ECE4580 Homework #4

Due: Feb. 9, 2017

Problem 1. (20 pts) You've got a surveillance camera setup and you'd like to calibrate it in order to reason about what the camera is seeing. So you go out into the field of view of the camera and place a bunch of markers to image. Taking a picture of the following markers,

$$p_1^W = \begin{Bmatrix} 13.18 \\ -3.362 \\ 12.48 \end{Bmatrix}, p_2^W = \begin{Bmatrix} 24.75 \\ -11.790 \\ 15.57 \end{Bmatrix}, p_3^W = \begin{Bmatrix} 14.47 \\ -9.918 \\ 5.456 \end{Bmatrix}, p_4^W = \begin{Bmatrix} 15.83 \\ -11.390 \\ 15.78 \end{Bmatrix}, p_5^W = \begin{Bmatrix} 22.51 \\ -1.419 \\ 15.30 \end{Bmatrix}, p_6^W = \begin{Bmatrix} 12.55 \\ -5.846 \\ 12.21 \end{Bmatrix}.$$

leads to the following image coordinates,

$$r_1 = \begin{Bmatrix} 157.0000 \\ 376.0000 \end{Bmatrix}, r_2 = \begin{Bmatrix} 92.0000 \\ 139.0000 \end{Bmatrix}, r_3 = \begin{Bmatrix} 416.0000 \\ 97.0000 \end{Bmatrix}, r_4 = \begin{Bmatrix} 3.0000 \\ 26.0000 \end{Bmatrix}, r_5 = \begin{Bmatrix} 91.0000 \\ 387.0000 \end{Bmatrix}, r_6 = \begin{Bmatrix} 167.0000 \\ 238.0000 \end{Bmatrix}.$$

Using the Direct Linear Transform derive the projection matrix M associated to the camera.

Note: To avoid having to enter the world points and the image coordinates, there is a Matlab file `calib01.mat` *in the Assignment section. The image points are given in ray form. Also, to make your life easier, you may want to consider making the code a function called* `calibrateM.m` *(a code stub is uploaded too).*

Problem 2. (20 pts) Your friend started a camera calibration process but did not finish it. He went so far as to compute the huge matrix to perform the SVD on, but sort of stopped there. The matrix generated is:

$$\begin{bmatrix}
10.4300 & 3.5150 & 7.1330 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & -7.6400 & -2.5750 & -5.2250 & -0.7325 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 10.4300 & 3.5150 & 7.1330 & 1.0000 & 2.6860 & 0.9051 & 1.8370 & 0.2575 \\
10.1300 & 3.3780 & 8.6890 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & -5.5720 & -1.8580 & -4.7790 & -0.5500 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 10.1300 & 3.3780 & 8.6890 & 1.0000 & 5.3940 & 1.7990 & 4.6270 & 0.5325 \\
10.0100 & 6.8080 & 7.2640 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & -0.8258 & -0.5617 & -0.5993 & -0.0825 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 10.0100 & 6.8080 & 7.2640 & 1.0000 & -0.5255 & -0.3574 & -0.3814 & -0.0525 \\
9.6690 & 8.8240 & 10.8200 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 5.7050 & 5.2060 & 6.3840 & 0.5900 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 9.6690 & 8.8240 & 10.8200 & 1.0000 & 3.2150 & 2.9340 & 3.5980 & 0.3325 \\
11.4000 & 8.4200 & 8.9750 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 2.9070 & 2.1470 & 2.2890 & 0.2550 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 11.4000 & 8.4200 & 8.9750 & 1.0000 & -0.1995 & -0.1473 & -0.1571 & -0.0175 \\
13.2400 & 6.5490 & 5.9180 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & -3.5750 & -1.7680 & -1.5980 & -0.2700 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 13.2400 & 6.5490 & 5.9180 & 1.0000 & -4.4350 & -2.1940 & -1.9830 & -0.3350
\end{bmatrix}$$

but your friend did one funny thing. Since your friend had already had calibrated the intrinsic parameters, rather than use image coordinates for the generation of the matrix, they were mapped to real world units via the inverse of the camera matrix Ψ. Thus, the only real unknown is the rotation and translation component (basically the extrinsic parameters).

