

# CS231A: Computer Vision, From 3D Perception to 3D Reconstruction and beyond



Homework #2

(Spring 2025)

Due: Thursday, May 2

## Submitting

Prepare a PDF with your answers and submit it to the corresponding Gradescope assignment. We recommend using the provided LaTeX templates from our website, but you may generate the PDF in any preferred manner.

For code implementations, modify the provided `.py` files located in the `code` folder. Problems involving only code implementation are marked with the  symbol. Written answers are required only where instructions are **bolded** and usually ask you to include the final output of your code and/or a brief description of your implementation. (These questions are also marked with the ).

Additionally, submit your completed Python files to the separate Gradescope coding assignment, which includes an autograder for verification. We grade primarily from the PDF and use the autograder submission as needed for additional checks.

To test your code, either install the required packages locally or use Google Colab by following these steps:

- Open Google Drive and click the settings (gear) icon at the top-right corner.
- Select **Settings** → **Manage Apps**.
- Click **Connect more apps**, search for **Colab**, and add it.
- Upload `PSET2.ipynb`, open it in Colab, and follow the instructions provided.

Before submission, transfer your code from the notebook into the `.py` files. Compress these Python files into a `.zip` file and submit it to the Gradescope code assignment.

## Honor Code

- **Collaboration:** You may collaborate with other students on the homework. However, each student should independently write up their own solutions and **clearly list the names of their collaborators** in their write-up.
- **Committing code to GitHub:** Please do not push your homework code to public GitHub repositories (for example, a fork of our repository). If you wish to commit your solutions to GitHub, please create a **private repository** by making a new copy of our repository, rather than forking it.

Onto the problems!

# 1 Fundamental Matrix Estimation From Point Correspondences (30 points)

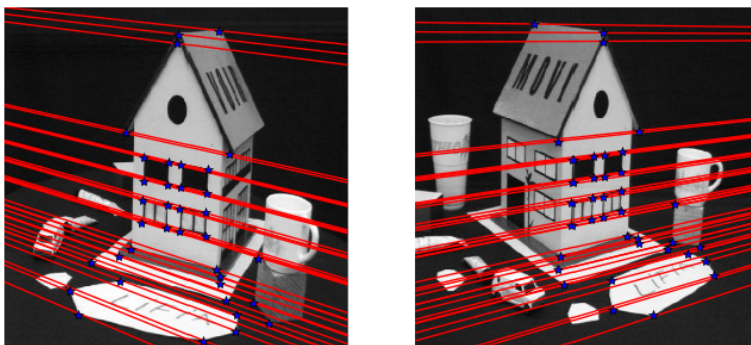










Figure 1: Example illustration, with epipolar lines shown in both images (Images courtesy Forsyth & Ponce)

This problem is concerned with the estimation of the fundamental matrix from point correspondences. In this problem, you will implement both the linear least-squares version of the eight-point algorithm and its normalized version to estimate the fundamental matrices. You will implement the methods in `p1.py` and complete the following:

- (a)  Implement the linear least-squares eight point algorithm in `lls_eight_point_alg()`. Remember to enforce the rank-two constraint for the fundamental matrix via singular value decomposition. [15 points]
- (b)  Include your resulting fundamental matrix with 4 decimal places minimum and briefly describe your implementation in your written report. [5 points]
- (c)  Implement the normalized eight point algorithm in `normalized_eight_point_alg()`. Hint: Remember to enforce the rank-two constraint for the fundamental matrix via singular value decomposition. [2.5 points]
- (d)  Report the returned fundamental matrix with 4 decimal places minimum. [2.5 points]
- (e)  After implementing methods to determine the Fundamental matrix, we can now determine epipolar lines. Specifically to determine the accuracy of our Fundamental matrix, we will compute the average distance between a point and its corresponding epipolar line in `compute_distance_to_epipolar_lines()`. [2.5 points]
- (f)  Briefly describe your implementation of `compute_distance_to_epipolar_lines()` in your written report. [2.5 points]

