Math Problems

1. Consider the pinhole camera model, discussed in lecture

$$\underset{\mathbf{p}}{\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}} = \underset{\mathbf{K}}{\begin{pmatrix} \phi_x & \gamma & \delta_x \\ 0 & \phi_y & \delta_y \\ 0 & 0 & 1 \end{pmatrix}} \underset{[\mathbf{R} \mid \mathbf{t}]}{\begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} & \tau_x \\ \omega_{21} & \omega_{22} & \omega_{23} & \tau_y \\ \omega_{31} & \omega_{32} & \omega_{33} & \tau_z \end{pmatrix}} \underset{\mathbf{P_W}}{\begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix}}$$

Written more compactly as $\lambda\, p = K\,[R \mid t]\, P_W$

In one of our derivations in video lecture, we multiplied both sides of this equation by K^{-1} to get $\lambda\, K^{-1}\, p = [R \mid t]\, P_W$.

1a) Prove that K^{-1} is also an upper triangular matrix

1b) Letting

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \phi_x & \gamma & \delta_x \\ 0 & \phi_y & \delta_y \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

write down equations for the values of x' and y' as a function of x,y and the elements of K .

Note: the values x', y' are coordinates of a 2D point in a "*normalized camera*" coordinate system with focal length 1, skew = 0, and coordinate system origin at the geometric center of the image plane. It is quite common in stereo, motion, and 3D reconstruction derivations to use normalized camera coordinates to simplify the math.

2. Ignoring camera intrinsic parameters to just focus on 3D camera pose, consider the following equations relating one world coordinate system with two local coordinate systems of two cameras C1 and C2

$$P_{C1} = R_1 P_W + t_1$$

$$P_{C2} = R_2 P_W + t_2$$

where P_W is a 3D point represented in world coordinates, P_{C1} and P_{C2} are the same point represented in each camera's local coordinate system, and R_1,t_1,R_2,t_2 are the pose parameters for each camera.

Consider now the relative pose between the two camera views, represented by rotation Ω and translation τ, such that

$$P_{C2} = \Omega P_{C1} + \tau$$

Derive equations for Ω and τ as functions of the camera pose parameters R_1,t_1,R_2,t_2 .

Note: from the relative pose parameters Ω and τ we would be able to compute the "essential matrix" E that linearly constrains point matches between the two image by treating the two cameras as a stereo pair. This would open up an alternative set of methods to search for image point correspondences and to remove outlier matches.

3. In lecture we will discuss camera calibration / pose estimation. Specifically, given a set of N known 3D to 2D point correspondences where world point (u_i, v_i, w_i) projects to image point (x_i, y_i), for i=1,2,...,N

p K [R | t] P_w

$$\lambda_i \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{pmatrix} \phi_x & \gamma & \delta_x \\ 0 & \phi_y & \delta_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} & \tau_x \\ \omega_{21} & \omega_{22} & \omega_{23} & \tau_y \\ \omega_{31} & \omega_{32} & \omega_{33} & \tau_z \end{pmatrix} \begin{bmatrix} u_i \\ v_i \\ w_i \\ 1 \end{bmatrix}$$

we would like to determine the intrinsic parameters in K and the extrinsic (pose) parameters in R and t.

Simon Prince's textbook contains an iterative solution method in Section 14.5 that alternates between solving for pose given known K and solving for K given known pose, but he admits this solution is slow to converge and not very practical. As I mentioned in my derivation notes (in lecture and posted on canvas), there is an alternative approach called DLT or "Direct Linear Transform." We will explore that in the following problems.

First, note that if we can combine both K and [R | t] as Pmat = K * [R | t] to get an arbitrary 3x4 matrix Pmat such that , for i=1,2,...,N

p Pmat P_w

$$\lambda_i \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{pmatrix} \begin{bmatrix} u_i \\ v_i \\ w_i \\ 1 \end{bmatrix}$$

Derive a homogeneous linear system of equations A x = 0 that can be solved by SVD to get the parameters x = [p11 p12 p13 p14 p21 p22 ... p44]'. The derivation is similar other problems we derived in lecture. I specifically want to know what the elements of the matrix A are, and how many rows and columns it has.

Applied Problems

4. For the remaining problems, we will be using a dataset and matlab starter code that I have provided. Files rubik1.jpg and rubick2.jpg are two images of a rubik's cube on a desktop. The file rubikPoints.mat contains a set of 37 measured 3D corner points on the cube in array Pworldpts, and the corresponding 2D pixel locations of those points for the two images in arrays Pimagepts1 and Pimagepts2. The file homework4_startercode.m is matlab starter code.

   Use your derived A matrix from problem 3 to write code for function calibrateDLT.m which takes an array of 3D points and corresponding array of 2D points and returns 3x3 intrinsic parameter matrix K and 3x4 pose matrix [R | t].

   Follow the sketched out solution in my posted handwritten course notes (that we also went over in lecture). Specifically, your function should:

   - Compute matrix Pmat using SVD based on your solution to problem 3.
   - Verify that Pmat * [u;v;w;1] for one of the world points in the rubik's cube model has third coordinate > 0, implying the camera is facing towards the cube (rather than looking in the opposite direction, away from the cube). If the third coordinate is < 0, negate all the values of Pmat. Note that this step is needed because A x = 0 has two solutions, x and –x, and we can't guarantee which one SVD is going to give us.
   - As in the course notes, partition Pmat into [Pa | Pb] where Pa is 3x3 and Pb is 3x1. From the setup for problem 3, we know that Pa = K R, where K is an upper triangular matrix, and R is a rotation matrix. We can solve for K and R in closed form using rq decomposition, a linear algebra function closely related to qr decomposition. Since matlab does not have rq decomposition, I have provided a function rq.m to do it for you. We now have K and R.
   - Finally, solve for t based on Pb, and the values for K and R computed above, so that Pmat = K * [R | t]. You will return [R | t] as the output pose matrix (and of course K is the output K matrix).