Finish the calibration job half-done by your friend by computing the singular value decomposition. The matrix is huge, so you should be able to find a zip file in the modules section with this matrix in the file `svd_prob.mat` which is a Matlab file with the matrix variable `Q`. The zero singular value right vector should be converted into a 3×4 matrix that encodes for the rotation and translation of world coordinates to camera coordinates (after correction of the scale). Recall that the matrix is built up across the rows, then down the columns. When you finish building it, you will have the matrix

$$\left[R^T \;\middle|\; -R^T T\right].$$

Compute the transpose of the left-most 3×3 sub-matrix to get R, and solve also for T using the last column (relative to world frame). What is the camera position and orientation?

Given that you know the intrinsic parameters of the camera to be

$$\Psi = \begin{bmatrix} 400.0000 & 0.0000 & 320.0000 \\ 0.0000 & -400.0000 & 240.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

(the negative is due to using Matlab matrix image coordinates), the world coordinate $p^W = (14.3423, 4.3856, 3.0725)^T$ most likely projects to which coordinate (up to quantization error)

(a) $\begin{Bmatrix} 492.0000 \\ 276.0000 \end{Bmatrix}$ (b) $\begin{Bmatrix} 376.0000 \\ 249.0000 \end{Bmatrix}$ (c) $\begin{Bmatrix} 633.0000 \\ 9.0000 \end{Bmatrix}$ (d) $\begin{Bmatrix} 380.0000 \\ 216.0000 \end{Bmatrix}$.

Problem 3. (15 pts) Use imfilter to perform a Gaussian blur on an image. An nrows x ncols Gaussian template can be obtained in the following method:

h = fspecial('gaussian', [nrows ncols], sd);

where `sd` is the standard deviation of the Gaussian (type 'help fspecial' for more info), and where `nrows` and `ncols` are be the number of rows and columns, respectively, of the filter template. Call the function smooth.m. Amongst the homework files there is a Matlab file called blurme that contains a simple image of a square. blur it with different standard deviations and explain what happens. In the homework document, include both your code and the square image smoothed with a standard deviation of 2.

Apply the code to another image of choice (can be from the homework, from the web, or wherever). Turn in a before and after, along with the standard deviation selected.

Note: the options `nrows` *and* `ncols` *indicate the size of the convolution kernel window to use. It is not the size of the image to filter. Using the image size will give a huge convolution kernel; way bigger than is needed. In general the window should be as big, or bigger, than twice the standard deviation.*

Problem 4. (25 pts) Let's continue the stereo problem from before, where your workshop had a stereo rig setup with the following left and right cameras:

$$g_L^W = \left[\begin{array}{ccc|c} 0.914 & -0.064 & 0.402 & -8.659 \\ 0.288 & 0.799 & -0.527 & 2.170 \\ -0.288 & 0.597 & 0.749 & 4.830 \\ \hline \multicolumn{3}{c|}{0} & 1 \end{array}\right], \quad g_R^W = \left[\begin{array}{ccc|c} 0.995 & -0.016 & 0.103 & 10.659 \\ 0.074 & 0.808 & -0.584 & 5.830 \\ -0.074 & 0.588 & 0.805 & 1.170 \\ \hline \multicolumn{3}{c|}{0} & 1 \end{array}\right].$$

Furthermore, the left and right cameras both have the same camera matrix,

$$\Psi_R = \Psi_L = \begin{bmatrix} 400.000 & 0.000 & 320.000 \\ 0.000 & 400.000 & 240.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}.$$

(a) Suppose that you imaged two points q_1 and q_2 in the left and right cameras,

$$r_1^L = \begin{Bmatrix} 302.0000 \\ 236.0000 \end{Bmatrix}, \; r_2^L = \begin{Bmatrix} 492.0000 \\ 193.0000 \end{Bmatrix}, \quad \text{and} \quad r_1^R = \begin{Bmatrix} 279.0000 \\ 235.0000 \end{Bmatrix}, \; r_2^R = \begin{Bmatrix} 372.0000 \\ 194.0000 \end{Bmatrix}.$$

Where are these two points q_1^W and q_2^W (in world coordinates)?

