Project 3: Camera Calibration and Fundamental Matrix Estimation with RANSAC

CS 4476/6476

Spring 2022

Brief

- Due: Check Canvas for up to date information
- Project materials including report template: GitHub
- Hand-in: through Gradescope
- Required files: `<your_gt_username>.zip`, `<your_gt_username>_proj3.pdf`

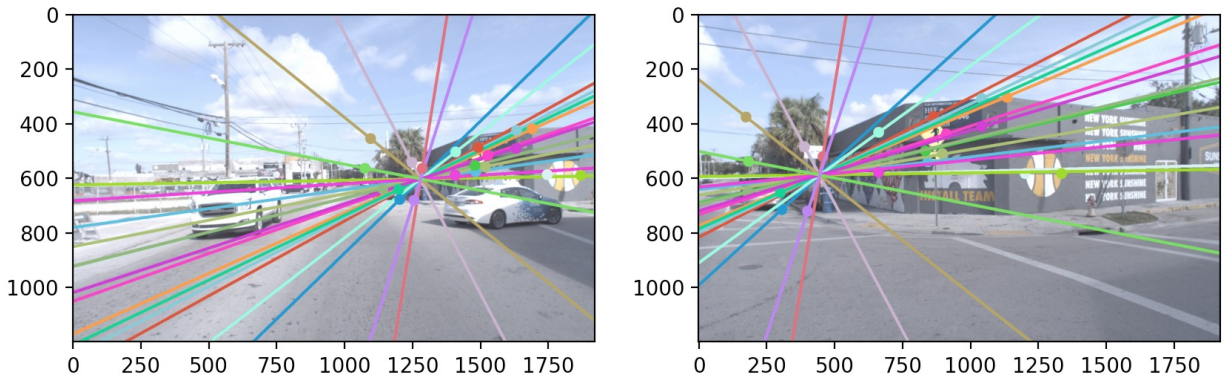

Figure 1: An autonomous vehicle makes a right turn and captures two images a moment apart. The first image happens to contain an identical autonomous vehicle ahead of it. Epipolar lines show the camera locations given corresponding points in two views of a scene. Corresponding points in the two images are marked by circles of the same color. The two vehicles have the same hardware, and note that the epipole in the left image is located at the same height as the camera mounted on the other vehicle.

Overview

The goal of this project is to introduce you to camera and scene geometry. Specifically we will estimate the camera projection matrix, which maps 3D world coordinates to image coordinates, as well as the fundamental matrix, which relates points in one scene to epipolar lines in another. The camera projection matrix and fundamental matrix can each be estimated using point correspondences. To estimate the projection matrix (camera calibration), the input is corresponding 3D and 2D points. To estimate the fundamental matrix the input is corresponding 2D points across two images. You will start out by estimating the projection matrix and the fundamental matrix for a scene with ground truth correspondences. Then you will move on to estimating the fundamental matrix using point correspondences that are obtained using SIFT.

Remember these challenging images of Gaudi's Episcopal Palace from project 2? By using RANSAC to find the fundamental matrix with the most inliers, we can filter away spurious matches and achieve near perfect point-to-point matching as shown below:

Figure 2: Gaudi's Episcopal Palace.

Setup

1. Check https://github.gatech.edu/cs4476/project-3 for environment installation.
2. Run the notebook using `jupyter notebook ./project-3.ipynb`
3. After implementing all functions, ensure that all sanity checks are passing by running `pytest tests` inside the main folder.
4. Generate the zip folder for the code portion of your submission once you've finished the project using `python zip_submission.py --gt_username <your_gt_username>`

1 Part 1: Camera projection matrix

Introduction

The goal is to compute the projection matrix that goes from world 3D coordinates to 2D image coordinates. Recall that using homogeneous coordinates the equation for moving from 3D world to 2D camera coordinates is:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \cong \begin{pmatrix} u * s \\ v * s \\ s \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{1}$$