4. (Continued)

Test your solution using the starter code. I have included code for projecting world points using your computed K and pose matrices, and comparing them to the originally measured image points while computing RMS image projection error.

In addition to turning in your code for calibrationDLT.m, please answer the following questions:

4a) What are the RMS projection errors that are computed for each of the two images?

4b) Do the two K matrices computed for the two images look reasonable? Is the principle point $\delta x, \delta y$ somewhat near the geometric center of the image? Is skew γ small? Is aspect ratio $\phi x / \phi y$ close to 1?

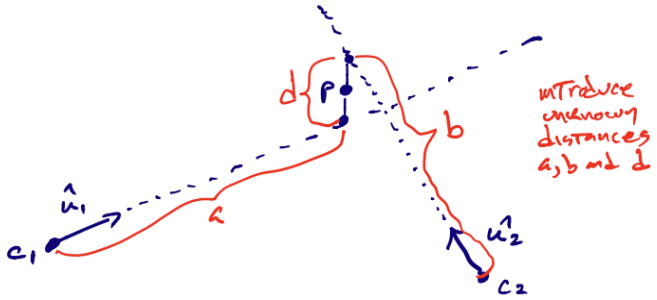
4c) Compute and report the horizontal and vertical fields of view of each camera.

4d) The images were actually taken by the same camera, so in reality should have the same intrinsic camera parameters. Explain why it would be difficult to modify your DLT method to constrain the two camera matrices K1 and K2 to be the same.

4e) If K1 and K2 were the same, then inv(K1)*K2 and inv(K2)*K1 should yield the identify matrix. What are the matrices produced for those cases when you use your computed K1 and K2? Noting that taking norm(eye(3)) = 1, [i.e. the svd norm of the identity matrix is 1], report norm(inv(K1)*K2) and norm(inv(K2)*K1) for your computed matrices. Are the values close to 1?

5. The next step in the starter code is taking the calibration values K1, [R1 | t1], K2, [R2 | t2] computed in problem 4 and doing two-camera triangulation to try to see how accurately we can recover the 3D rubic cube world points from the 37 pairs of corresponding 2D points in the two images.

   Wriite function triangulateDLT that takes the two sets of camera parameters and two 2xN arrays of 2D image points (see starter code for argument order) and returns a 3xN array of 3D triangulated points along with a 1xN array of distances that measure how close the 3D viewing rays for each point come to each other (if the two rays exactly intersect, this distance is 0, otherwise it is the perpendicular distance between them). Use the algorithm I sketched out in lecture and in the handwritten lecture notes to set up a linear system of equations for each pair of point correspondences, to solve for a,b,d of the "circuit diagram"



   which allows you to solve for 3D point P closest to both viewing rays. Note that abs(d) will be the perpendicular distance between the viewing rays (d could be positive or negative depending on what direction unit vector u3 is pointing, so taking absolute value makes sure the distance is nonnegative).

   Use the starter code to display a 3D plot of your computed triangulated points and the actual ground truth 3D point locations. The "rotate3D" command in the code allows you to use the mouse to rotate the points in 3D. Make sure what you see corresponds to what you think you should see.

   In addition to turning in your code for triangulateDLT.m, please answer the following question:

   5a) Compute and report the 3D (root mean square) error between the triangulated 3D points you computed and the actual ground truth model locations.

6. ***This following part is completely optional***

I have included code that extracts additional feature points in both images using the "KAZE" feature detector, and then matches them to get a set of hypothesized point correspondences. If you are unable to run this part of the code for whatever reason (like having an out of date version of matlab), I've included the set of detected point correspondences in variables newpoints1 and newpoints2.

6a) Use your triangulation code to compute 3D points for these additional point correspondences, display using plot3 and rotate3D.

6b) Note that incorrect correspondences give some very bad outlier points in the above triangulated 3D point set. Choose a threshold on the distances returned by the triangulateDLT code that removes most of the outliers, leaving a cleaner set of 3D points. Again, use the mouse to interact with the 3D plotted points window to get a nice view of the clean 3D point set. Note that you *should* be able to make out some of the other surfaces in the scene, such as the flat piece of typewritten paper on the desk, and the curves surfaces of the coffee cups. Cool, right? By completing this homework, you have implemented a very simple version of what COLMAP is doing when it does sparse reconstruction.

Note for those interested: the way we are filtering out bad matches using distances between 3D viewing rays works OK in this case, but doesn't work as well for scenes where points are spread over a large range of scene depths. This is because, for the same amount of 2D point detection error in the images, points farther away from the camera will tend to have larger perpendicular distances between their triangulated viewing rays, because as the rays get farther from the camera they have more ability to diverge away from each other. Another way of detecting bad triangulated points that works better in that case is to project the triangulated point into the image and measure the largest pixel distance between the projected point and the two original image points used for triangulation. If this distance is too large, you can reject the point match as being incorrect.

What to hand in:

Submit derivations, code, screenshots, answers as a single pdf or zip file in Canvas. Please make it easy to see where each answer is. A single pdf answering things in order, with equations and pictures and code pasted in, would be ideal. For example, the pdf assignment description you are reading right now was "written" in powerpoint before printing out as pdf.