Fall 2016 - CS 477/577 - Introduction to Computer Vision

Assignment Three

Due: 11:59pm (*) Thursday, Sept 29.

(*) There is grace until 8am the next morning, as the instructor will not grade assignment before then. However, once the instructor starts grading assignments, no more assignments will be accepted.

Weight: 9 points

This assignment must be done individually

General instructions

Unless there is a good reason others, you should use Matlab for this assignment. **If you want to use any other language, you must discuss this with the instructor at lease one week before the due date**. If permission is granted, you will have to sort out any image handling and numerical library support on your own or as a group. (The IVILAB has support for C/C++ , with example files geared towards this course that can be made available on request).

You need to create a PDF document that tells the story of the assignment, copying into it output, code snippets, and images that are displayed when the program runs. Even if the question does not remind you to put the resulting image into the PDF, if it is flagged with ($), you should do so. I should not need to run the program to verify that you attempted the question. See

http://kobus.ca/teaching/assignment-instructions.pdf

for more details about doing a good write-up. While it takes work, it is well worth getting better and more efficient at this. A substantive part of each assignment grade is reserved for exposition.

Assignment specification

This assignment has four parts, three of which are required for undergrads. (For keeners, there are some additional extra optional parts.)

To simplify things, you should hard code the file names in the version of your program that you hand in.

Part A

Overview. In part A you will explore fitting lines to data points using two different methods, and consider the difference between them. Recall that when there are more than two points, we do not expect a perfect fit even if the points come from a line because of noise. Fitting the line to many points mitigates the effect of noise (similar to averaging), providing that the noise is well behaved (no outliers). However, we need to be clear about what line we are looking for (i.e., what is the definition of the "best" line). There are a number of possibilities, and we explored two of them in class.

The file

http://kobus.ca/teaching/cs477/data/line_data.txt

is an ASCII file containing coordinates of points that are assumed to lie on a line. Apply the two kinds of least squares line fitting to the data. You need to implement the fitting yourself based on formulas given in class.

There may be Matlab routines that simplify this question (e.g. REGRESS), but you need to use slightly lower level routines to show that you understand how things are computed. However, you may find it interesting to compare with the output of REGRESS or like routines.

Your program should fit lines using both methods, and create two plots ($) showing the points and the fitted lines. For each method you should output the slope, the intercept, and the RMS error of the fit **under both models** (four numbers per fit, eight numbers total) **($)**. Comment in your writeup how you expect that the error under the one model to compare with the error of the line found by the alternative model, and vice versa, referring to your results as an example **($)**.

Part B

Overview. In this part you will calibrate a camera from an image of a coordinate system. You will use that image to extract the image location of points with known 3D coordinates by clicking on them. You will then use the camera calibration method developed in class to estimate a matrix M that maps locations represented by in 3D world coordinates to image locations. Having done so, you will be able to take any 3D point expressed in that coordinate system and compute where it would end up in the image.

Note that the following parts of this assignment will not work out especially accurately. You are purposely being asked to work with real, imprecise data that was collected quickly many years ago with an old camera.

Use the **first** of the following set of images to calibrate the camera that took it using at least 15 calibrations points. The other images are supplied for later parts.

http://kobus.ca/teaching/cs477/data/IMG_0862.jpeg
http://kobus.ca/teaching/cs477/data/IMG_0861.jpeg
http://kobus.ca/teaching/cs477/data/IMG_0863.jpeg
http://kobus.ca/teaching/cs477/data/IMG_0864.jpeg
http://kobus.ca/teaching/cs477/data/IMG_0865.jpeg

To set up a world coordinate system, note that the grid lines are 1 inch apart. Also, to be consistent with the light direction given later, the X-axis should be the axis going from the center leftwards, the Y-axis is going from the center rightwards, and the Z-axis is going up. (It is a right handed coordinate system). To get the pixel values of the points, you need to either write a Matlab script to get the coordinates of clicked points (pretty easy) or work out some other system. To help keep track of the points, I suggest using an image editing program to roughly label the points before you begin. For example, you could insert circles with labels P1, P2, etc. Put this image, or some other visual indication of exactly what your data is, in your report with a description **($)**. You should also hand in two files, world_coords.txt and image_coords.txt which the grader can use to run your program to get your results **($)**.

Note. *Be sure that your 3D points represent all three spatial dimensions (i.e., in "general position". Intuitively, if they were all in one plane (or mostly in one plane), we would not expect our calibration to work (well) for points not in that plane .*

Using the points, determine the camera matrix (denoted by M in class) using homogeneous least squares. Report the matrix **($)**.