Another way of writing this equation is:

$$u = \frac{m_{11}X + m_{12}Y + m_{13}Z + m_{14}}{m_{31}X + m_{32}Y + m_{33}Z + m_{34}} \tag{2}$$

$$\rightarrow (m_{31}X + m_{32}Y + m_{33}Z + m_{34})u = m_{11}X + m_{12}Y + m_{13}Z + m_{14}$$

$$\rightarrow 0 = m_{11}X + m_{12}Y + m_{13}Z + m_{14} - m_{31}uX - m_{32}uY - m_{33}uZ - m_{34}u$$

$$v = \frac{m_{21}X + m_{22}Y + m_{23}Z + m_{24}}{m_{31}X + m_{32}Y + m_{33}Z + m_{34}} \tag{3}$$

$$\rightarrow (m_{31}X + m_{32}Y + m_{33}Z + m_{34})v = m_{21}X + m_{22}Y + m_{23}Z + m_{24}$$

$$\rightarrow 0 = m_{21}X + m_{22}Y + m_{23}Z + m_{24} - m_{31}vX - m_{32}vY - m_{33}vZ - m_{34}v$$

At this point, you're almost able to set up your linear regression to find the elements of the matrix M. There's only one problem–the matrix M is only defined up to a scale. Therefore, these equations have many different possible solutions (in particular M = all zeros is a solution, which is not very helpful in our context). The way around this is to first fix a scale, and then do the regression. There are several options for doing this: 1) You can fix the last element, $m_{34} = 1$, and then find the remaining coefficients, or 2) you can use the singular value decomposition to directly solve the constrained optimization problem:

$$\begin{array}{ll} \arg\min\limits_{x} & \|Ax\| \\ \text{s.t.} & \|x\| = 1 \end{array} \tag{4}$$

To make sure that your code is correct, we are going to give you a set of "normalized points" in the files `pts2d-norm-pic_a.txt` and `pts3d-norm.txt`. If you solve for M using all the points, you should get a matrix that is a scaled equivalent of the following:

$$M_{\text{norm}A} = \begin{pmatrix} -0.4583 & 0.2947 & 0.0139 & -0.0040 \\ 0.0509 & 0.0546 & 0.5410 & 0.0524 \\ -0.1090 & -0.1784 & 0.0443 & -0.5968 \end{pmatrix}$$

For example, this matrix will take the last normalized 3D point, $< 1.2323, 1.4421, 0.4506, 1.0 >$, and project it to $< u, v >$ of $< 0.1419, 0.4518 >$, converting the homogeneous 2D point $< us, vs, s >$ to its in homogeneous version (the transformed pixel coordinate in the image) by dividing by s.

First, you will need to implement the least squares regression to solve for M given the corresponding normalized points. The starter code will load 20 corresponding normalized 2D and 3D points. You have to write the code to set up the linear system of equations, solve for the unknown entries of M, and reshape it into the estimated projection matrix. To validate that you've found a reasonable projection matrix, we've provided evaluation code which computes the total "residual" between the projected 2D location of each 3D point and the actual location of that point in the 2D image. The residual is just the distance (square root of the sum of squared differences in u and v). This should be very small.

Once you have an accurate projection matrix M, it is possible to tease it apart into the more familiar and more useful matrix K of intrinsic parameters and matrix $[R|T]$ of extrinsic parameters. For this project we will only ask you to estimate one particular extrinsic parameter: the camera center in world coordinates. Let us define M as being composed of a 3×3 matrix, Q, and a 4th column, m_4:

$$M = \begin{bmatrix} Q \mid m_4 \end{bmatrix} \tag{5}$$

From class we said that the center of the camera C could be found by:

$$C = -Q^{-1}m_4 \tag{6}$$

To debug your code, if you use you the normalized 3D points to get the M given above, you would get a camera center of:

$$C_{\text{norm}A} =< -1.5125, -2.3515, 0.2826 >$$

We've also provided a visualization which will show the estimated 3D location of the camera with respect to the normalized 3D point coordinates.

