CS 231A: Computer Vision (Winter 2017)
Problem Set 2

Due Date: May 3th 2017 11:59pm

1 Fundamental Matrix Estimation From Point Correspondences (30 points)

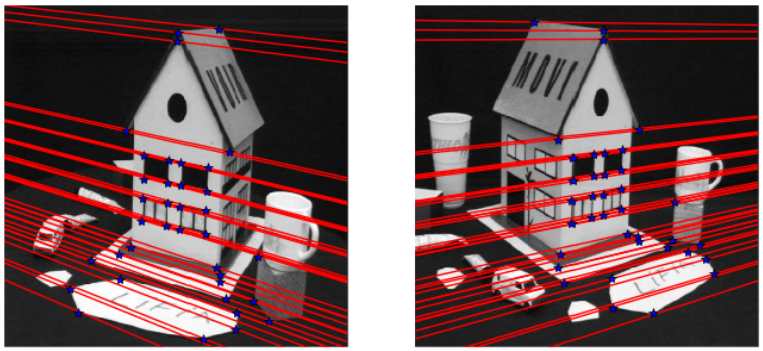

Figure 1: Example illustration, with epipolar lines shown in both images (Images courtesy Forsyth & Ponce)

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 `fundamental_matrix_estimation.py` and complete the following:

(a) Implement the linear least-squares eight point algorithm in `lls_eight_point_alg()` and report the returned fundamental matrix. Remember to enforce the rank-two constraint for the fundamental matrix via singular value decomposition. Attach a copy of your code. **[10 points]**

(b) Implement the normalized eight point algorithm in `normalized_eight_point_alg()` and report the returned fundamental matrix. Remember to enforce the rank-two constraint for the fundamental matrix via singular value decomposition. Attach a copy of your code. **[5 points]**

(c) In this problem, we provide two datasets, which consist of a pair of images with matching points. For each of the two eight point algorithms, draw a figure similar to Figure 1, which shows the selected corresponding points and their epipolar lines, for each dataset. To do

Four figures: w/o or w/ normalization for data set1&2

so, fill out the method `plot_epipolar_lines_on_images()`. Attach a copy of your code.
[10 points]

Tip: Matplotlib is a bit finicky about plotting images next to each other using `subplot()`. If you find yourself seeing that images are of different sizes, then simply make the figure size wider, so that it has more room for the smaller (probably wider) image.

(d) 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()`. Attach a copy of your code.
[5 points]

2 Matching Homographies for Image Rectification (20 points)

Building off 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. A detailed description of the standard homography generation for image rectification can be found in Course Notes 3. You will implement the methods in `image_rectification.py` and complete the following:

(a) The first step in rectifying an image is to determine the epipoles. Complete the function `compute_epipole()`. Submit the epipoles you calculated and a copy of your code.
[5 points]

Tip: If you want to check your answer, recall that the epipole is where all the epipolar lines intersect.

(b) Finally, we can determine the homographies H_1 and H_2. We first compute H_2 as the homography that takes the second epipole e_2 to a point at infinity $(f, 0, 0)$. The matching homography H_1 is computed by solving a least-squares problem. Complete the function `compute_matching_homographies()`. Submit the homographies computed for the sample image pair a copy of your code.
[10 points]

(c) Submit a copy of your rectified images plotted using `plot_epipolar_lines_on_images()`.
[5 points]

3 The Factorization Method (15 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 `factorization_method.py` and complete the following:

(a) Implement the factorization method as described in lecture and in the course notes. Complete the function `factorization_method()`. Submit a copy of your code.
[5 points]

(b) 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]

(c) 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]

(d) 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 similar.
[3 points]

(e) 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 (35 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 `triangulation.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()`. For now, we provide the correct R, T, which should be contained in your computed four pairs of R, T. Submit the four pairs of R, T and a copy of your code.
[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 $p_i = M_iP$, 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

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

   where $p_{i,1}$ and $p_{i,2}$ are the xy coordinates in image i and $m^{k\top}$ is the k-th row of M.
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()`. Like before, we provide a unit test that you can use to verify that your code is working. Submit the output generated by this unit test and a copy of your code.
[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 $y = M_iP$, 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 provide a unit test that you can use to verify that your code is working. Submit the output generated by this unit test and a copy of your code.
[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 provide a unit test that you can use to verify that your code is working. Submit the output generated by this unit test and a copy of your code.
[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.

Submit the output generated by this unit test and a copy of your code.
[5 points]

(f) 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`. Submit the final plot of

the reconstructed statue. Hopefully, you can see a point cloud that looks like the frontal part of the statue in the above sequence of images.

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 though.
[10 points]