Warning. *It is possible to map the 3D points to the 2D points using a linear transformation which you can find using non-homogeneous least squares. But it is important to realize that the mapping from 3D to 2D is fundamentally non-linear (the divide-by-w part), and thus a linear*

mapping is not as good. If you have the correct answer, your matrix (represented as a column vector should have norm 1, and further, its norm is arbitrary (you can double it and still get the same image coordinates). On the other hand, if you accidentally use the non-homogeneous method, then the matrix will not have unit norm, and it will matter (doubling it would lead to different image coordinates). You may find it interesting to do both ways and compare, but make sure that you at least hand in the homogeneous least squares solution that we developed in class.

Using M, project the points into the image. Your visualization should make it clear where the model estimates where the points **should go** in the image, as compared with where they did end up (i.e., the locations you clicked). This visualization should provide a check on your answer. Provide an image showing your visualization **($)**.

In addition, compute the error between where your model thinks each of the 3D points **should end up** in the image, and where they **actually did go** (i.e., where you found them by clicking). Report the error in terms of RMS, which (as mentioned in assignment two) is the square root (the root, "R") of the average (the mean, "M") of the total (the sum, "S"). This is a measure of how well your model predicts point locations. Don't forget to report the error **($)**.

Finally, note that the RMS error is monotonic with the sum of the squared error. Is the sum of the squared error the same error that the calibration process minimized? Why? Provide an answer and comment on whether or not this behavior is good or bad in your writeup **($)**.

Part C

Computer vision meets graphics

Introduction: One of the consumers of vision technology is graphics. Applications include acquiring models for objects based on images, and blending virtual worlds with image data. This requires understanding the image. By contrast, if you create a graphics image, the camera location and parameters are supplied by some combination of hard coded constants and user input. In the following, we have a different situation—we want to use the camera that **took the image**. Fortunately, we now know how to do this (consult M).

Now the task: Reportedly, the light was (roughly) at coordinates 33, 29, and 44. Ask yourself if this makes sense given the shading of the objects in the images that have objects. We now want to render a sphere into one of the images with one or more objects in it with plausible shading. Using the **Lambertian reflectance model,** render a sphere in the **second image** with radius 1/2 inches and located at (3,2,3) using any color you like. In order that this assignment does not rely on having taken graphics, we will accept any dumb algorithm for rendering a sphere. For example, you could model a sphere as:

```
x = x0 + cos(phi)*cos(theta)*R
y = y0 + cos(phi)*sin(theta)*R
z = z0 + sin(phi)*R
```

Now step phi from -pi/2 to pi/2 and step theta from 0 to 2*pi to get a bunch of 3D points that you will draw into the image using the camera model, G(X) developed in class. If your sphere has holes, use a smaller step size. Note that if you are working in Matlab, then, depending on how dumb your algorithm is, and how you implemented it, it may be slow.

There is one tricky point. We need to refrain from drawing the points that are not visible (because they are on the backside of the sphere). Determining whether a point is visible requires that we know where the camera is. The grad students will compute the location of the camera (see Part D), but since this is not required of undergraduates, a serviceable estimate is: 9, 14, 11.

Assume that the camera is at a point **P**. For each point on the sphere, **X**=(x,y,z), the outward normal direction for the point on the sphere can be determined (you should figure out what it is). Call this **N**(**X**). To decide if a point is visible, consider the tangent plane to the sphere at the point, and compute whether the camera is on the side of the plane that is outside the sphere. Specifically, if

(**P**-**X**).**N**(**X**) > 0

then the vector from the point to the camera is less than 90 degrees to the surface normal, and the point is visible. Provide an image showing off your sphere **($)**, and some description regarding what you did to get it **($)**.

> *Hint. Negative values of the Lambertian shading dot product suggest that the point is in self-shadow, so they should become zero. Also, it is instructive to check your code by making a light vector direction that is more behind the sphere and asking yourself if the resulting image makes sense.*

If you are an undergraduate student, and you have done all the questions until here, then congratulations, you are done! However, if you are looking for alternatives, or extra problems, feel free to keep reading.

Part D (required for grad students only).

For students whose research might involve 3D computer vision, I recommend doing both D and E (for modest extra credit, or, if you are quite familiar with graphics, substituting E for B)

Determining the extrinsic/intrinsic parameters

General advice for all assignments questions in this class (and other ones also), but might be particularly germane here: Consider how you can check your answers for every step. For example, try making up some numbers for your mapping in #1 to see that it does what you expect. For #4, you can check that your decomposition multiplies back to yield the matrix you are decomposing. Going slow and steady is faster in the end.