In `part1_projection_matrix.py`, you will implement the following:

- `projection()`: Projects homogeneous world coordinates $[X, Y, Z, 1]$ to non-homogeneous image coordinates (u, v). Given projection matrix M, the equations that accomplish this are (2) and (3).
- `calculate_projection_matrix()`: Solves for the camera projection matrix using a system of equations set up from corresponding 2D and 3D points.
- `calculate_camera_center()`: Computes the camera center location in world coordinates.

2 Part 2: Fundamental matrix

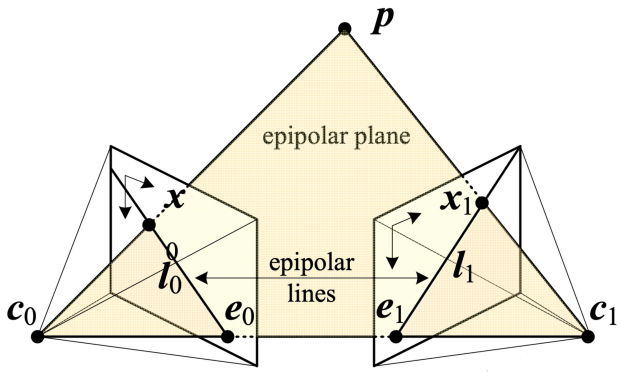

Figure 3: Two-camera setup. Reference: Szeliski, p. 682.

The next part of this project is estimating the mapping of points in one image to lines in another by means of the fundamental matrix. This will require you to use similar methods to those in part I. We will make use of the corresponding point locations listed in `pts2d-pic_a.txt` and `pts2d-pic_b.txt`. Recall that the definition of the fundamental matrix is:

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = 0 \tag{7}$$

for a point $(u, v, 1)$ in image A, and a point $(u', v', 1)$ in image B. See Appendix A for the full derivation. Note: the fundamental matrix is sometimes defined as the transpose of the above matrix with the left and right image points swapped. Both are valid fundamental matrices, but the visualization functions in the starter code assume you use the above form.

Another way of writing this matrix equations is:

$$\begin{pmatrix} u' & v' & 1 \end{pmatrix} \begin{pmatrix} f_{11}u + f_{12}v + f_{13} \\ f_{21}u + f_{22}v + f_{23} \\ f_{31}u + f_{32}v + f_{33} \end{pmatrix} = 0 \tag{8}$$

Which is the same as:

$$\left(f_{11}uu' + f_{12}vu' + f_{13}u' + f_{21}uv' + f_{22}vv' + f_{23}v' + f_{31}u + f_{32}v + f_{33}\right) = 0 \tag{9}$$

Starting to see the regression equations? Given corresponding points you get one equation per point pair. With 8 or more points you can solve this (why 8?). Similar to part I, there's an issue here where the matrix is only defined up to scale and the degenerate zero solution solves these equations. So you need to solve using the same method you used in part I of first fixing the scale and then solving the regression.

The least squares estimate of F is full rank; however, a proper fundamental matrix is a rank 2. As such we must reduce its rank. In order to do this, we can decompose F using singular value decomposition into the

matrices $U\Sigma V' = F$. We can then construct a rank 2 matrix by setting the smallest singular value in Σ to zero thus generating Σ_2 . The fundamental matrix is then easily calculated as $F = U\Sigma_2 V'$. You can check your fundamental matrix estimation by plotting the epipolar lines using the plotting function provided in the starter code.

