Computer Vision Assignment 3

Nicolas Wicki

November 2021

2.1 Setup

No problems encountered.

2.2 Calibration

Approach: Given the correspondences, the (simplified) process of calibrating the camera consists of data normalization, estimating the projection matrix using DLT, optimizing based on the reprojection errors, and then decomposing it into camera intrinsics and pose. Question: Given the camera models that you saw in the lecture, which part of the full model is not calibrated with our approach? We do not consider the radial distortion introduced by the camera lens.

a) Data Normalization

The `impl.calib.geometry` module already provides normalization of 2D and 3D points. On one hand, we store the normalized points, but also keep track of the transformation matrices used for the denormalization on the other hand. Briefly explain the potential problems if we skip this step in our algorithm: Computing the fundamental matrix without normalizing the coordinates is quite unstable, since coordinates could be noisy with large variance.

b) DLT

Here, we build the projection constraint matrix. For each correspondence, we get two constraints, which are represented by rows in the constraint matrix. The projection matrix is represented by a (12,1) vector, as the solution vector is represented by the null space of the matrix V of the singular value decomposition. The constraints are given by the equality $x \times PX = 0$, where $\times$ is the cross product.

Assembling the constraint matrix How many independent constraints can you derive from a single 2D-3D correspondence? The constraint matrix $A \in \mathbb{R}^{3x12}$ for every correspondence is of rank 2 meaning that only two constraints (rows) are independent.

c) Optimizing reprojection errors

To compute the reprojection error, we convert our 3D points to homogeneous 3D coordinates and project it into the image plane using the estimated projection matrix P. After the projection, we normalize the result and compute the difference between the 2D point and the projected 3D point. How does the reported reprojection error change during optimization? Discuss the difference between algebraic and geometric error and explain the problem with the error measure $e = x \times PX$: The reprojection error decreases during optimization in a step-like fashion. However, the algebraic error has a completely different behavior. While the geometric error gradually decreases during optimization, the algebraic error changes arbitrarily in magnitude with no clear indication what direction to take. This might be obvious when looking at the geometric meaning of the cross product. The coefficients of the resulting perpendicular vector do not get gradually smaller as the angle between the two vector operands gets smaller. In optimization procedures, we often gradually improve our result, which is why we need to choose a fitting loss function to indicate the proper path to take during our optimization journey.

d) Denormalizing the projection matrix

Denormalizing the optimized projection matrix is (actually) quite easy. Simply compute the following: $P = T^{-1}PU$ where T is the normalizing matrix for the 2D points and U is the normalizing matrix for the 3D points. (However, for some reason I forgot to invert the 2D transformation matrix, which certainly cost me some debugging time :(.)

e) Decomposing the projection matrix

Decomposing the projection matrix is a bit more involved. First, extract the 3×3 left sub-matrix M from our projection matrix P of shape 3×4. Then, we compute the QR decomposition of M^{-1} to get R^{-1} and K^{-1}. R is a rotation matrix, which means that the determinant should be equal to 1. So, if it is negative, we can invert it. If we invert it, we need to invert K too, as KR must equal M. Then, K should have a positive diagonal as it is the length of the focal lens. This can be done by the transformation matrix $T = diag(sign(diag(K)))$. To still uphold the equality to M, we multiply R by T^{-1} ($K = KT, R = T^{-1}R$). To compute the camera center C, we compute the SVD of the projection matrix P and get the null space of V (The null space is in homogeneous coordinates, which is why we need to normalized it.). The translation t can then easily be computed by $-RC$. *Do your estimated values seem reasonable?* They do seem reasonable indeed. The camera is positioned correctly, and the 3D points are projected into the image plane with only a small error (see Figure 1).