## 2 Matching Homographies for Image Rectification (20 points)

Building off of the previous problem, this problem seeks to rectify a pair of images given a few matching points. The main task in image rectification is generating two homographies  $H_1, H_2$  that transform the images in a way that the epipolar lines are parallel to the horizontal axis of the image. You will implement the methods in `p2.py` as follows:

- (a)  The first step in rectifying an image is to determine the epipoles. Complete the function `compute_epipole()`. Hint: Recall that  $F^T e = 0$ , and how you can use SVD to solve for  $e$ . [2 points]
- (b)  Let's solve for the homography  $H$  that maps an epipole  $e$  to a point on the horizontal axis at infinity  $(f, 0, 0)$ . This may at first look complicated, but it is just a sequence of several relatively simple steps. Complete the function `find_H()`. [5 points]

We have copied the relevant portions from the course notes below to make this straightforward:

As noted in the course notes, a good practice is to find a homography that acts like a transformation that translates and rotates points near the center of the image.

1. First, define the translation matrix that moves the center to  $(0, 0, 1)$  in homogeneous coordinates:

$$T = \begin{bmatrix} 1 & 0 & -\frac{\text{width}}{2} \\ 0 & 1 & -\frac{\text{height}}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

2. Then, create a rotation matrix to place the epipole on the horizontal axis at some point  $(f, 0, 1)$ . If the translated epipole  $Te$  is located at homogeneous coordinates  $(e_1, e_2, 1)$ , then the rotation applied is:

$$R = \begin{bmatrix} \alpha \frac{e_1}{\sqrt{e_1^2 + e_2^2}} & \alpha \frac{e_2}{\sqrt{e_1^2 + e_2^2}} & 0 \\ -\alpha \frac{e_2}{\sqrt{e_1^2 + e_2^2}} & \alpha \frac{e_1}{\sqrt{e_1^2 + e_2^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where  $\alpha = 1$  if  $e_1 \geq 0$  and  $\alpha = -1$  otherwise.


3. Now we just need to get  $(f, 0, 1)$  to  $(f, 0, 0)$ , which we can do with the following matrix:

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{f} & 0 & 1 \end{bmatrix}$$

4. Finally, you can compute the output to `find_H()` as follows:

$$H_2 = T^{-1}GRT$$

We don't require you to explain it, but it should be relatively easy to understand why the above matrices do what we want them to.

- (c)  Now for the trickier bit - finding a matching pair of homographies  $H_1$  and  $H_2$  by implementing `compute_matching_homographies()`. [10 points]

Once again, we include the relevant portions of the course notes below:

Begin by using `compute_H()` to solve for  $H_2$ , so we now just need to find an  $H_1$  to match this  $H_2$ . To do this, we find an  $H_1$  that minimizes the sum of square distances between the corresponding points of the images

$$\arg \min_{H_1} \sum_i \|H_1 p_i - H_2 p'_i\|^2$$

Although the derivation<sup>1</sup> is outside the scope of this class, we know that  $H_1$  is of the form:

$$H_1 = H_A H_2 M$$

where  $F = [e]_{\times} M$  and

$$H_A = \begin{bmatrix} a_1 & a_2 & a_3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

with  $(a_1, a_2, a_3)$  composing the elements of a certain vector  $\mathbf{a}$  that will be computed later.

First, as explained in the course notes to solve for  $M$  we need to implement the following:


$$M = [e]_{\times} F + e v^T$$

Where  $v^T = [1 \ 1 \ 1]$ . Recall that  $[e]_{\times}$  is the cross-product matrix of  $e$ .

To finally solve for  $H_1$ , we need to compute the  $\mathbf{a}$  values of  $H_A$ . Again as explained in the course notes, we can do this by solving a least-squares problem  $W\mathbf{a} = b$  for  $\mathbf{a}$  where

$$W = \begin{bmatrix} \hat{x}_1 & \hat{y}_1 & 1 \\ & \vdots & \\ \hat{x}_n & \hat{y}_n & 1 \end{bmatrix} \quad b = \begin{bmatrix} \hat{x}'_1 \\ \vdots \\ \hat{x}'_n \end{bmatrix} \quad (1)$$

After computing  $\mathbf{a}$ , we can compute  $H_A$  and finally  $H_1$ . Thus, we generated the homographies  $H_1, H_2$  to rectify any image pair given a few correspondences.