Coordinate normalization

As discussed in lecture, your estimate of the fundamental matrix can be improved by normalizing the coordinates before computing the fundamental matrix (see [1] by Hartley). It is suggested for this project you perform the normalization through linear transformations as described below to make the mean of the points zero and the average magnitude 1.0.

$$\begin{pmatrix} u' \\ v' \\ 1 \end{pmatrix} = \begin{pmatrix} s_u & 0 & 0 \\ 0 & s_v & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -c_u \\ 0 & 1 & -c_v \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{10}$$

The transform matrix T is the product of the scale and offset matrices. c_u and c_v are the mean coordinates. To compute a scale s you could estimate the standard deviation after subtracting the means. Then the scale factor s would be the reciprocal of whatever estimate of the scale you are using. You could use one scale matrix based on the statistics of the coordinates from both images or you could do it per image.

In `part2_fundamental_matrix.py` you will need to use the scaling matrices to adjust your fundamental matrix so that it can operate on the original pixel coordinates. This is performed as follows:

$$F_{orig} = T_b^T * F_{norm} * T_a \tag{11}$$

In `part2_fundamental_matrix.py`, you will implement the following:

- `normalize_points()`: Normalizes the 2D coordinates.
- `unnormalize_F()`: Adjusts the fundamental matrix to account for the normalized coordinates. See Appendix B.
- `estimate_fundamental_matrix()`: Calculates the fundamental matrix.

3 Part 3: Fundamental matrix with RANSAC

For two photographs of a scene it's unlikely that you'd have perfect point correspondence with which to do the regression for the fundamental matrix. So, next you are going to compute the fundamental matrix with point correspondences computed using SIFT. As discussed in class, least squares regression alone is not appropriate in this scenario due to the presence of multiple outliers. In order to estimate the fundamental matrix from this noisy data you'll need to use RANSAC in conjunction with your fundamental matrix estimation.

You'll use these putative point correspondences and RANSAC to find the "best" fundamental matrix. You will iteratively choose some number of point correspondences (8, 9, or some small number), solve for the fundamental matrix using the function you wrote for part II, and then count the number of inliers. Inliers in this context will be point correspondences that "agree" with the estimated fundamental matrix. In order to count how many inliers a fundamental matrix has, you'll need a distance metric based on the fundamental matrix. (Hint: For a point correspondence (x, x') what properties does the fundamental matrix have?). You'll need to pick a threshold between inliers and outliers and your results are very sensitive to this threshold, so explore a range of values. You don't want to be too permissive about what you consider an inlier, nor do you want to be too conservative. Return the fundamental matrix with the most inliers.

Recall from lecture the expected number of iterations of RANSAC to find the "right" solution in the presence of outliers. For example, if half of your input correspondences are wrong, then you have a $(0.5)^8 = 0.39\%$

chance to randomly pick 8 correspondences when estimating the fundamental matrix. Hopefully that correct fundamental matrix will have more inliers than one created from spurious matches, but to even find it you should probably do thousands of iterations of RANSAC.

For many real images, the fundamental matrix that is estimated will be "wrong" (as in it implies a relationship between the cameras that must be wrong, e.g., an epipole in the center of one image when the cameras actually have a large translation parallel to the image plane). The estimated fundamental matrix can be wrong because the points are co-planar or because the cameras are not actually pinhole cameras and have lens distortions. Still, these "incorrect" fundamental matrices tend to do a good job at removing incorrect SIFT matches (and, unfortunately, many correct ones).

For this part, you will implement the following methods in `part3_ransac.py`:

- `calculate_num_ransac_iterations()`: Calculates the number of RANSAC iterations needed for a given guarantee of success.
- `ransac_fundamental_matrix()`: Uses RANSAC to find the best fundamental matrix.

4 Part 4: Performance comparison

In this part, you will compare the performance of using the direct least squares method against RANSAC. You can use the code in the notebook for visualization of the fundamental matrix estimation. The first one uses a subset of the matches and the direct linear method (which corresponds to the `estimate_fundamental_matrix()` in part 2) to compute the fundamental matrix. The second method uses RANSAC, which corresponds to `ransac_fundamental_matrix()` in part 3. Based on your output visualizations, answer the reflection questions in the report.