(b) What depth (or distance along the optical axis) do they have relative to each camera?

(c) Having to always know where the camera is in world coordinates is kind of lame, since the camera can't really move. If the camera is expected to move, then what is typically done is that the camera frame from one of the camera's is taken to be the world frame. In that case, its transformation of coordinates from world frame to the chosen camera frame do nothing (basically, the transformation matrix is the identity matrix). In contrast, then second camera's transformation describes the changes of coordinates from the chosen camera frame to its own camera frame (if that makes sense). Doing this trick, then allows one to compute the coordinates of an imaged point relative to the chosen camera frame. Thus as the stereo rig moves around imaging points, one can always recover the coordinates of the points in the scene relative to the chosen camera frame at the moment in time that the images were taken.

Suppose that you move the stereo rig, you want the left camera to be the main camera frame, and you imaged the following two points in the left and right cameras,

$$\vec{r}_1^L = \begin{Bmatrix} 206.0000 \\ 204.0000 \end{Bmatrix}, \; \vec{r}_2^L = \begin{Bmatrix} 299.0000 \\ 249.0000 \end{Bmatrix} \quad \text{and} \quad \vec{r}_1^R = \begin{Bmatrix} 293.0000 \\ 203.0000 \end{Bmatrix}, \; \vec{r}_2^R = \begin{Bmatrix} 382.0000 \\ 250.0000 \end{Bmatrix}. \tag{1}$$

Where are these two points q_1^L and q_2^L (relative to the left camera frame)?

Note: In this case, the only transformation needed is g_L^R, as per the pragraph above.

You are free to use any of the methods in the notes. The easiest thing would be to write a Matlab function that takes in the two camera transformations, the matched points in each image, and then returns the necessary info. A (mostly empty) code stub called `stereodepth.m` is given that can be used as a template.

Note: To avoid having to enter the matrices and vectors into Matlab, there is a Matlab file `stereo02.mat` *in the Assignment section.*

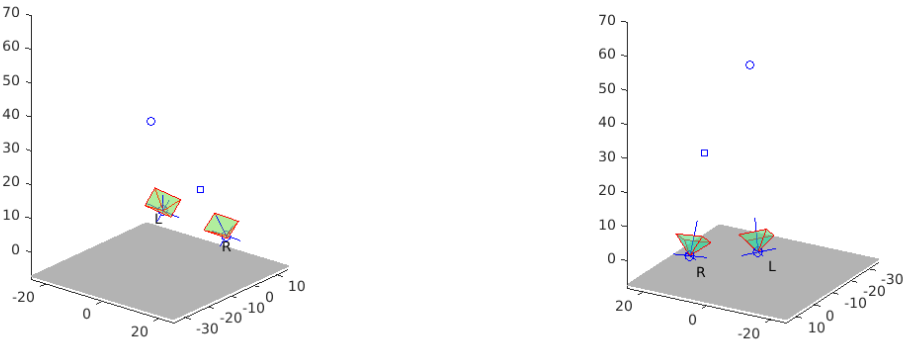

Figure 1: Two views of the stereo rig setup and the imaged points in the real world. The pyramid things at the left and right coordinate frames visualize the image plane using a viewer-centered coordinate system.

Problem 5. (20 pts) Select a learning module to work through, under the presumption that you will commit to it for the duration of the semester. Perform the first week's activity. You are fine to work in groups of two, as well as to consult with other groups working on the same topic as you if need be. The group submission should reflect the work of the group, and should also be submitted individually with the name of your partner in the document.

The below is discussion of expected deliverables:

The submission can be published in with the main document, or submitted separately. Whatever makes most sense for the particular topic. For these submissions though, the expectation is that what gets turned in will be a little more extensive than the normal homework in terms of documentation of outcomes. You should roughly reiterate the activity, discuss how it was accomplished, demonstrate through images (if possible) the functionality, then briefly note any observations (including cases where it may not work so faithfully, or what is needed to work very well).

