TTK4255 Robotic Vision, Spring 2020

Exercise 6: Multiple View Geometry

Authored by Simen Haugo (simen.haugo@ntnu.no)

Due: See Blackboard

This exercise is a mix of programming problems and theory questions. You are free to use any programming language, but solutions will be provided in Python and Matlab, and we may not be able to help you effectively if you use a different language. We therefore recommend that you use either Python or Matlab. You should also do exercise 0 to get familiar with language features and libraries that are useful for the exercises.

Instructions

This exercise is **approved / not approved**. To get your exercise approved, submit your answers to the questions as a **PDF** to Blackboard. You **don't** have to submit your code. Feel free to use LaTeX, Word, handwritten scans, or any other tool to write your answers.

Getting help

If you have questions about the exercise, you can:

- Post a question on Piazza (you can post anonymously if you want)
- Ask for help during the tutorial sessions (see Piazza for time and place)

Primary resources

- Lecture 7
- (HZ) Hartley and Zissermann, *Multiple View Geometry in Computer Vision.*
  - Ch. 9.1: Epipolar geometry
  - Ch. 9.2: The fundamental matrix
  - Ch. 9.6: The essential matrix
  - Ch. 10.2: Reconstruction ambiguity
  - Ch. 11.1: Basic equations
  - Ch. 11.2: The normalized 8-point algorithm
  - Ch. 12.2: Linear triangulation methods

Overview

In this exercise you will implement a two-view structure-from-motion (SfM) method, which simultaneously recovers the 3D structure and camera motion from image point correspondences. The method is broken down into the following steps:

1. Estimate the fundamental matrix $\mathbf{F}$ using the normalized 8-point algorithm.
2. Recover the essential matrix $\mathbf{E}$ from $\mathbf{F}$ and decompose it into the camera motion.
3. Use the recovered motion to triangulate the 3D coordinate of image point pairs.

This exercise is slightly longer than the others, so the deadline has been extended to allow you to make an attempt at all of the tasks.

Provided data and code

We have included a script for each task. These take care of reading in the images, making calls to functions you will be implementing, and generating the necessary figures. The zip includes the following data

- `data/im*.png`: Two selected images from the “Temple” data set in the Middlebury Multi-View Stereo benchmark.
- `data/K*.txt`: Camera intrinsic matrix for image 1 and image 2.
- `data/matches.txt`: An array of good point correspondences. Each row contains a pair of 2D pixel coordinates $(u_1, v_1) \leftrightarrow (u_2, v_2)$ between `im1` and `im2`, that correspond to the same 3D point in space.
- `data/goodPoints.txt`: An additional array of points in `im1` that should be easy to localize in the second image.

$\boxed{1}$ Data normalization

In the task after this one, you will implement the 8-point algorithm for estimating the fundamental matrix. Unfortunately, the original algorithm can be quite inaccurate. The *normalized* 8-point algorithm is a widely used alternative that applies a normalizing transformation to the input points, which dramatically improves the performance.

A common normalization is a translation and scaling of the points so that their centroid is at the origin and their average distance from the origin is $\sqrt{2}$. That is, if $(u_i, v_i), i = 1...n$, are the input points, then the normalized points $(\hat{u}_i, \hat{v}_i)$ satisfy the following:

$$\sum_{i=1}^{n} \hat{u}_i = \sum_{i=1}^{n} \hat{v}_i = 0 \quad \text{and} \quad \frac{1}{n}\sum_{i=1}^{n} \sqrt{\hat{u}_i^2 + \hat{v}_i^2} = \sqrt{2} \tag{1}$$

(a) Find an expression for a 3×3 homogeneous matrix $\mathbf{T}$ such that

$$\begin{bmatrix} \hat{u}_i \\ \hat{v}_i \\ 1 \end{bmatrix} = \mathbf{T} \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} \tag{2}$$

(b) Implement the function `normalize_points` that computes and returns $\mathbf{T}$ along with the normalized points. Test your implementation by running the `task1` script.