More general advice. It is often a good strategy to create synthetic data versions of what you are doing. In assignment two, this was roughly structured into the assignment. For this part of this assignment you should realize the advantage of building a synthetically generated M from an intrinsic matrix, a projection matrix, and an extrinsic matrix. To build the extrinsic matrix, you can easily get a random orthogonal matrix, V, from a random matrix U, by `[V,D]=eig(U'*U)`*. However, you should realize that the result is not necessarily a right hand system. If the determinant is -1 (you can use* `det()` *to find out) then you can reduce confusion for yourself by flipping the sign of one of the columns to make it a right hand coordinate system. Regardless, your synthetic M makes it easy to check your decomposition because you can compare what you get to the matrices that are known (since you created them).*

Recall that in class we learned that M is not an arbitrary matrix, but the product of 3 matrices, one that is known, and two others that have 11 parameters between them. Since there are 11 values available from M, this suggests that we can solve for those parameters. Let's give this a go!

1. Determine the intrinsic parameter matrix, *K*. Let's assume that the camera has perpendicular axes, so that we can assume that the skew angle is 90 degrees, which means that we can essentially ignore the concept (and we have only 10 unknown parameters). *K* first scales the canonical image coordinates by α (for horizontal) and β (vertical). This is followed by a mapping **which must be consistent** with how you recorded pixel locations in part B. If you followed the one common convention, then, referring to Figure 1, the mapping transforms the coordinates in the right-hand-side image to the left hand side image. There is obviously a shift involved, and to help the grader, please denote it by

(u_0, v_o) . Show your derivation and report K in algebra form, introducing (and clearly defining) any notation you need.

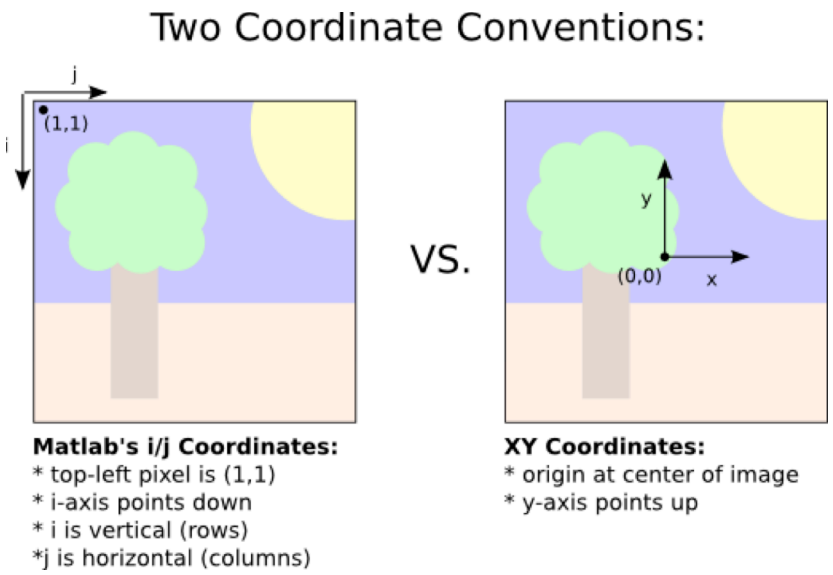

Figure 1. Standard camera coordinate system (right) and a typical way to refer to pixels in analogy with matrix element indexing. Assuming the left hand image is how you refer to the pixels you clicked in Part B, the second transformation in K will map coordinates on the right to coordinates on the left. (Figure credit: Kyle Simek a previous TA).

2. Now provide the extrinsic parameter matrix *X* in terms of three orthonormal rotation vectors and a translation vector. Do this by writing out the matrix as a product of two matrices, and then providing the product in algebraic form. Show your derivation and report the matrix in algebra form, introducing (and clearly defining) any notation you need **($)**.

3. Now algebraically multiply out *K*, the projection matrix, and *X*, to provide the camera matrix *M* discussed repeated in class, and estimated in Part B. Again, provide your best judgment of derivation details, and commentary to help the grader follow what you have done **($)**.

4. **Now the fun part**. The algebraic form of *M* just derived should be equal to the estimate M_{obs} found in Part B. Use this to determine all the unknown parameters. Begin by listing the unknown parameters as way of linking them to any notation you are using **($)**. Then solve for the unknowns in algebraic format (i.e., you can introduce notation for the observables (see first hint below)), showing your work (**$**).