Make sure to add a brief video if there is video processing; only one of you needs to upload the video or have a link to some online video (past submissions have used youtube). For those using Matlab, you can use their video writer to save the image sequences being output to a figure. For others, there are usually ways to record the desktop.

Singular Value Decomposition and Camera Matrix Estimation. Six known coordinate pairs of q and r lead to a system of equations that satisfies,

$$Q\vec{m} = 0,$$

where Q is a 12×12 matrix. Now, this problem does not look like what you are used to because of the 0 on the right hand side. Certainly, one answer is $\vec{m} = 0$, but it can't be the right answer since the zero matrix transformation all points to zero. What is happening is that the matrix Q has a non-trivial null space. There exists another vector $\vec{m}^*$, besides the zero vector, such that multiplication of Q by this $\vec{m}^*$ gives the zero vector. Another way of putting it, is that Q is not full rank. It is in fact rank deficient by 1 (Q has a rank of 11 instead of 12).

We would like to find this special vector $\vec{m}^*$ to figure out what the rotation and translation are for a given camera placement. Recall from your linear algebra class that a matrix can be broken up into eigenvectors and eigenvalues by solving for:

$$Ax = \lambda x.$$

Combining the eigenvectors into a matrix V and the eigenvalues into a diagonal matrix Σ, the following equation holds

$$A = V\Sigma V^T.$$

Well, turns out that Q has one eigenvalue that should be 0. If we can find this eigenvalue, then its eigenvector gives us the special vector $\vec{m}^*$. Unfortunately, eigenvalues are sensitive to noise and become complex quite easily, so solving for eigenvalues and eigenvectors is not the preferred way to do it.

There is another decomposition that is very similar to the eigenvalue/eigenvector decomposition. It is called the singular value decomposition and is guaranteed to return positive, real eigenvalues. The singular value decomposition returns matrices U, S, and V such that

$$A = USV^T. \tag{2}$$

The matrix S is diagonal and is equal to the (complex) magnitude squared of the eigenvalues in decreasing magnitude. So, the bottom right corner of the matrix S should be zero if we compute the SVD of Q. Furthermore, the last column of the right vector V is the solution vector. Matlab has a function called `svd` that performs this decomposition. Once you find the solution, then the last column can be used to get the vector $\vec{m}$. Of course, the solution is only known up to scale.

In general, the SVD will often give a solution that is known only up to scale. Additional information must be used to establish the proper scale value, which is then multiplied (or divided) to get the correctly scaled matrix solution.

The Direct Linear Transform and Projection Matrix Estimation. Recall from the reading, that some people prefer to write the projection equations

$$\vec{r} \sim \Psi \left[R_W^C \;\middle|\; T_W^C \right] q^W$$

as

$$\vec{r} \sim M q^W$$

and basically lump everything into one big matrix of intrinsic and extrinsic parameters. The Direct Linear Transform (DLT) utilizes the cross-product trick,

$$\vec{r} \times \left(M q^W \right) = 0,$$

as a means to solve for the unknown matrix M. Since this equation has only two independent rows, it provides two independent equations. Since the projection matrix has twelve values in it, then six world to image coorespondences are needed. Having six (r, q^W) pairings leads to twelve equations and twelve unknowns. Some people use more points in order to improve on the error due to quantization effects (in effect performing a least-square fit via the SVD). Since the reading did not go over the full derivation, I will provide it here plus a couple tips on how to implement it in Matlab.

Moving on, remember that the cross product, being a linear operation, can be written as a matrix product instead

$$\hat{r} \left(M q^W \right) = 0, \quad \text{where} \quad \hat{r} = \begin{bmatrix} 0 & -r^3 & r^2 \\ r^3 & 0 & -r^1 \\ -r^2 & r^1 & 0 \end{bmatrix}.$$

