Problem Set 6

CMSC 426

Assigned Tuesday, April 11, Due Tuesday, April 25

In this problem set you will implement algorithms to determine the relative motion between two cameras, based on images taken by them of the same set of points. You will use the eight point algorithm to determine the Essential matrix that relates the two images, use the Essential matrix to find the epipolar lines, use the epipolar lines to find the epipole, and use the epipole to find the camera translation.

In order to do this, we will need to model cameras that produce images with camera coordinates, not world coordinates, as we have been doing. This means that every camera is described by a focal point, a focal length, and three vectors describing the x, y, and z coordinates of points relative to that camera. The z vector indicates the direction that the camera is facing. Until now, we haven't really thought beyond that about each camera as having its own coordinate system. We have to think about this now, because we need to figure out the x and y coordinates of points in an image when we take a picture. And, when we calculate the relative position of two cameras, we need to think about the rotation that will make one camera's x coordinate match the x coordinate of a second camera.

Remember that to find the coordinates of a point relative to a new coordinate system, we need to take inner products with unit vectors that represent the x, y and z coordinates. Let's consider some examples.

Suppose we have a camera with a focal point at (4,2,6), with an x direction of (1,0,0), a y direction of (0,1,0), and a z direction of (0,0,1) (these are just the standard directions). We have a point in the world with coordinates of (10,8,9), and we want to determine its coordinates in the camera's coordinate system. First we take: v = (10,8,9) – (4,2,6) = (6,6,3). Then, to find the x coordinate we take (6,6,3).(1,0,0) and get an x coordinate of 6 (the . means dot product). Similarly, we get y and z coordinates of 6 and 3. This gives us the coordinates of the point, (6,6,3) in the camera coordinate system, in which we use the focal point as the origin.

Next, suppose we have a camera with a focal point at (2,1,2), with an x direction of $(0, 1/\sqrt{2}, 1/\sqrt{2})$, a y direction of (1,0,0) and a z direction of $(0, 1/\sqrt{2}, -1/\sqrt{2})$. Then if we have a point in the world at (10,9,8), the x coordinate of this point in the camera's coordinate system will be $((10,9,8)-(2,1,2)).\ (0, 1/\sqrt{2}, 1/\sqrt{2}) = (8,8,6).(0, 1/\sqrt{2}, 1/\sqrt{2}), = 14/\sqrt{2}$. We compute the y and z coordinates of the point in the same way.

We are including a main function, called ps6, and some functions that display some of your results visually. Note that there are also some lines in ps6 that do not end in semi-colons. Do not modify these functions. They will ensure that your results are printed and displayed in a standard format, to make them easier to grade. You just need to fill in the bodies of functions that are empty. We also include a function called `test_points`. This function generates points to test your code. We will show the results of some of these tests below, so that you can tell whether your code is working properly. Ps6 will also print some results that are not shown below. We can use these results to further evaluate your code. So you might want to do additional tests to ensure that your code is correct. In your write-up, include copies of everything that your code prints or displays in a figure, along with your code.

1. Standard projection. First, write a function that takes a picture of 3D points assuming that the camera has a focal point at (0,0,0) and is looking in the z direction. The function should have the form:

```
function qs = project_standard(Ps, f)
```

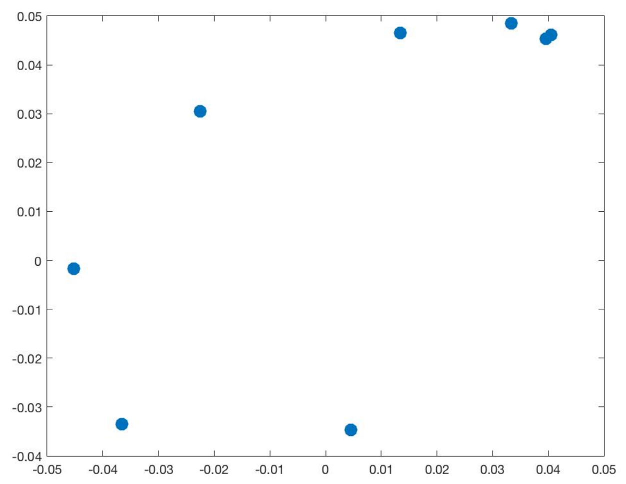
Ps is a 3xn matrix containing the coordinates of n 3D points. Each column of Ps contains the (X,Y,Z) coordinates of a point. f contains the focal length of the camera. We assume that the camera is facing in the direction (0,0,1) and that the x and y coordinates of the camera point in the directions (1,0,0) and (0,1,0) respectively. The output, qs, is a 2xn matrix. Each column contains the (x,y) coordinates of a point in the image. The first column of qs contains the image of the first column of Ps.

