COMPSCI 527 — Homework 6

Due on December 7, 2015, 11:59pm

This assignment is a set of instructions for your final project for COMPSCI 527. In this context, a project differs from a regular homework assignment in that it is more open-ended in its goals, and in what constitutes an acceptable "solution."

IMPORTANT INSTRUCTIONS: Work in pairs on this project if you can, alone if you must. Each pair sends in one document. Email a single file in PDF format to `tomasi@cs.duke.edu` (do *not* email it to Hyunsoo) by the deadline above.

The Software

Software is provided with this project to create a simulated world with an object, two cameras, and the coordinates of salient object points in images of the object taken by the two cameras. The software then adds Gaussian noise to the image-point coordinates and runs the eight-point algorithm on the noisy input. Finally, it computes several measures of result quality, as described below. The code is provided "as is," in the sense that if it does not work for your purposes it is your responsibility to modify it as needed, or to rewrite it if you prefer. You are then asked to run various experiments and report about the results.

Unzip the code into some directory, make that directory the current directory for MATLAB, and type `RUNME` at the prompt. The code will then do the following (item headers below are the names of the functions that perform each step):

`world`**:** Create a synthetic world with three faces of a cubic box with some squares painted on it, placed near the origin of the world reference system (no need to create faces we cannot see). Also make two cameras, as well as the coordinates of the visible cube vertices and square corners in the images taken by those cameras. Box coordinates are world coordinates, and each camera has its own reference system, so there are three reference systems in all.

The input `side` to `world` is the side length of the box. The input `t` is a 3×2 matrix whose columns are the vectors from the origin of the world reference system to the centers of projection of the two cameras. Side length and translations are in arbitrary units. However, the function `world` scales camera positions and box, so that the distance between the two camera centers becomes one after scaling. This will make performance evaluation easier. If you change `t` (within reason), then the function `world` will re-orient the cameras so they point towards the box, and adjust the focal distances so that the image of the box fills most of the image.

The output `img` from `world` contains enough information to draw the two images of the box. So `img` does not just contain point coordinates but also connectivity information to draw the various patches that make up the image, and their colors. Specifically, `img` is a 2×2 array of structures. Two of these describe the corners of the squares painted on the faces of the box (one structure for each image), and the other two describe the visible faces of the box (one structure per image). If you use the display and drawing functions provided, you need not concern yourself with the fields `faces` and `colors`, but just with `img(1, 1).x` (the homogeneous image coordinates of the square corners in the first image), `img(2, 1).x` (the homogeneous image coordinates of the visible box vertices in the first image), and the two analogous entries `img(1, 2).x` and `img(2, 2).x` for the second image. So the first index of `img` selects between square corners and box vertices, and the second subscript is the image index.

The output `box` describes the box. This description is formally the same as that of, say, `img(:, 1)`, except that the point coordinates are in three dimensions instead of two (in the world reference frame), and a capital `X` is used rather than a lowercase `x`.

The output `camera` is an array of two structures that describe the two cameras, with the following fields:

`resolution` is the size of the image in pixels, width first.

`G` is the rigid transformation from the world reference system to the camera reference system, in the format described in the class notes on homogeneous coordinates.

`Ks`**,** `Kf` are the internal camera calibration matrices K_s and K_f described in the class notes.

`P` is the camera projection matrix $P = K_s K_f \Pi G$ where

$$\Pi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} ,$$

as described in the class notes. Note that the world reference system is not attached to one of the two cameras, so G differs from the identity matrix for both cameras.

Feel free to peruse the file `world.m` to understand these structures in greater detail, but lines 33 to 40 of `RUNME` show you how to extract the homogeneous coordinates `X` of all the world points from `box` and the canonical coordinates `x1` and `x2` of all the image points from `img`, and this is really all you need to know about these structures. Of course, **corresponding columns of** `X`, `x1`, `x2` **contain corresponding points.**

`addNoise`: Add Gaussian pseudo-random noise to the images. The noise has standard deviation `sigma` pixels. Currently, `sigma` is set to zero, and you will experiment with different values.

`longuetHiggins`: Compute the rigid transformation from camera 1 to camera 2 and the scene structure in the reference system of camera 1 from the two images using the eight-point algorithm described in the class notes.[1] Since structure can only be recovered up to a global scale factor, the solution returned by this function is normalized so that the distance between the two centers of projection is equal to one.[2]

`motionError`: This function compares true and computed motion. Please look at the body of the function if you are interested in how the error is computed. Both rotation and translation errors are reported in degrees (translation is a unit vector, so errors are in degrees for that as well).

