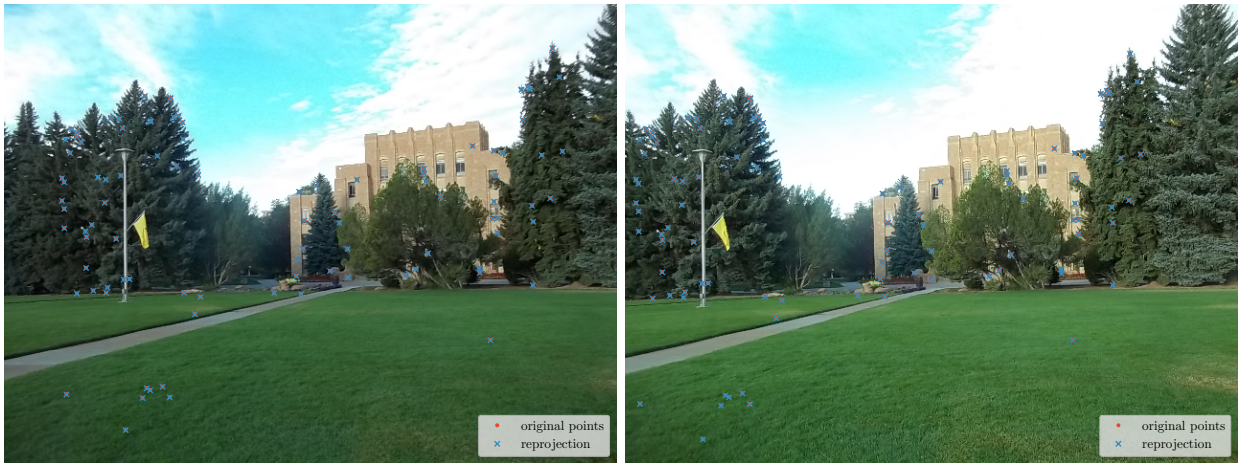

Figure 2: Reprojections of Correspondence Points Between Image 3 and Image 4

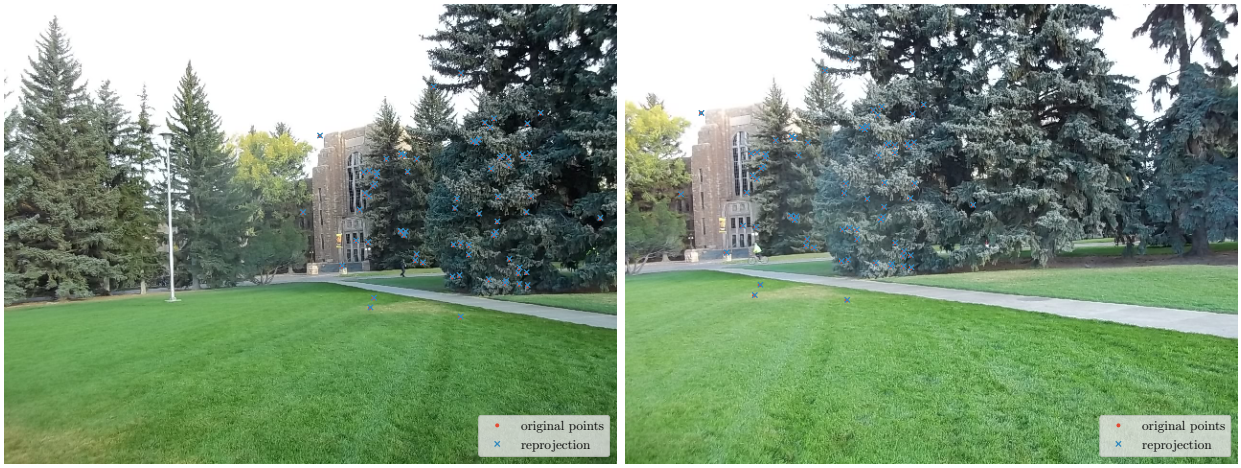

Figure 3: Reprojections of Correspondence Points Between Image 6 and Image 7

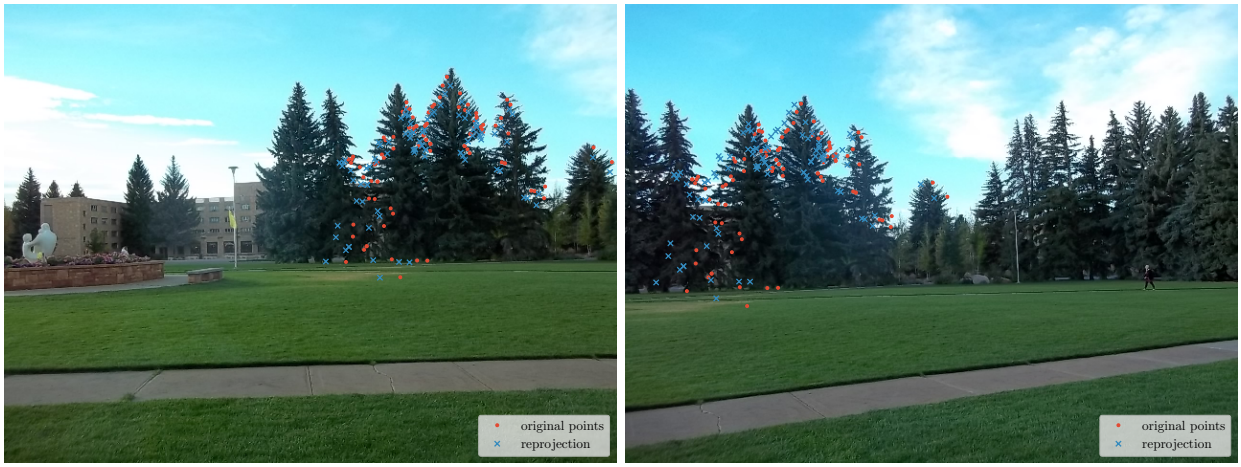

Figure 4: Reprojections of Correspondence Points Between Image 1 and Image 2

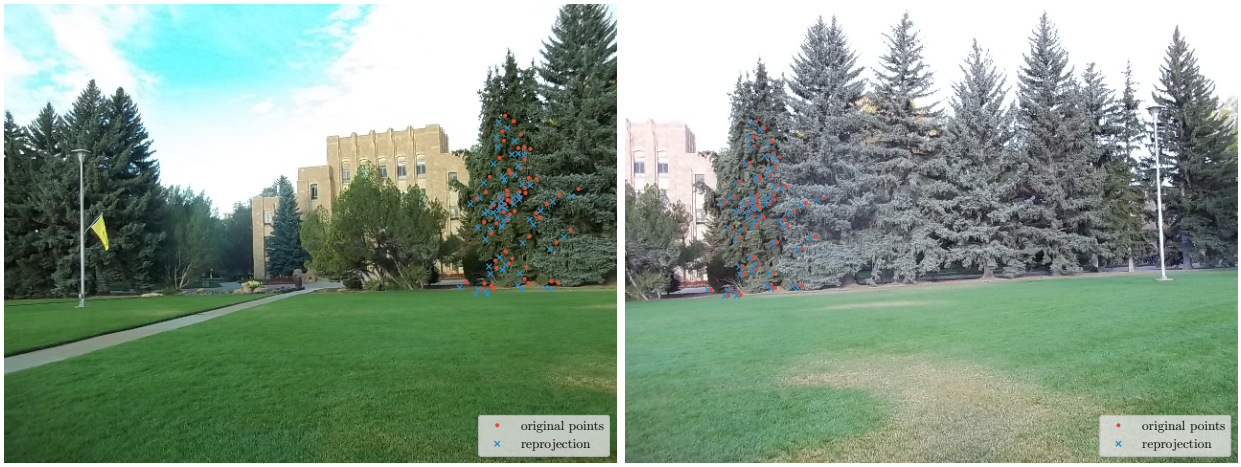

Figure 5: Reprojections of Correspondence Points Between Image 4 and Image 5

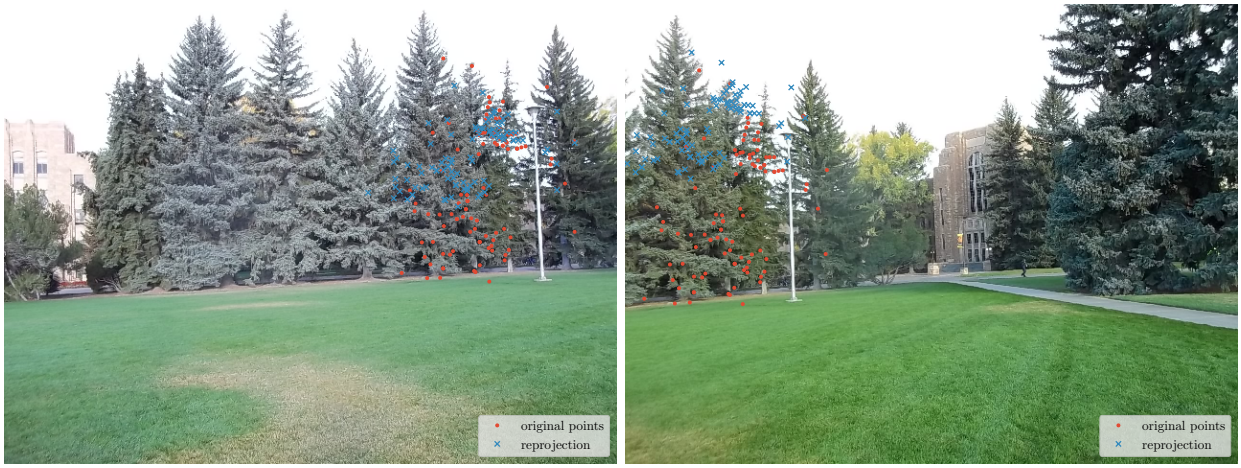

Figure 6: Reprojections of Correspondence Points Between Image 5 and Image 6

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

```
#! /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
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=888)

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

    def read_csv(self, output_dir):
        R_list = []
        for i in range(6):
            R = np.loadtxt('{:s}/R_{:d}.csv'.format(output_dir, i + 1),
                           delimiter=',')
            R_list.append(R)
        K = np.loadtxt('{:s}/K.csv'.format(output_dir), delimiter=',')
        return (R_list, K)

    def print(self, A, filename=''):
        nrow, ncol = A.shape
        if len(filename) == 0:
            print(r'\begin{equation*}')
            print(r'\begin{bmatrix}')
            for i in range(nrow):
                print(r'{:8.4f} & {:8.4f} & {:8.4f} \\'.format(
                    A[i, 0], A[i, 1], A[i, 2]))
            print(r'\end{bmatrix}')
            print(r'\end{equation*}')
            return

```

```
        with open(filename, 'w') as f:
            f.write('\\begin{equation*}\n')