When you run ps6(1), the first things that are printed out should be:

```
ps =

   0.0333    0.0396   -0.0366    0.0405    0.0135   -0.0452   -0.0225    0.0045
   0.0485    0.0454   -0.0336    0.0461    0.0465   -0.0016    0.0305   -0.0347
```

Your program should also display the following figure.



2. Changing coordinate systems. Now write a function of the form:

```
function Qs = change_coordinates(origin, xvec, yvec, zvec, Ps)
```

This takes a set of 3D points, Ps, and finds their coordinates in a new coordinate system with the origin, and x, y, and z vectors specified by `origin, xvec, yvec, zvec`. The output, Qs, is also a set of 3D points. Running ps6(2) should produce:

```
ans =

  6.4015
  9.0000
  8.0635
```

3. General projection. Now you should write a function of the form:

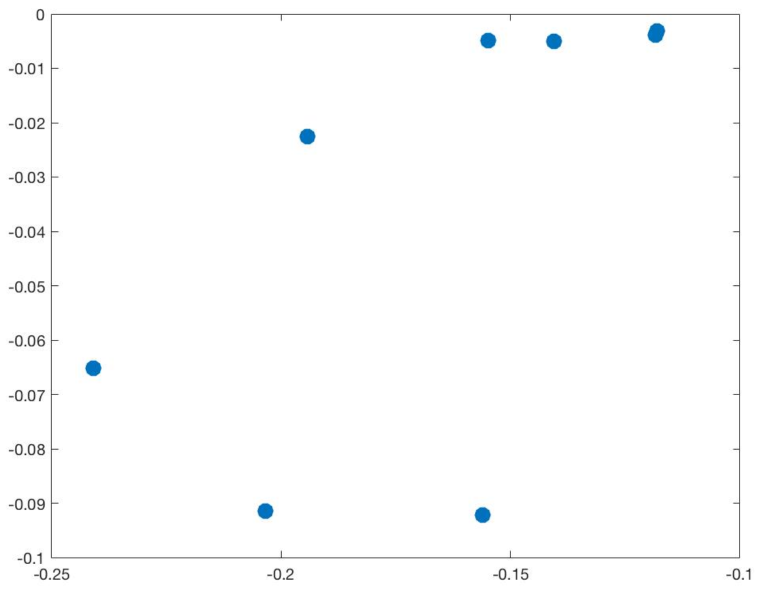
```
function qs = project_general(fp, xvec, yvec, zvec, f, Ps)
```

This function should project 3D points into an image for a camera in general position. f, Ps and qs mean the same thing as in problem (1), but now we also specify the focal point of the camera (fp), and the directions of the coordinates (xvec, yvec, zvec). The image points are defined relative to these coordinates. Note that you can implement this using the functions you created for the previous two problems. When you run ps6(3), your program should print out:

```
qs =

 -0.1405  -0.1184  -0.2035  -0.1180  -0.1548  -0.2409  -0.1943  -0.1560
 -0.0050  -0.0038  -0.0915  -0.0032  -0.0048  -0.0652  -0.0225  -0.0921
```

You should also display the figure:



This may look similar to the last figure, but if you look at the values on the x and y axis you will see that it is quite different.