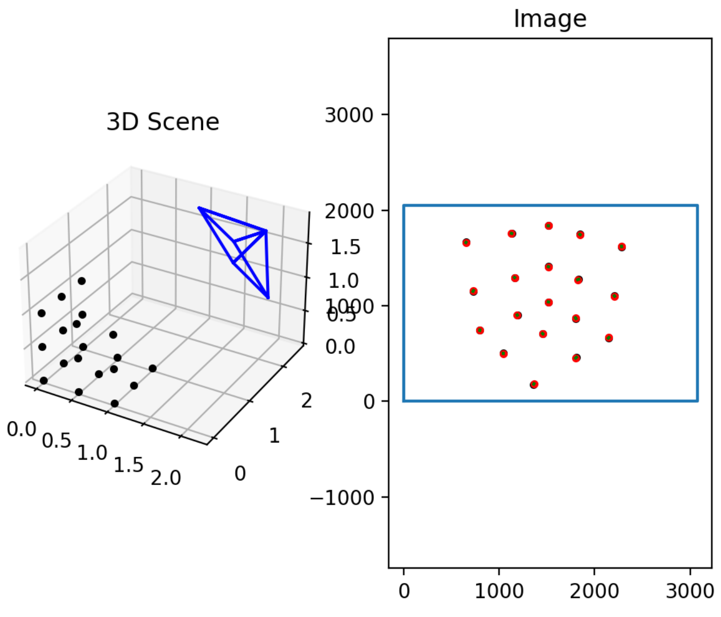

Figure 1: Result from `calibration.py`

2.3 Structure From Motion

a) Essential matrix estimation

During the step of computing the essential matrix, we first need to normalize the key-points from our two images (divide each point by its length) and project the points into the image plane using the inverted camera matrix K. Then, we need to define the constraint matrix. The constraint for each match of key-point pairs x_1 and x_2 is defined as follows: $x_1^T E x_2 = 0$, where E is the essential matrix. To actually set the constraint in the constraint matrix to fit the 9-dimensional vector representing the essential matrix E, we can just compute the result of the above constraint:

$$x_1^T E x_2 = 0 \tag{1}$$
$$E = [[e_1, e_2, e_3], [e_4, e_5, e_6], [e_7, e_8, e_9]] \tag{2}$$
$$x_1 = [x_{11}, x_{12}, x_{13}]^T \tag{3}$$
$$x_2 = [x_{21}, x_{22}, x_{23}]^T \tag{4}$$
$$x_1^T E = \tag{5}$$
$$[x_{11}e_1 + x_{12}e_4 + x_{13}e_7, \tag{6}$$
$$x_{11}e_2 + x_{12}e_5 + x_{13}e_8, \tag{7}$$
$$x_{11}e_3 + x_{12}e_6 + x_{13}e_9] = \tag{8}$$
$$[(x_1^T E)_1, (x_1^T E)_2, (x_1^T E)_3] \tag{9}$$
$$x_1^T E x_2 = \tag{10}$$
$$[(x_1^T E)_1, (x_1^T E)_2, (x_1^T E)_3] * \tag{11}$$
$$[x_{21}, x_{22}, x_{23}]^T = \tag{12}$$
$$x_{21} * (x_1^T E)_1 + x_{22} * (x_1^T E)_2 + x_{23} * (x_1^T E)_3 = 0 \tag{13}$$

Now, we just need to map the result 1-to-1 to code. To fulfill the internal constraints of the essential matrix, we simply compute the singular value decomposition, and set the diagonal of the singular value matrix to ones with a trailing zero ([1,1,0]). We recompute the essential matrix from its decomposition, which completes the estimation.

b) Point Triangulation

Here, we implement the point triangulation function where we simply need to filter all points from the 3D points, which lie behind our camera (simply checking whether the last coordinate is negative suffices). Not only do we need to remove those 3D points, but also the corresponding indices from the key-points of each image.

c) Finding the correct decomposition

Our essential matrix was decomposed into four possible relative poses (combinations for rotation matrices and translations). To find the correct one, we check which pose applied to our camera delivers the most points in front of it (Use an arbitrary pose for the first image, one of the four possible poses for the second image to compute the 3D points. Repeat for every possible relative pose). Then, we set the pose of the first image to an arbitrary pose and the pose of the second image to the relative pose with the most points in front of the camera.

d) Absolute pose estimation

As mentioned in the assignment description, no additional implementation is needed here.

e) Map extension

Here, we iterate over each image. We estimate the pose of each image using the same type of constraint matrix as for the calibration task. With this pose, we can compute the 2D to 3D point correspondences between the current image and each already registered image. We accumulate all 3D points computed, the 2D indices for each registered image, and the 2D indices of the image in question in a correspondence dictionary. We reuse that dictionary in the update reconstruction state function to elevate the indices of the 3D points to the global index level and add the new 2D-to-3D correspondences for each image. To see a visualization of the result, consider Figure 2 or Figure 3 with no cameras showing.

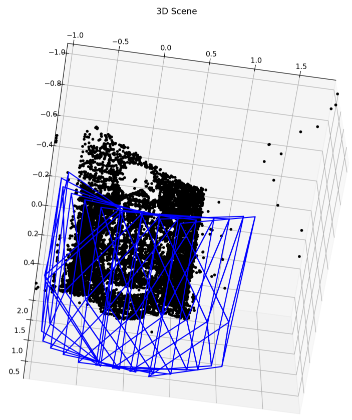

Figure 2: Result from `sfm.py`

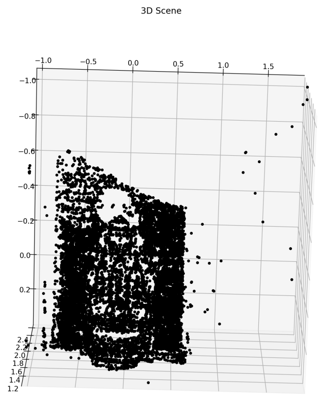

Figure 3: Result from `sfm.py` with no cameras