            f.write('\\begin{bmatrix}\n')
            for i in range(nrow):
                f.write('{:8.4f} & {:8.4f} & {:8.4f} \\\\\n'.format(
                    A[i, 0], A[i, 1], A[i, 2]))
            f.write('\\end{bmatrix}.\n')
            f.write('\\end{equation*}\n')

    def show_image(self, I, x, xr, filename):
        with PdfPages(filename) as pdf:
            fig = plt.figure(frameon=False)
            ax = plt.Axes(fig, [0., 0., 1., 1.])
            ax.set_axis_off()
            fig.add_axes(ax)
            if I.ndim == 2:
                plt.imshow(I, cmap='gray')
            else:
                plt.imshow(I)
            h1, = plt.plot(x[0, :],
                           x[1, :],
                           '.',
                           markersize=4,
                           label='original points')
            h2, = plt.plot(xr[0, :],
                           xr[1, :],
                           'x',
                           markersize=4,
                           label='reprojection')
            plt.legend(handles=[h1, h2], loc='lower right')
            plt.grid()
            plt.axis([-0.5, 1279.5, 959.5, -0.5])
            plt.axis('off')
            plt.show(block=False)
            plt.savefig(filename.replace('pdf', 'png'), format='png', dpi=300)
            pdf.savefig(fig)
            plt.close()

    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, z, v):
        D = np.zeros((4, 4))
        D[0, 1:] = self.hat(z).dot(v).T
        D[1:, 0] = self.hat(z).dot(v).reshape(3)
        D[1:, 1:] = z.dot(v.T) + v.dot(z.T) - 2 * z.T.dot(v) * 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):
```

```
        D = np.zeros((4, 4))