5 Part 5: Visual odometry

Visual odometry (VO) is an important part of the simultaneous localization and mapping (SLAM) problem. In this part, you can observe the implementation of VO on a real-world example from Argoverse. VO will allow us to recreate most of the ego-motion of a camera mounted on a robot – the relative translation (but only up to an unknown scale), and the relative rotation. See [2] for a more detailed understanding. Based on the output and your understanding of the process, answer the reflection questions in the report.

6 Part 6: Panorama stitching

*This section is **required** for CS 6476 students and **optional** for CS 4476.*

Note: This section will not be autograded - the code will be submitted to gradescope in the code assignment for project 3 and the report must include the README and visualizations asked for to get credit. This part will be hand-graded, and points will be awarded in the report section. However, we will still require a submission of code in part6.

In this part of the project, you will perform an application of the homography matrix that RANSAC helps us compute image/panorama stitching. Given a pair/group of images from multiple view points, you can automatically stitch these images together to create a panorama-like effect of a wider view.

To perform panorama stitching on your own pair of images (feel free to reuse the ones you took for project 2), you will:

1. Choose/get interest points for the pair of images.
2. Find candidate matches among the interest points.

3. Use your implementation of RANSAC from part 3 to estimate the homography matrix (between these interest points).

4. Project each image onto the same surface and stitch (warp operation).

We do not expect perfect stitching. Rather, we are simply asking you to apply the homography matrix that RANSAC helps us calculate to blend two images using some basic linear projections. Remember that the homography will help you warp one image into another in some sense. From this, it follows that computing the homography will allow you to perform the warping operation that will be the basis behind your panorama stitching algorithm. In the report, please outline your method and place the appropriate visualizations.

You're more than welcome to reuse your interest point pipeline from project 2, but you may choose instead to use any interest point/feature matching functions from OpenCV, which is already installed in your environment. You are **not** allowed to use `cv2.warpPerspective()` (or any similar function) to perform the warping step though.

7 Writeup

For this project (and all other projects), you must do a project report using the template slides provided to you. Do ***not*** change the order of the slides or remove any slides, as this will affect the grading process on Gradescope and you will be deducted points. In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of your algorithm. The template slides provide guidance for what you should include in your report. A good writeup doesn't just show results–it tries to draw some conclusions from the experiments. You must convert the slide deck into a PDF for your submission, and then assign each PDF page to the relevant question number on Gradescope.

If you choose to do anything extra, add slides *after the slides given in the template deck* to describe your implementation, results, and analysis. You will not receive full credit for your extra credit implementations if they are not described adequately in your writeup.

Potentially useful NumPy (Python library) functions

- `np.linalg.svd()` - This function returns the singular value decomposition of a matrix. Useful for solving the linear systems of equations you build and reducing the rank of the fundamental matrix.
- `np.linalg.inv()` - This function returns the inverse of a matrix.
- `np.random.randint()` - Lets you pick integers from a range. Useful for RANSAC.

Forbidden functions

(You can use these for testing, but not in your final code). You may not use the SciPy constrained least squares function `scipy.optimize.lsq_linear()` or any similar function. You may also not use anyone else's code that estimates the fundamental matrix or performs RANSAC for you. If it feels like you're sidestepping the work, then it's probably not allowed. Ask the TAs if you have any doubts.

Testing

We have provided a set of tests for you to evaluate your implementation. We have included tests inside `proj3.ipynb` so you can check your progress as you implement each section. When you're done with the entire project, you can call additional tests by running `pytest proj3_unit_tests` inside the root directory of the project, as well as checking against the tests on Gradescope. *Your grade on the coding portion of the project will be further evaluated with a set of tests not provided to you.*

Bells & whistles (extra points)

We don't have good suggestions for extra credit on this project. If you have ideas, come talk to us! If you choose to do extra credit, you should add slides ***at the end*** of your report further explaining your implementation, results, and analysis. You will not be awarded credit if this is missing from your submission.