Recall that we are treating the image coordinate r to be the ray $\vec{r}$ by adding a third coordinate value of 1. Since the above equation is linear in the elements of M, the elements of M can be pulled out as though the were the vector $\vec{m}$. The DLT reading doesn't cover it, so we'll go over it here a little bit. Manipulating the equation to expose the rows of the matrix and also open up $\hat{r}$,

$$\begin{bmatrix} 0 & -r^3 & r^2 \\ r^3 & 0 & -r^1 \\ -r^2 & r^1 & 0 \end{bmatrix} \begin{Bmatrix} M^1 q^W \\ M^2 q^W \\ M^3 q^W \end{Bmatrix} = 0,$$

where M^i is the i^{th} row of M. Since the right element is a vector, each row is a scalar. The cool thing about scalars is that the transpose of a scalar is the same value, so the above equation is equivalent to

$$\begin{bmatrix} 0 & -r^3 & r^2 \\ r^3 & 0 & -r^1 \\ -r^2 & r^1 & 0 \end{bmatrix} \begin{Bmatrix} (q^W)^T (M^1)^T \\ (q^W)^T (M^2)^T \\ (q^W)^T (M^3)^T \end{Bmatrix} = 0.$$

Ha-ha! Now, we can factor out the pieces of M,

$$\begin{bmatrix} 0 & -r^3 & r^2 \\ r^3 & 0 & -r^1 \\ -r^2 & r^1 & 0 \end{bmatrix} \begin{bmatrix} (q^W)^T & 0 & 0 \\ 0 & (q^W)^T & 0 \\ 0 & 0 & (q^W)^T \end{bmatrix} \begin{Bmatrix} (M^1)^T \\ (M^2)^T \\ (M^3)^T \end{Bmatrix} = 0,$$

where the middle element is a 3x12 matrix (the 0 terms are really row vectors consisting of three 0 values). Let's defined the vector $\vec{\vec{m}}$ to consist of the first row elements, the second row elements, and the third row elements horizontally concatenated then transposed (to give a 12x1 vector). That means we have,

$$\begin{bmatrix} 0 & -r^3 & r^2 \\ r^3 & 0 & -r^1 \\ -r^2 & r^1 & 0 \end{bmatrix} \begin{bmatrix} (q^W)^T & 0 & 0 \\ 0 & (q^W)^T & 0 \\ 0 & 0 & (q^W)^T \end{bmatrix} \vec{\vec{m}} = 0,$$

which is totally solvable using the SVD approach once we build up a 12x12 matrix (currently we have a 3x12 with a dependent row). With a little manipulation, it is possible to show that a simplified form of the equation is

$$\begin{bmatrix} 0 & -r^3 (q^W)^T & r^2 (q^W)^T \\ r^3 (q^W)^T & 0 & -r^1 (q^W)^T \\ -r^2 (q^W)^T & r^1 (q^W)^T & 0 \end{bmatrix} \vec{\vec{m}} = 0,$$

and we've recovered the equation from the DLT reading. Note that the 0 entries are row vectors with four zero values. The matrix is a 3x12 matrix as it should be. Define $\text{Take2}(\vec{r}, q^W)$ to be the function that takes the top two rows from the above matrix given the image ray $\vec{r}$ and the world coordinate q^W. Given the six $(\vec{r}, q^W)$ pairings, form the master matrix equation:

$$A\vec{m} = \begin{bmatrix} \text{Take2}(\vec{r}_1, q_1^W) \\ \text{Take2}(\vec{r}_2, q_2^W) \\ \text{Take2}(\vec{r}_3, q_3^W) \\ \text{Take2}(\vec{r}_4, q_4^W) \\ \text{Take2}(\vec{r}_5, q_5^W) \\ \text{Take2}(\vec{r}_6, q_6^W) \end{bmatrix} \vec{m} = 0$$

Performing SVD on the master matrix A will give you the proper answer. Be careful to properly reconstruct the matrix M (don't confuse the rows and columns).

"Wait!" you say, "What about the scale?" Well, it turns out that the scale doesn't matter because the answer will give you a ray. If you have a new (seventh) world point q_7^W, then using the computed projection matrix M, gives

$$\vec{r}_7 \sim M q_7^W,$$

but with a ray whose last entry is not unit valued. To get the image coordinates requires dividing by the last row in order to make it unit valued,

$$\vec{\vec{r}} = \begin{Bmatrix} \vec{r}_7^1 \\ \vec{r}_7^2 \end{Bmatrix} / \vec{r}_7^3.$$

This final division step actually has the effect of cancelling out the unknown scale. Magic (or, that's the beauty of rays!).