Notation hint. A common way to set up this problem is to write:

$$\rho M_{obs} = \rho \begin{pmatrix} A & \boldsymbol{b} \end{pmatrix} = M$$

Here, ρ accounts for the arbitrary scale factor, and should be solved for relatively early on. (Remember that we had to choose a norm for M_{obs} to solve for it , but the real norm for M is $|\rho|$ which we have to determine). We have written the observed M_{obs} as a 3x3 matrix *A* and a 3x1 matrix (vector), $\boldsymbol{b}$. Once we have found ρ we can reason about the rows of *A*. A typical notation for them is $(\boldsymbol{a}_1^T, \boldsymbol{a}_2^T, \boldsymbol{a}_3^T)$.

The additional hints below can be consulted to the extent that they support your learning—perhaps best to try making progress without them for a modest period of time. Once you have solved the equations algebraically, plug in the observed numbers to provide estimates of the unknowns. For example, you will recover the orientation and location of the camera and alpha and beta. Report all your estimates **($)**. As a further direct check on the results, in the following image

http://kobus.ca/teaching/cs477/data/IMG_0859.jpeg

the camera was 11.5 inches from the wall. This can be used to compute alpha and beta more directly. Do that computation and compare with the results from the intrinsic parameter matrix **($)**..

Hint on handling the algebra. *It is probably quite challenging to solve this "head on" (but I am interested in alternative methods that people find). However if you focus on the rows of (to begin), you can assert that various norms, dot products, and cross products (see below) are equal. For example try setting constructs like* $\left|\rho \boldsymbol{a}_3^T\right|$, $\rho \boldsymbol{a}_1^T \bullet \rho \boldsymbol{a}_3^T$, *and* $\rho \boldsymbol{a}_1^T \times \rho \boldsymbol{a}_3^T$ *to their counterparts in your algebraic M and see what you can solve for. (Hopefully you got M in a tidy form with respect to unknown rotation basis vectors and a translation vector as well as a few other unknowns like alpha and beta and the offset of the center. Don't forget that your rotation basis vectors are an orthonormal triplet!). This should enable solving for enough of the unknowns that more straightforward (matrix) algebra can handle the rest.*

Vector cross products *(denoted by* $\times$*) are very useful in computer vision and you should be familiar with them. (To compute them in Matlab use* `cross()`*). Further, it might be hard to solve this problem without them. There are many sources of information on cross products (e.g., Wikipedia, math text books) so it should not be too hard to find a source that you like. Let me know if this is proving difficult. Note that the vector cross product cares about the order of the operands, and you need apply the right hand rule to be sure about the direction of the resulting the vector. So, if you have followed the advice about working with a synthetic rotation matrix, and if you are getting the negative of a vector that you want, then check these two things. (If you have both wrong, the mistakes will cancel, and you will not notice!). Alternatively, your issue might be related to sign ambiguities and coordinate systems discussed next.*

Hint on signs (e.g., is the square root of one +1 or -1?) and coordinate systems. *If you are careful in your derivation, you might come to the conclusion that there are some sign ambiguities. The sign of* ρ *can be ambiguous because you may find it easiest to solve for its absolute value. The consequence of this is that one of you rotation vectors can be negated compared to the one you might be looking for. After spending quality time with Matlab, the instructor suggests the following strategy. After you have determined your rotation vectors, check that they are a right-hand coordinate system (if that is your preference). If they are not, then negate lambda, which implies you also negate the third rotation vector. To check for the right-hand versus left-hand coordinate system, you can see if* `r3=cross(r1,r2)` *or you can look at the sign of the determinant (using* `det()`*) of the rotation matrix because left hand systems have determinant –1. Also, as already mentioned above, checking for a left-hand system might be needed to fully understand synthetic data experiments.*

The sign of α *and* β *can similarly be confusing. If your intrinsics are consistent with Figure 1, and you choose a right-hand coordinate system, then you may find that you want* α *and* β *to be negative. The reason is that the Z axis is now pointing backwards from the looking direction. The sign choice does not affect M (which is why there is an ambiguity), but forces swapping the direction of two of the rotation vectors. Hence you can consider if the rotation matrix makes sense. Having said that, the most critical effect of having these signs flipped is that the results of (optional) part E might be confusing.*

*Finally, it is worth being aware that the eigenvectors you get from `eig()` have a sign ambiguity because the negative value of an eigenvector is equivalent to the one you are looking for, and might be what you get from `eig()`. However, for **this** problem and the line fitting problem, the sign **does not matter** as it cancels when needed.*

Once you are done this problem, you should take a moment to appreciate what you have accomplished. From a single image of a coordinate system you have figured out where the camera was, which way it was pointing, and its focal length. Almost magic!