4. For this problem, you must implement a function of the form:

```
function E = eight_point(ps, qs)
```

This function takes two sets of points, ps and qs, which are the 2D image points from two sets of images. It outputs E, the essential matrix, which captures the epipolar geometry between the two images. If E is correct, it should be the case that ps'*E*qs = 0. When you run ps6(4), the first thing that is printed should look like:

```
E =

  -0.0000    0.3780   -0.1890
  -0.3780   -0.0000    0.5669
   0.1890   -0.5669    0.0000
```

Because there is no rotation between the two cameras in this case, you should also be able to see that this essential matrix matches the translation between the two cameras, up to a scale factor.

5. Now you will use the essential matrix to compute the epipolar lines. Write a function of the form:

```
function lines = epipolar_lines(qs, E)
```

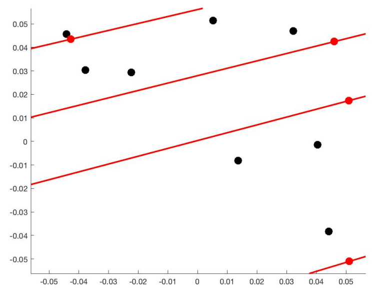
qs contains n points in the second image and E contains the essential matrix relating the first and second image. lines will be a 3xn matrix that represents a set of lines, one for each point in qs. The i'th column of lines describes the epipolar line for the i'th point in qs. If we think of a column of lines as having the form [a;b;c] then it represents the line ax + by + c = 0.

To get an epipolar line from E, recall that if p is a point in the first image, and q is the corresponding point in the second image, p'*E*q = 0. The corresponding column of lines, call it *line*, should be chosen so that p'*line = 0. When you run ps6(5), the program should output:

```
lines =

  0.2053   0.2340   0.1971   0.1917
 -0.6164  -0.6153  -0.6660  -0.6091
  0.0002  -0.0434   0.0374   0.0170
```

It should also display the figure:



The black points are the 8 points in the first image used to compute E. The red points are four additional points in the image. Four epipolar lines for these points are also displayed. These should each go through one of the red points.

6. The epipolar lines will either be parallel, or will all intersect at the same point in the image. This point is called the *epipole*. In this problem, we choose the cameras so that the epipolar lines all intersect. Write a function of the form:

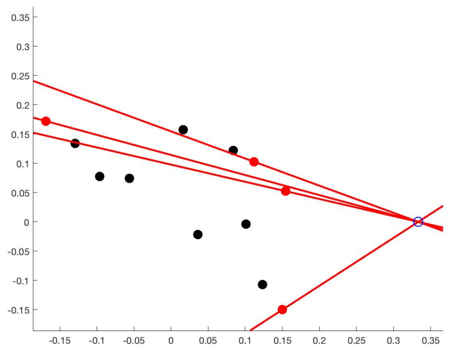
```
function epipole = find_epipole(lines)
```

to find the epipole. To do this, you just need to find the intersection point of the lines. The output, epipole, should just be the coordinates of a 2D point. When you run this, the first thing the program should print is:

```
epipole =

   0.3333
  -0.0000
```

It should also display the figure:



Note that there is a little blue circle at the epipole.

7. Now you can use the epipole to find the direction of translation between the two cameras. To see why this is, recall that we get conjugate epipolar lines by taking any *epipolar plane* that goes through the cameras' two focal points, and intersecting that plane with the two image planes. So every epipolar plane contains the line connecting the two focal points. The points where this line intersects the two image planes are called the *epipoles*. Consider, for example, the epipole in image 1. Since this epipole lies on every epipolar plane, it is always on the intersection between the epipolar plane and the first image plane. So it is on every epipolar line in the first image. That means that all epipolar lines intersect at the epipole. (It is also possible that the line connecting the two focal points is parallel to the first image plane, and doesn't intersect it. In this case, all the epipolar lines will be parallel).

   Ok, so now we know that the epipole lies on the line connecting the two focal points. Suppose we know the epipole in the first image (we computed it in the last problem). The vector from the epipole to the first focal point tells us the direction from the first focal point to the second, that is, the direction of translation between the two cameras. Compute this direction. Note that to do this, you have to think about the 3D coordinates of the epipole, in the coordinate system of the first camera. You also have to think about the coordinates of the focal point of the camera in this coordinate system, but we can assume that that is the origin.