$\boxed{2}$ The normalized 8-point algorithm

The 8-point algorithm was originally invented for estimating the essential matrix $\mathbf{E}$ in the case of two views with calibrated cameras. It was later discovered that the same algorithm can be used in the uncalibrated setting. The matrix derived in this case is called the fundamental matrix $\mathbf{F}$. In either case, the basic equation used to derive the 8-point algorithm is the epipolar constraint:

$$(\tilde{\boldsymbol{u}}')^T \mathbf{F} \tilde{\boldsymbol{u}} = 0 \tag{3}$$

which holds for any pair of matching points $\tilde{\boldsymbol{u}} = (u, v, 1), \tilde{\boldsymbol{u}}' = (u', v', 1)$ in two images. The 8-point algorithm is derived from this in a similar manner as the DLT algorithm in exercise 3. First, label the entries of $\mathbf{F}$ as

$$\mathbf{F} = \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \tag{4}$$

and form a 9×1 vector containing the entries of $\mathbf{F}$ in row-major order

$$\mathbf{f} = \begin{bmatrix} F_{11} & F_{12} & F_{13} & F_{21} & F_{22} & F_{23} & F_{31} & F_{32} & F_{33} \end{bmatrix}^T \tag{5}$$

Then we can write the epipolar constraint for a single point pair as

$$\begin{bmatrix} u'u & u'v & u' & v'u & v'v & v' & u & v & 1 \end{bmatrix} \mathbf{f} = 0 \tag{6}$$

This is one linear equation in the unknown entries of $\mathbf{F}$. By stacking the equations from n point matches, we can build a system of linear equations of the form

$$\mathbf{A}\mathbf{f} = \begin{bmatrix} u_1'u_1 & u_1'v_1 & u_1' & v_1'u_1 & v_1'v_1 & v_1' & u_1 & v_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_n'u_n & u_n'v_n & u_n' & v_n'u_n & v_n'v_n & v_n' & u_n & v_n & 1 \end{bmatrix} \mathbf{f} = 0 \tag{7}$$

Given $n \geq 8$ point matches, the desired solution $\mathbf{f}$ can be obtained from the singular value decomposition (SVD) of $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ as the column of $\mathbf{V}$ corresponding to the smallest singular value. The fundamental matrix $\mathbf{F}$ can then be obtained by reshaping the vector back into a 3×3 matrix.

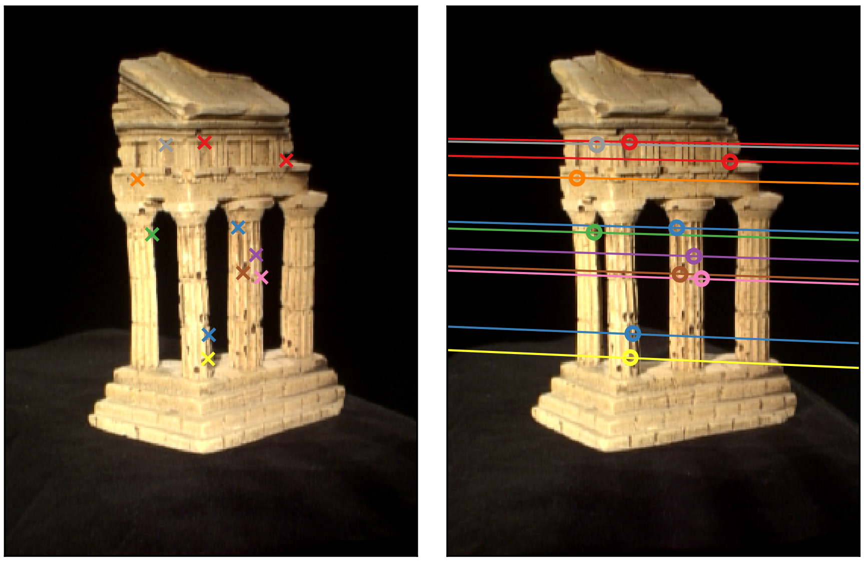

Figure 1: Example result for task (c). For a given point in the left image, the corresponding point in the right image lies somewhere on the epipolar line defined by the fundamental matrix.

(a) Implement the function `eight_point` which estimates the fundamental matrix from n point pairs $(u_i, v_i) \leftrightarrow (u'_i, v'_i)$ using the above procedure. You can test your function by running the `task2` script, which draws the epipolar lines as in Fig. 1. Your results should look similar, but not identical.

(b) As discussed in HZ Ch. 11.1.1 (p.280), the matrix obtained above may not satisfy the properties of a fundamental matrix. Implement `closest_fundamental_matrix` which returns the closest fundamental matrix to a given 3×3 matrix using the method described in the book. Modify `eight_point` to use this method.