Part E (optional, possible substitute for D, modest extra credit)

This problem requires results from D. If you want to try this problem, but do not have what you need from problem D, let the instructor know.

For this part, let's use the second image (the one with figurines). I recall that the TA who created these images was careful to place them without bumping the camera or the coordinate system, but if there seems to be issues that could be explained by the *M* found for the first image not applying to the second one, you can use the first image but then you need to specify your own targets for clicking.

Create three 3D segments for the three axes and project them into the image. Color them blue. This is a nice additional demonstration that calibration works (within reason). Now use the clicking process to find 2D points for the top of each sword. A point in a 2D image represents a ray in 3D. Compute the ray for each of the points you clicked (consult the hints below if you are stuck). The camera center is the endpoint for the rays is. Chop the rays to where they intersect the coordinate system (i.e., where they intersect the first of the planes XY, XZ, YY, which is simply what you get if you chop it with respect to all three of the planes in sequence). Provide these endpoints in 3D **($)**. Finally, draw the segments into the image in red for the red sword and green for the green sword. Provide the image with these extra lines **($)**, and explain the result of the projection of the two segments that intersect the ends of the swords **($)**.

We will now compute a different view of the axis and the two sword segments. Lets put the camera at (40,10,5), and have it oriented so that it is pointing in the negative X direction (corresponds to its third axis, Z', being in the positive X direction if you are using a right hand coordinate system). Further, its first axis, X', is pointed in the positive Y direction, and is second axis, Y', is in the Z direction. Provide the new camera matrix and explanations about how you found it **($)**. Provide the image of the axes and the two sword line segments as seen by the new camera, again with some explanation **($)**. Can you provide an intuitive definition for the point where the two lines are converging to in the image (**$**)?

Similar to what you did with the sword tops, click on a central point where each figurine contacts the ground. Using those points, and your knowledge of the camera, estimate the distance between the feet **($).** You need to assume that the figurines are on the ground plane (z=0), which you do not know for sure in general, although it is reasonable based on the scene. Of course, you should pretend that the graph paper is not there when you do this problem, but you can use it to check your answer.

Hints: To go from an image point to a ray in 3D world coordinates you have several options. If you have decomposed the intrinsic and extrinsic parameters, then you can create a segment in camera coordinates from the image point and the focal lengths. You could then use the extrinsic parameters to transform those points to world coordinates.

Alternatively, and perhaps more instructive than the above approach, you can solve the matrix equation for all world coordinates that map to that image point. This can be done

for an image point (u, v, 1) by solving the equations using Gauss-Jordan elimination. In Matlab, this can be done using the function rref(). The reduced form gives the answer where the free variable (here w) is set to zero. In homogeneous coordinates, w=0 represents a point at infinity.

*Now any solution to M*__P__ = 0 can be added to that first solution to get more. These solutions are given by the null space of M (Matlab function null()). Two of them give you homogeneous coordinates for points that can be demoted to regular coordinates, and then used to get a line in 3D world coordinates.*

For more information on Gauss-Jordan elimination and null spaces, see any linear algebra text or surf the web. (My preference Gilbert Strang's book).

Part F (optional, modest extra credit)

More fun with graphics

A) Add a specularity on the sphere.
B) Render the shadow of the sphere.

Note that these problems are not completely trivial For (A), you should develop equations for where you expect the specularity to be. You will need to consider the light position, the sphere, and the camera location. You may find it easiest to write the point on a sphere as X = X0 + R*n, where n is the normal which varies over the sphere.

What to Hand In

As usual, the main deliverable will be PDF document that tells the story of your assignment as described above. Ideally the grader can focus on that document, simply checking that the code exists, and seems up to the task of producing the figures and results in the document. But you need hand in your code as well as follows.

If you are working in Matlab (recommended): You should provide a Matlab program named hw3.m, as well any additional dot m files if you choose to break up the problem into multiple files. Do not package up the files into a tar or zip file–this messes up the D2L conventions.

If you are working in C/C++ or any other compiled language you need to discuss this with the instructor at least one week before the due date: You should provide a Makefile that builds a program named hw3, as well as the code. The grader will type:

```
make    ./hw3
```

You can also hand in hw3-pre-compiled which is an executable pre-built version that can be consulted if there are problems with make. However, the grader has limited time to figure out what is broken with your build. In general a C/C++ solution will require nonstandard libraries, and you should discuss with the instructor how they can be provided as part of your submission, or assumed to exist on the system that is used for testing.

If you are working in any other interpreted/scripting language (again you need to discuss this with the instructor at least one week before the due date): Hand in a script named hw3 and any supporting files. The grader will type:

```
./hw3
```