   So, implement the function:

```
function trans = find_translation(epipole, fp, xvec, yvec, zvec, f)
```

This takes as input the location of the epipole in the first image, and the parameters that describe the first camera. It outputs a 3D vector. This should be the same as the translation between the two cameras, up to an unknown scale factor. (Recall that we can never know the magnitude of the translation. If we scale the whole world, including the points and camera locations, we scale the translation, but produce exactly the same images). Running ps6(7) should produce:

```
E =

  -0.0000    0.6708    0.0000
  -0.7053   -0.0000   -0.0501
  -0.0000   -0.2236   -0.0000


ans =

   0.3333
  -0.0000
   1.0000

fp2 =

     1
     0
     3
```

Here the translation is given as (1/3; 0; 1). Note that in this problem, the first focal point is at (0;0;0), and the second is at (1;0;3). So this translation is correct.

8. In the previous problems we have computed the essential matrix given the image locations of corresponding points. This is how we might do it in real life. Now we are going to compute the essential matrix assuming that we know the positions of the cameras. This will allow us to check that we have things right (and hopefully, help you to understand the essential matrix better). We begin by computing the rotation that relates the two cameras, with the function:

```
function R = compute_rotation(xvec1, yvec1, zvec1, xvec2, yvec2, zvec2)
```

The rotation, R, should take a point in the second coordinate system and return a point in the first coordinate system. See the comments in ps6, case 8 for more description of this. Equivalently, R should rotate xvec2 to xvec1, yvec2 to yvec1, and zvec2 to zvec1. Running ps6(8) should produce:

```
q1 =

   5.7018
   3.0000
   4.5265


ans =

   5.7018
   3.0000
   4.5265
```

9. We now use this rotation to create the essential matrix relating two cameras. Looking at the slides from class, we can see that the essential matrix can be written as E = S*R. R is the rotation from the second camera's coordinate system to the first camera's coordinate system. S is a matrix formed by the vector that represents translation from the first focal point to the second one. If we encode this translation as T = (T_x,T_y,T_z). Then we have:

$$S = \begin{pmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{pmatrix}$$

You should implement the function:

```
function E = compute_essential(fp1, xvec1, yvec1, zvec1, …
      fp2, xvec2, yvec2, zvec2)
```

If you run ps6(9) you should see:

```
E1 =

   0.0000   0.5000  -0.5000
   0.0000  -0.0000   0.0000
   0.7071   0.0000  -0.0000


E2 =

        0  -7.0711   7.0711
        0        0        0
 -10.0000        0        0
```

This compares our two methods of computing the Essential matrix. Notice that the two Essential matrices differ by a scale factor. This is ok, since the essential matrix is defined by the equation qs'*E*ps = 0. Scaling E doesn't change this equality.

10. Finally, this problem doesn't require any new implementation. It just tests your code from problem 9 on a more difficult case. It's easy to write your code in a way that works on 9 but not on 10, so if that is the case, fix it. Your results should be:

```
ans =

    0.0674    0.0674    0.0674
    0.0674    0.0674    0.0674
    0.0674    0.0674    0.0674
```

This is the ratio of the essential matrix computed in the two different ways you have implemented. The fact that it is always the same value shows that the two matrices are identical up to a scale factor.

11. **Challenge Problem:** In problem 7 we used the Essential matrix to determine the direction of translation of the cameras. Using this, write a function to determine the rotation of the cameras. You can have a look at https://en.wikipedia.org/wiki/Essential_matrix#Determining_R_and_t_from_E or Szeliski, Section 7.2.

12. **Challenge Problem:** As described here, the eight-point algorithm can be sensitive to noise. To see this, try adding a small amount of independent Gaussian noise to each coordinate of each image point. Report how much this affects the Essential matrix that you compute. Now look at the normalized eight point algorithm (see https://en.wikipedia.org/wiki/Eight-point_algorithm#The_normalized_eight-point_algorithm or http://webserver2.tecgraf.puc-rio.br/~manuel/TeseDoutorado/ReferencePapers/InDefense8PointAlgortihm.pdf or Szeliski, Section 7.2). See if you can make implement this to improve performance of the algorithm when there is noise. Write up an explanation of what you did and how it affected the results.