`structureError`: This function returns a measure of similarity between true and computed structure (the shape of the box). To this end, the function centers the true and computed cloud of 3D points around their centroids, scales them so that their RMS scale is 1, rotates one so as to match the other as well as possible by solving the Procrustes problem described in the class notes[3], and then measures the RMS residual between the transformed clouds. The result is divided by the square root of the number of points, so that the unit of measure for the resulting error is in units of length per point, where the overall (RMS) size of the object is one unit. So a structure error of 0.015 means that the average error is 1.5 percent of the overall size of the object. This is a generous definition of structure error, as significant efforts are made to align true and computed shape before measuring their discrepancy.

`reprojectionError`: This function measures the RMS error between the original image points and the points projected from the solution of 3D reconstruction. The units are pixels per point.

Other functions provided display results in various ways, as you will see when you run the code. These figures are only drawn as sanity checks, to see what happens as you run your code. Please try to understand what these figures mean, and read the code when needed. Keep in mind that structure is displayed in different reference systems for different displays, so you should drag the mouse around in the structure figures to rotate the results and view them from useful viewpoints. You will likely turn off these displays when you run your experiments.

Your Task

Your are to modify this code to analyze empirically what happens as you vary the noise level and other parameters. There are several questions to address, and I leave to you the choice of which questions to ask. Here are some examples. Words in square brackets list alternatives for you to pick from:

- It does not take very large amounts of noise before the reconstructed shape becomes unrecognizable. Which types of error fail first when this happens?
- How do [rotation, translation, structure, reprojection] errors depend on image noise?
- How does the [angle between the camera optical axes, distance from the camera to the scene] affect the errors above?
- Noise is in pixels. What happens if you change image resolution but keep `sigma` fixed?
- If you feel adventurous, modify the code in `box` or `world` to squish the box into an increasingly flatter object. Does that affect results? How and to what extent?

Some questions that appear intriguing at first may give uninteresting answers in practice, so give yourself some time to explore alternatives. Asking good questions is as important as answering them well. If something surprises you, describe it and try to explain.

When you answer questions like the ones above, keep in mind that 3D reconstruction is an ill-posed computation, in which small input errors can lead to very large output errors. In particular, large values (say, more than 3 pixels standard deviation) of noise may lead to meaningless results. So you will have to find reasonable noise ranges and values by trial and error.

In addition, each point on each of your plots should be the average from running the experiment multiple times (say 30 times or more) with the same parameter values (same `sigma` as well), but different instances of pseudo-noise. Without this, your plots will look very random. This means also that **you may need some time to design and run your experiments, so do not start this project late.** Write a single script file that generates all the plots and figures, so you can make modifications and run everything again without confusion.

[1] If you supply a third output argument, then this function also returns structure in the system of reference of camera 2.

[2] As stated earlier, this is also how `world` normalizes everything.

[3] Appendix A of the notes on the eight-point algorithm.

What to Hand In

The main product of your work should be a clear, well-organized document that answers some of the questions above—or different ones if you like—mainly in the form of readable plots and clear, succinct text. I am not looking for answers to many questions. Just pick two or three questions but answer those carefully, with well-designed plots and good descriptions and explanations in your text, written in good English. You will be evaluated on how well you reason about the problem, rather than how many pages you write. A "failed" experiment is still rewarded with a high grade if the reasons for the failure are well explained.

It is important to design plots that support your reasoning well, and convey information clearly and succinctly. Once you have enough plots to say what you need to say, adding more plots will likely make your document worse. Try to superimpose plots on the same diagram when this makes sense, and as long as the result is readable. Make sure you label all axes and plot lines, and add meaningful, detailed legends and figure captions. Always specify units for all axes, either with the axis labels on in the figure captions. No unit, no credit.

Your document should have a brief introduction that explains what you set out to discuss. You need not explain how the eight-point algorithm works, except to the extent that your explanation is useful for your arguments. Then there should be a main section with your results and discussion, followed by a brief but clear and non-perfunctory conclusion that states what you have learned from your work, good or bad. When referring to a figure in your text, do so by also including the figure number: "*Figure 3 shows that...*"

A good document with 4-5 pages of high-quality text is close to optimal. Good thinking about the problem is the main criterion, but good structure, syntax, and grammar matter as well, and will be part of the evaluation criteria.

Do not hand in any code. All I will read is a single PDF file that you should email to me (`tomasi@cs.duke.edu`) by midnight of the deadline. It is highly advisable to make the PDF with LaTeX, unless you can create a document of comparable typographical consistency by other means (unlikely).