Rubric

- +20 pts: Implementation of projection matrix estimation in `part1_projection_matrix.py`
- +25 pts: Implementation of fundamental matrix estimation in `part2_fundamental_matrix.py`
- +25 pts: Implementation of RANSAC in `part3_ransac.py`
- +30 pts: Report
- -5*n pts: Lose 5 points for every time you do not follow the instructions for the hand-in format

Submission format

This is very important as you will lose 5 points for every time you do not follow the instructions. You will submit two items to Gradescope:

1. `<your_gt_username>.zip` containing:
   (a) `src/`: directory containing all your code for this assignment
   (b) `setup.cfg`: setup file for environment, no need to change this file
   (c) `additional_data/` - (required for 6476, optional for 4476) the images you used for Part 6, and/or if you use any data other than the images we provide, please include them here
   (d) `README.txt`: (optional) if you implement any new functions other than the ones we define in the skeleton code (e.g., any extra credit implementations), please describe what you did and how we can run the code. We will not award any extra credit if we can't run your code and verify the results.
2. `<your_gt_usernamme>_proj3.pdf` - your report

Do ***not*** install any additional packages inside the conda environment. The TAs will use the same environment as defined in the config files we provide you, so anything that's not in there by default will probably cause your code to break during grading. Do ***not*** use absolute paths in your code or your code will break. Use relative paths like the starter code already does. Failure to follow any of these instructions will lead to point deductions. Create the zip file using `python zip_submission.py --gt_username <your_gt_username>` (it will zip up the appropriate directories/files for you!) and hand it in with your report PDF through Gradescope (please remember to mark which parts of your report correspond to each part of the rubric).

Credits

Assignment developed by James Hays, Cusuh Ham, Arvind Krishnakumar, Jing Wu, John Lambert, Samarth Brahmbhatt, Grady Williams, and Henry Hu, based on a similar project by Aaron Bobick.

References

[1] Richard I Hartley. "In defense of the eight-point algorithm". In: *IEEE Transactions on pattern analysis and machine intelligence* 19.6 (1997), pp. 580–593.

[2] D. Scaramuzza and F. Fraundorfer. "Visual Odometry [Tutorial]". In: *IEEE Robotics Automation Magazine* 18.4 (2011), pp. 80–92. DOI: 10.1109/MRA.2011.943233.

Appendix A Fundamental matrix derivation

Recall that the definition of the fundamental matrix is:

$$\begin{bmatrix}\mathbf{x}_2\\1\end{bmatrix} K^{-T}\left({}^2E_1\right)K^{-1}\begin{bmatrix}\mathbf{x}_1\\1\end{bmatrix}=0 \tag{12}$$

Where does this equation come from? Szeliski shows that a 3D point $\mathbf{p}$ being viewed from two cameras (see Figure 3) can be modeled as:

$$d_1\hat{\mathbf{x}}_1=\mathbf{p}_1={}^1\mathbf{R}_0\mathbf{p}_0+{}^1\mathbf{t}_0={}^1\mathbf{R}_0(d_0\hat{\mathbf{x}}_0)+{}^1\mathbf{t}_0 \tag{13}$$

where $\hat{\mathbf{x}}_j=\mathbf{K}_j^{-1}\mathbf{x}_j$ are the (local) ray direction vectors. Note that ${}^1\mathbf{R}_0$ and ${}^1\mathbf{t}_0$ define an SE(3) '1_T_0' object that transforms $\mathbf{p}_0$ from camera 0's frame to camera 1's frame. We'll refer to these just as $\mathbf{R}$ and $\mathbf{t}$ for brevity in the following derivation.