- (d)  **Include the pair of rectified image in your written report. Briefly comment on why rectification makes it easier to find corresponding points in the two images.** [3 points]

---

<sup>1</sup>If you are interested in the details, please see Chapter 11 of Hartley & Zisserman's textbook *Multiple View Geometry*

### 3 The Factorization Method (20 points)

In this question, you will explore the factorization method, initially presented by Tomasi and Kanade, for solving the affine structure from motion problem. You will implement the methods in `p3.py` and complete the following:


- (a)  Implement the factorization method as described in lecture and in the course notes. Complete the function `factorization_method()`. [8 points]
- (b)  Briefly describe your implementation in your written report. [2 points]
- (c)  Run the provided code that plots the resulting 3D points. Compare your result to the ground truth provided. The results should look identical, except for a scaling and rotation. Explain why this occurs. [3 points]
- (d)  Report the 4 singular values from the SVD decomposition. Why are there 4 non-zero singular values? How many non-zero singular values would you expect to get in the idealized version of the method, and why? [2 points]
- (e)  The next part of the code will now only load a subset of the original correspondences. Compare your new results to the ground truth, and explain why they no longer appear as similar (if you rotate the reconstruction, you can see the points are not quite right). [3 points]
- (f)  Report the new singular values, and compare them to the singular values that you found previously. Explain any major changes. [2 points]

## 4 Triangulation in Structure From Motion (30 points)



Figure 2: The set of images used in this structure from motion reconstruction.

Structure from motion is inspired by our ability to learn about the 3D structure in the surrounding environment by moving through it. Given a sequence of images, we are able to simultaneously estimate both the 3D structure and the path the camera took. In this problem, you will implement significant parts of a structure from motion framework, estimating both  $R$  and  $T$  of the cameras, as well as generating the locations of points in 3D space. Recall that in the previous problem we triangulated points assuming affine transformations. However, in the actual structure from motion problem, we assume projective transformations. By doing this problem, you will learn how to solve this type of triangulation. In Course Notes 4, we go into further detail about this process. You will implement the methods in `p4.py` and complete the following:

- (a)  Given correspondences between pairs of images, we compute the respective Fundamental and Essential matrices. Given the Essential matrix, we must now compute the  $R$  and  $T$  between the two cameras. However, recall that there are four possible  $R, T$  pairings. In this part, we seek to find these four possible pairings, which we will later be able to decide between. In the course notes, we explain in detail the following process:


1. To compute  $R$ : Given the singular value decomposition  $E = UDV^T$ , we can rewrite  $E = MQ$  where  $M = UZU^T$  and  $Q = UWV^T$  or  $UW^TV^T$ , where

$$Z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that this factorization of  $E$  only guarantees that  $Q$  is orthogonal. To find a rotation, we simply compute  $R = (\det Q)Q$ .

2. To compute  $T$ : Given that  $E = U\Sigma V^T$ ,  $T$  is simply either  $u_3$  or  $-u_3$ , where  $u_3$  is the third column vector of  $U$ .

Implement this in the function `estimate_initial_RT()`. We provide the correct  $R, T$ , which should be contained in your computed four pairs of  $R, T$ . **[5 points]**

- (b)  In order to distinguish the correct  $R, T$  pair, we must first know how to find the 3D point given matching correspondences in different images. The course notes explain in detail how to compute a linear estimate (DLT) of this 3D point:

1. For each image  $i$ , we have  $p_i = M_i P$ , where  $P$  is the 3D point,  $p_i$  is the homogenous image coordinate of that point, and  $M_i$  is the projective camera matrix.

2. Formulate matrix


$$A = \begin{bmatrix} p_{1,1}m_1^{3\top} - m_1^{1\top} \\ p_{1,2}m_1^{3\top} - m_1^{2\top} \\ \vdots \\ p_{n,1}m_n^{3\top} - m_n^{1\top} \\ p_{n,2}m_n^{3\top} - m_n^{2\top} \end{bmatrix}$$

where  $p_{i,1}$  and  $p_{i,2}$  are the xy coordinates in image  $i$  and  $m_i^{k\top}$  is the  $k$ -th row of  $M_i$ .