Matlab implementation note: One trick to creating the Take2 function is to use an inline function in Matlab. For this DLT problem, it would be

```
Take2 = @(r, q) [ 0 0 0 0  , -r(3)*(q') , r(2)*(q') ; ...
                  r(3)*(q') ,  0 0 0 0   , -r(1)*(q') ];
```

Then, it is literally possible to just do:

```
A = [ Take2( r1, q1 ) ; Take2( r2, q2 ) ; Take2( r3, q3 ) ;  ...
      Take2( r4, q4 ) ; Take2( r5, q5 ) ; Take2( r6, q6 ) ];
```

and Matlab will build the 12x12 matrix for you. Remember that r should be in ray-homogeneous coordinates (a 3x1) and q must be in homogeneous coordinates (a 4x1).

Solving for Extrinsic Parameters Given Intrinsic Parameters. If we know the intrinsic parameters of a camera, then the method derived in class can be used to solve for the extrinsic variables only (a modified version of the DLT can also be used, but won't be covered here). Let's work this out a little bit. Recall that

$$\begin{Bmatrix} r^1 \\ r^2 \\ \hline 1 \end{Bmatrix} \sim \Psi \left[R^T \;\middle|\; -R^T T \right]_W^C q^W.$$

where R and T are really with respect to the world frame, i.e., R_C^W and T_{WC}^W (yeah, it's kinda funny but that's what it is with respect to the coordinates we have). Well, if we invert by Ψ, which is assumed to be known, then

$$\Psi^{-1} \begin{Bmatrix} r^1 \\ r^2 \\ \hline 1 \end{Bmatrix} \sim \left[R^T \;\middle|\; -R^T T \right]_W^C q^W,$$

Define the transformed $\vec{r}$ to be given by the vector $\vec{z}$, then we have

$$\vec{z} \sim \left[R^T \;\middle|\; -R^T T \right] q^W = D q^W, \quad \text{where } \vec{z} = \Psi^{-1} \begin{Bmatrix} r^1 \\ r^2 \\ \hline 1 \end{Bmatrix}. \tag{3}$$

Following the derivation from class, the projection similarity equations above lead to the follow set of equations for each world point plus image coordinate pairing,

$$\begin{bmatrix} q^1 & q^2 & q^3 & 1 & 0 & 0 & 0 & 0 & -z^1 q^1 & -z^1 q^2 & -z^1 q^3 & -z^1 \\ 0 & 0 & 0 & 0 & q^1 & q^2 & q^3 & 1 & -z^2 q^1 & -z^2 q^2 & -z^2 q^3 & -z^2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{4}$$

With six such q and r pairs, you can solve for the unknown 3×4 matrix D. That's how your friend created the large matrix.

Now, there is the usual problem with using the SVD to generate the matrix. The length of the vector $\vec{m}$ is not possible to solve for without additional information. However, we do have one solution at our disposal. The left-most 3×3 matrix of the resulting solution matrix is a rotation matrix (transposed). Thus it must have a determinant of 1. If the value is not 1, then that is the scale factor we must apply to the matrix. In math talk, we know that

$$D = \left[R^T \;\middle|\; -R^T T \right], \tag{5}$$

where the vector $\vec{m}^*$ can be used to build D. Further, we know that $\det(R^T) = 1$. If the determinant is not equal to 1, then we must scale the solution to make equal to 1. Dividing by the cube root of $\det(D(1:3, 1:3))$ (in Matlab speak) will rescale the matrix to be what it should be. The n^{th}-root in Matlab is logically called `nthroot`. This then gives the real D matrix, or the best you can compute given what you have.

In general, the SVD will often give a solution that is known only up to scale. Additional information must be used to establish the proper scale value, which is then multiplied (or divided) to get the correctly scaled matrix solution.