        for i in range(len(x1)):
            z, v = x1[i].reshape(3, 1), x2[i].reshape(3, 1)
            D += self.D(z, v)
        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_k(self, x1, x2, tol, max_iter):
        R = self.find_R(x1, x2, np.random.rand(5, 1))
        K = lambda k: self.K(k)
        K_inv = lambda k: self.invert_K(k)
        x1_reproj = lambda R, k: np.array([
            np.linalg.norm(x2[:, i] - K(k).dot(R.T.dot(K_inv(k).dot(x2[:, i]))))
            for i in range(x2.shape[1])
        ]) / x2.shape[1]
        x2_reproj = lambda R, k: np.array([
            np.linalg.norm(x1[:, i] - K(k).dot(R.dot(K_inv(k).dot(x1[:, i]))))
            for i in range(x1.shape[1])
        ]) / x1.shape[1]
        J = lambda k: np.sum(x1_reproj(R, k)) + np.sum(x2_reproj(R, k))
        k = np.random.rand(5, 1)
        i = 0
        while J(k) > tol and i < max_iter:
            print('{:3d}: {:.4f}'.format(i, J(k)))
            R = self.find_R(x1, x2, k)
            J = lambda k: np.sum(x1_reproj(R, k)) + np.sum(x2_reproj(R, k))
            y = minimize(J, k.T)
            k = y['x'].reshape(5, 1)
            i += 1
        print('{:3d}: {:.4f}'.format(i, J(k)))
        return k

    def find_R(self, x1, x2, k):
        K_guess = self.K(k)
        K_inv_guess = np.linalg.inv(K_guess)
        X1 = [K_inv_guess.dot(x1[:, i]) for i in range(x1.shape[1])]
        X2 = [K_inv_guess.dot(x2[:, i]) for i in range(x2.shape[1])]
        R = self.optimal_quaternion(X1, X2)
        return R

    def find_R_K(self, x1, x2, tol, max_iter):
        k = self.find_k(x1, x2, tol, max_iter)
        R = self.find_R(x1, x2, k)
        K = self.K(k)
        return (R, K)