3. The 3D point can be solved for by using the singular value decomposition.

Implement the linear estimate of this 3D point in `linear_estimate_3d_point()`.

**[5 points]**

- (c)  However, we can do better than linear estimates, but usually this falls under some iterative nonlinear optimization. To do this kind of optimization, we need some residual. A simple one is the reprojection error of the correspondences, which is computed as follows:  
For each image  $i$ , given camera matrix  $M_i$ , the 3D point  $P$ , we compute  $y = M_i P$ , and find the image coordinates

$$p'_i = \frac{1}{y_3} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$


Given the ground truth image coordinates  $p_i$ , the reprojection error  $e_i$  for image  $i$  is

$$e_i = p'_i - p_i$$

The Jacobian is written as follows:

$$J = \begin{bmatrix} \frac{\partial e_1}{\partial P_1} & \frac{\partial e_1}{\partial P_2} & \frac{\partial e_1}{\partial P_3} \\ \vdots & \vdots & \vdots \\ \frac{\partial e_m}{\partial P_1} & \frac{\partial e_m}{\partial P_2} & \frac{\partial e_m}{\partial P_3} \end{bmatrix}$$


Recall that each  $e_i$  is a vector of length two, so the whole Jacobian is a  $2K \times 3$  matrix, where  $K$  is the number of cameras. Fill in the methods `reprojection_error()` and `jacobian()`, which computes the reprojection error and Jacobian for a 3D point and its list of images. Like before, we print out a way to verify that your code is working. **[5 points]**

- (d)  Implement the Gauss-Newton algorithm, which finds an approximation to the 3D point that minimizes this reprojection error. Recall that this algorithm needs a good initialization, which we have from our linear estimate in part (b). Also recall that the Gauss-Newton algorithm is not guaranteed to converge, so, in this implementation, you should update the estimate of the point  $\hat{P}$  for 10 iterations (for this problem, you do not have to worry about convergence criteria for early termination):

$$\hat{P} = \hat{P} - (J^T J)^{-1} J^T e$$

where  $J$  and  $e$  are the Jacobian and error computed from the previous part.


Implement the Gauss-Newton algorithm to find an improved estimate of the 3D point in the `nonlinear_estimate_3d_point()` function. Like before, we print out a way to verify that your code is working. **[5 points]**

- (e)  Now finally, go back and distinguish the correct  $R, T$  pair from part (a) by implementing the method `estimate_RT_from_E()`. You will do so by:

1. First, compute the location of the 3D point of each pair of correspondences given each  $R, T$  pair
2. Given each  $R, T$  you will have to find the 3D point's location in that  $R, T$  frame. The correct  $R, T$  pair is the one for which the most 3D points have positive depth (z-coordinate) with respect to both camera frames. When testing depth for the second camera, we must transform our computed point (which is the frame of the first camera) to the frame of the second camera.

**[5 points]**

Congratulations! You have implemented a significant portion of a structure from motion pipeline. Your code is able to compute the rotation and translations between different cameras, which provides the motion of the camera. Additionally, you have implemented a robust method to triangulate 3D points, which enable us to reconstruct the structure of the scene. In order to run the full structure from motion pipeline, please change the variable `run_pipeline` at the top of the main function to `True`. Hopefully, you can see a point cloud that looks like the frontal part of the statue in the above sequence of images.

- (f)  **Submit the final plot of the reconstructed statue. Now what if we wanted to reconstruct a larger scene, such as a house? Comment on why that may be harder than reconstructing the statue. [5 points]**

Note: Since the class is using Python, the structure from motion framework we use is not the most efficient implementation. It will be common that generating the final plot may take a few minutes to complete. Furthermore, Matplotlib was not built to be efficient for 3D rendering. Although it's nice to wiggle the point cloud to see the 3D structure, you may find that the GUI is laggy. If we used better options that incorporate OpenGL (see Glumpy), the visualization would be more responsive. However, for the sake of the class, we will only use the numpy-related libraries.