We can eliminate the $+\mathbf{t}$ term by a cross-product. This can be achieved by multiplying with a skew-symmetric matrix as $[\mathbf{t}]_\times\mathbf{t}=0$. Then:

$$d_1[\mathbf{t}]_\times\hat{\mathbf{x}}_1=d_0[\mathbf{t}]_\times\mathbf{R}\hat{\mathbf{x}}_0. \tag{14}$$

Swapping sides and taking the dot product of both sides with $\hat{\mathbf{x}}_1$ yields

$$d_0\hat{\mathbf{x}}_1^T([\mathbf{t}]_\times\mathbf{R})\hat{\mathbf{x}}_0=d_1\hat{\mathbf{x}}_1^T[\mathbf{t}]_\times\hat{\mathbf{x}}_1=0, \tag{15}$$

Since the cross product $[\mathbf{t}]_x$ returns 0 when pre- and post-multiplied by the same vector, we arrive at the familiar epipolar constraint, where $\mathbf{E}=[\mathbf{t}]_\times\mathbf{R}$:

$$\hat{\mathbf{x}}_1^T\mathbf{E}\hat{\mathbf{x}}_0=0 \tag{16}$$

The fundamental matrix is defined as $F=\mathbf{K}_1^{-T}\mathbf{E}\mathbf{K}_0^{-1}$. Thus,

$$\begin{pmatrix}u' & v' & 1\end{pmatrix}\mathbf{K}_1^{-T}\mathbf{E}\mathbf{K}_0^{-1}\begin{pmatrix}u\\v\\1\end{pmatrix}=0 \tag{17}$$

We can write this as:

$$\begin{pmatrix}u' & v' & 1\end{pmatrix}\begin{pmatrix}f_{11} & f_{12} & f_{13}\\f_{21} & f_{22} & f_{23}\\f_{31} & f_{32} & f_{33}\end{pmatrix}\begin{pmatrix}u\\v\\1\end{pmatrix}=0 \tag{18}$$

Appendix B Unnormalizing normalized coordinates

The main idea of coordinate normalization is to replace coordinates $\mathbf{u}_a$ in image a with $\hat{\mathbf{u}}_a=T_a\mathbf{u}_a$, and coordinates $\mathbf{u}_b$ in image b with $\hat{\mathbf{u}}_b=T_b\mathbf{u}_b$. If T is chosen to be invertible, then we can recover the original coordinates from the transformed ones, as

$$\begin{aligned}\hat{\mathbf{u}}_a&=T_a\mathbf{u}_a\\T_a^{-1}\hat{\mathbf{u}}_a&=T_a^{-1}T_a\mathbf{u}_a\\T_a^{-1}\hat{\mathbf{u}}_a&=\mathbf{u}_a\end{aligned} \tag{19}$$

Substituting in the equation $\mathbf{u}_b^TF\mathbf{u}_a=0$, we derive the equation

$$\begin{aligned}\mathbf{u}_b^TF\mathbf{u}_a&=0\\(T_b^{-1}\hat{\mathbf{u}}_b)^TFT_a^{-1}\hat{\mathbf{u}}_a&=0\\\hat{\mathbf{u}}_b^TT_b^{-T}FT_a^{-1}\hat{\mathbf{u}}_a&=0\end{aligned} \tag{20}$$

If we use the normalized points $\mathbf{u}_a, \mathbf{u}_b^T$ when fitting the fundamental matrix, then we will end up estimating $\hat{F} = T_b^{-T} F T_a^{-1}$. In other words, $\mathbf{u}_b^T F \mathbf{u}_a = \hat{\mathbf{u}}_b^T \hat{F} \hat{\mathbf{u}}_a$. If we want to find out the original F that corresponded to raw (unnormalized) point coordinates, than we need to transform backwards:

$$\begin{aligned} \hat{F} &= T_b^{-T} F T_a^{-1} \\ T_b^T \hat{F} &= T_b^T T_b^{-T} F T_a^{-1} \\ T_b^T \hat{F} T_a &= F T_a^{-1} T_a \\ T_b^T \hat{F} T_a &= F \end{aligned} \tag{21}$$