```

```
176     def find_all_R(self, x1_list, x2_list, tol, max_iter):
177         R_list = []
178         for i in range(len(x1_list)):
179             print('Finding R between Image {:d} and Image {:d}...'.format(
180                 i, i + 1))
181             R, K = self.find_R_K(x1_list[i], x2_list[i], tol, max_iter)
182             R_list.append(R)
183         return R_list
184 
185     def calibrate(self, x1_list, x2_list, tol, max_iter, output_dir):
186         R_list = self.find_all_R(x1_list, x2_list, tol, max_iter)
187         K = lambda k: self.K(k)
188         K_inv = lambda k: self.invert_K(k)
189         x1_reproj = lambda R, k, j: np.array([
190             np.linalg.norm(x2_list[j][:, i] - K(k).dot(
191                 R.T.dot(K_inv(k).dot(x2_list[j][:, i]))))
192             for i in range(x2_list[j].shape[1])
193         ]) / x2_list[j].shape[1]
194         x2_reproj = lambda R, k, j: np.array([
195             np.linalg.norm(x1_list[j][:, i] - K(k).dot(
196                 R.T.dot(K_inv(k).dot(x1_list[j][:, i]))))
197             for i in range(x1_list[j].shape[1])
198         ]) / x1_list[j].shape[1]
199         J = lambda k: np.sum([
200             np.sum(x1_reproj(R_list[j], k, j)) + np.sum(
201                 x2_reproj(R_list[j], k, j)) for j in range(len(x1_list))
202         ])
203         k = np.random.rand(5, 1)
204         print('Finding the calibration matrix K for all rotations...')
205         y = minimize(J, k)
206         k = y['x']
207         for i in range(len(R_list)):
208             R = R_list[i]
209             np.savetxt('{:s}/R_{:d}.csv'.format(output_dir, i + 1),
210                        R,
211                        delimiter=',',
212                        fmt='%15.8f',
213                        newline='\n')
214         np.savetxt('{:s}/K.csv'.format(output_dir, i + 1),
215                    self.K(k),
216                    delimiter=',',
217                    fmt='%15.8f',
218                    newline='\n')
219         return (R_list, self.K(k))
220 
221     def reproject(self, x, R=np.eye(3), K=np.eye(3)):
222         K_inv = np.linalg.inv(K)
223         x_reproj = K.dot(R.dot(K_inv.dot(x)))
224         return x_reproj
225 
226     def print_matrix(self, output_dir):
227         R_list, K = self.read_csv(output_dir)
228         for i in range(len(R_list)):
229             self.print(R_list[i], '{:s}/R_{:d}.tex'.format(output_dir, i + 1))
230         self.print(K, '{:s}/K.tex'.format(output_dir))
231 
232     def solve(self, mat_file, images, output_dir, tol=1, max_iter=200):
233         I = [img.imread(image) for image in images]
234         x1pMat, x2pMat = self.read_data(mat_file)
```

```
        ncols = []
        for i in range(len(I) - 1):
            for j in range(x1pMat[:, :, i].shape[1]):
                if sp.linalg.norm(x1pMat[:, j, i]
                                  ) == 0 or j == x1pMat[:, :, i].shape[1] - 1:
                    ncols.append(j)
                    break
        x1_list = [x1pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
        x2_list = [x2pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
        if (os.path.exists('{:s}/K.csv'.format(output_dir))):
            R_list, K = self.read_csv(output_dir)
        else:
            R_list, K = self.calibrate(x1_list, x2_list, tol, max_iter,
                                       output_dir)