(c) Modify `eight_point` to use the normalization from task 1. As described in HZ Ch. 11.2 (p.282), you should use normalized point coordinates to build the $\mathbf{A}$ matrix, and denormalize the matrix $\mathbf{F}$ after enforcing the fundamental constraint in (b).

You should now be able to get a very good estimate. Run `task2` again and **include the figure in your submission.**

$\boxed{3}$ Triangulation

Triangulation is the problem of determining a point's 3D position from a pair of matching image coordinates and known camera poses. In this task you will implement the linear triangulation method described in HZ Ch. 12.2 (p.312), summarized below.

The method starts with the equations for the projection of a 3D point $\mathbf{X}$ into the two views, which can be written using homogeneous coordinates as

$$\tilde{\boldsymbol{u}} = \mathbf{P}\tilde{\mathbf{X}} \quad \text{and} \quad \tilde{\boldsymbol{u}}' = \mathbf{P}'\tilde{\mathbf{X}} \tag{8}$$

Here, $\mathbf{P}$ and $\mathbf{P}'$ is the 3×4 camera projection matrix for the first and second view, which in general has the form $\mathbf{P} = \mathbf{K}[\mathbf{R} \quad \mathbf{t}]$, where $\mathbf{K}$ is the camera intrinsics and $\mathbf{R}$, $\mathbf{t}$ is the rotation and translation from world to camera coordinates. The actual image coordinate is found by dividing by the third component:

$$u = \frac{\boldsymbol{p}_1^T\tilde{\mathbf{X}}}{\boldsymbol{p}_3^T\tilde{\mathbf{X}}} \quad \text{and} \quad v = \frac{\boldsymbol{p}_2^T\tilde{\mathbf{X}}}{\boldsymbol{p}_3^T\tilde{\mathbf{X}}} \tag{9}$$

where $\boldsymbol{p}_i$ is the i'th row of $\mathbf{P}$. We can write the same for u', v', giving a total of four equations. Multiplying both sides by the denominator and collecting terms, we can write these as a homogeneous system of equations

$$\mathbf{A}\tilde{\mathbf{X}} = \begin{bmatrix} u\boldsymbol{p}_3^T - \boldsymbol{p}_1^T \\ v\boldsymbol{p}_3^T - \boldsymbol{p}_2^T \\ u'\boldsymbol{p}_3'^T - \boldsymbol{p}_1'^T \\ v'\boldsymbol{p}_3'^T - \boldsymbol{p}_2'^T \end{bmatrix} \begin{bmatrix} \tilde{X} \\ \tilde{Y} \\ \tilde{Z} \\ \tilde{W} \end{bmatrix} = 0 \tag{10}$$

which can be solved using the SVD.

(a) Implement the function `linear_triangulation`, which takes in a pair of projection matrices $\mathbf{P}, \mathbf{P}'$ and a single point correspondence $(u, v) \leftrightarrow (u', v')$, and computes the non-homogeneous 3D point $\mathbf{X}$ using the linear triangulation method.

(b) Before we can triangulate any point correspondences we first need the camera matrices $\mathbf{P}$ and $\mathbf{P}'$. Since the camera is calibrated, we already know the intrinsic matrix for both views. The book (HZ Ch. 9.6) shows how to recover the rotation and translation between the two views from the essential matrix. We provide `motion_from_essential` which implements this, but requires the essential matrix.

Implement `essential_from_fundamental`, which computes the essential matrix from the fundamental matrix using the equations in HZ Ch. 9.6 (p.257).

(c) Implement `camera_matrices` which computes the camera matrices $\mathbf{P}$ and $\mathbf{P}'$, given the rotation and translation between the two views and the camera intrinsics. You can arbitrarily choose the first view as the reference frame, which means that $\mathbf{P} = \mathbf{K}[\mathbf{I} \quad \mathbf{0}]$ and $\mathbf{P}' = \mathbf{K}'[\mathbf{R} \quad \mathbf{t}]$.

(d) The essential matrix gives four possible solutions for the relative motion. We need to choose the right one. Fortunately, in only one of the solutions will the triangulated points $\mathbf{X}$ actually be in front of both cameras (HZ Ch. 9.6.3), meaning they have a positive Z coordinate in both camera frames.

Implement the function `choose_solution` which chooses the solution that gives the most points visible in both cameras. Do this by iterating over each potential solution $\mathbf{R}, \mathbf{t}$ and forming the camera matrices $\mathbf{P}, \mathbf{P}'$. Use this to triangulate (at least one) of the point correspondences, and testing if the 3D point is visible in *both* views.