        for i in range(len(R_list)):
            print('Rotation from Image {:d} to Image {:d}:'.format(
                i + 1, i + 2))
            print(R_list[i])
        print('Calibration matrix:')
        print(K)

        for i in range(len(I) - 1):
            I1, I2 = I[i].copy(), I[i + 1].copy()
            x1pmat = x1pMat[:, 0:ncols[i], i]
            x2pmat = x2pMat[:, 0:ncols[i], i]
            x1pReproMat = self.reproject(x1pmat, R_list[i], K)
            x2pReproMat = self.reproject(x2pmat, R_list[i], K)
            filename_1 = '{:s}/prexy{:d}_2.pdf'.format(output_dir, i + 1)
            filename_2 = '{:s}/prexy{:d}_1.pdf'.format(output_dir, i + 2)
            self.show_image(I1, x1pmat, x1pReproMat, filename_1)
            self.show_image(I2, x2pmat, x2pReproMat, filename_2)

    def solve_cv2(self, mat_file, images, output_dir, tol=16, max_iter=100):
        I = [cv2.imread(image) for image in images]
        x1pMat, x2pMat = self.read_data(mat_file)
        ncols = []
        for i in range(len(I) - 1):
            for j in range(x1pMat[:, :, i].shape[1]):
                if sp.linalg.norm(x1pMat[:, j, i]
                                  ) == 0 or j == x1pMat[:, :, i].shape[1] - 1:
                    ncols.append(j)
                    break
        x1_list = [x1pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
        x2_list = [x2pMat[:, 0:ncols[i], i] for i in range(len(ncols))]
        if (os.path.exists('{:s}/K.csv'.format(output_dir))):
            R_list, K = self.read_csv(output_dir)
        else:
            R_list, K = self.calibrate(x1_list, x2_list, tol, max_iter,
                                       output_dir)
        for i in range(len(R_list)):
            print('Rotation from Image {:d} to Image {:d}:'.format(
                i + 1, i + 2))
            print(R_list[i])
        print('Calibration matrix:')
        print(K)

        for i in range(len(I) - 1):
            I1, I2 = I[i].copy(), I[i + 1].copy()
            x1pmat = x1pMat[:, 0:ncols[i], i]
```

```
            x2pmat = x2pMat[:, 0:ncols[i], i]
            x1pReproMat = self.reproject(x1pmat, R_list[i], K)
            x2pReproMat = self.reproject(x2pmat, R_list[i], K)
            for j in range(0, x1pmat.shape[1]):
                cv2.putText(I1, "{}".format("X"),
                            (int(x1pmat[0, j]), int(x1pmat[1, j])),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.3, (0, 0, 255), 2)
                cv2.putText(I2, "{}".format("X"),
                            (int(x2pmat[0, j]), int(x2pmat[1, j])),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.3, (0, 0, 255), 2)
                cv2.putText(I1, "{}".format("O"),
                            (int(x1pReproMat[0, j]), int(x1pReproMat[1, j])),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.3, (255, 0, 0), 2)
                cv2.putText(I2, "{}".format("O"),
                            (int(x2pReproMat[0, j]), int(x2pReproMat[1, j])),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.3, (255, 0, 0), 2)
            cv2.imshow("Image 1 reproj (prexy{:d}.jpg)".format(i + 1), I1)
            cv2.imshow("Image 2 reproj (prexy{:d}.jpg)".format(i + 2), I2)
            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()


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, 1, 200)
    # c.solve_cv2(mat_file, images, output_dir, 1, 200)
    # c.print_matrix(output_dir)
'''
End of file
'''
```