(e) We can now combine everything to create a 3D reconstruction using our point correspondences. You should now be able to run the `task3` script. This will triangulate all the point correspondences and show the 3D reconstruction.

Include a figure of the 3D reconstruction in your submission.

4 Epipolar matching

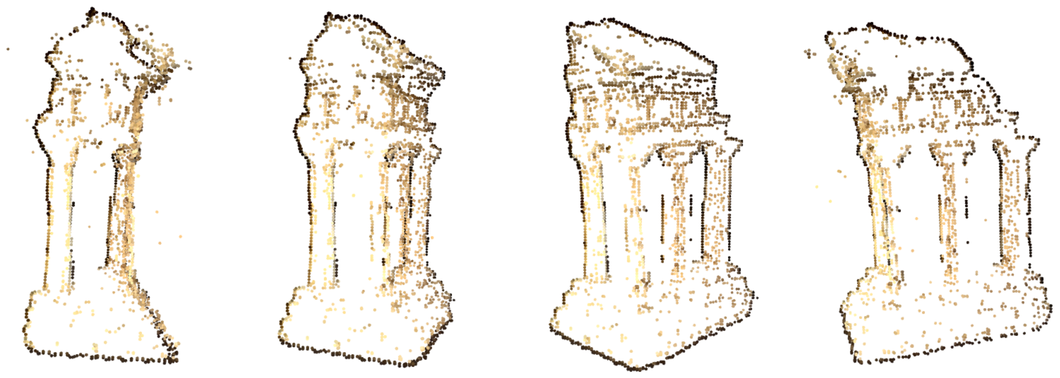
Figure 2: Example 3D reconstruction using the steps described in this exercise.

To reconstruct a 3D scene we may want a larger number of points. We could obtain these by detecting and matching feature descriptors in both images, but the scene may not have enough points with unique appearance. However, once we have the fundamental matrix, we can reduce the point correspondence problem to a search along epipolar lines. This greatly reduces the potential matches a given point in the first image can have. We will now use this to reconstruct additional points provided in `data/goodPoints.txt`.

Iterating over an epipolar line

Recall that the fundamental matrix transforms a point in one image to a line in the other. In particular, if $(\tilde{\boldsymbol{u}}')^T\mathbf{F}\tilde{\boldsymbol{u}} = 0$, then $\boldsymbol{l} = \mathbf{F}\tilde{\boldsymbol{u}} = (l_1, l_2, l_3)^T$ is the line in the second image corresponding to the point (u, v) in the first image. A point (u', v') in the second image is on this line if it satisfies the equation

$$u'l_1 + v'l_2 + l_3 = 0 \tag{11}$$

Given a query point (u, v) in the first image, this then lets us iterate over the corresponding line in the second image by iterating over either u' or v' in an appropriate range, and solving the line equation for the other coordinate.

Finding similar pixel regions

For a given pair of potentially matching image points $(u, v) \leftrightarrow (u', v')$, we can compute a similarity score by comparing the pixel values in a small window around both points. There are many similarity measures to choose from. The simplest is the sum of squared intensity differences (SSD), or the sum of absolute intensity difference (SAD), both defined for grayscale images. Corke's book lists several others in Table 12.1.

Once you have chosen a similarity measure, you can determine the best match by iterating over the epipolar line and finding the point at which the similarity score is best (e.g. smallest, if using SSD or SAD).

(a) Implement `epipolar_match` which, for each point (u, v) in image 1, finds the best matching point (u', v') in image 2 along the epipolar line.

Run `task4` to test your implementation on the provided true matches. Your found matches should ideally all coincide with the true matches. Your function does not need to be perfect, but it should get most easy points correct, like corners, etc.

(b) Modify `task3` to use the additional point data `data/goodPoints.txt` (in image 1) for triangulation. You should still use the original matched pairs to compute the camera matrices as before. After that you should load the additional points, find correspondences by epipolar matching, and triangulate each pair.

Include a figure of your final 3D reconstruction in your submission.

Optional task (Not required to get exercise approved)

(c) Colorize your point cloud! See the documentation for `scatter3` for how to pass in an array of RGB values. Use the RGB value at $I_1(v_1, u_1)$ for each (u_1, v_1) in the `goodPoints` array.