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

Assignment Two

Due: 11:59pm (*) Tuesday, Sept 13.

(*) 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: 6 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 three parts. The second part is only required for grad students. Modest extra credit is available to undergraduates who do some of the grad student part. To simplify things, you should hard code the file names in the version of your program that you hand in. You can assume that if the grader needs to run the program, they will do so in a directory that has the input files. Specific deliverables are flagged with ($).

Part A

The file

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

is an ASCII file containing a 101 by 3 matrix representing estimates for the spectral sensitivities of a camera that the instructor used for his PHD work. There are 101 rows because the spectrophotometer used to determine them samples light at wavelengths from 380 to 780 nanometers in steps of 4 nanometers, inclusively.

1. Generate 1600 random light spectra as 101 element row vectors. The standard random number generator is fine, as the spectra need to be positive. To make grading and debugging easier, precede your call to `rand()` with a call to `rng()` with the argument 477 (i.e., `rng(477)`). Convince

yourself that your experiments are repeatable. (If you are working in C and using the IVILAB library, you can use `kjb_seed_rand(477, 577)`);

Note that rng() is not available on old versions of Matlab.

Compute the (R,G,B) responses for the 1600 generated light spectra using the sensors linked above. The deliverable will be specified shortly---first some informational comments.

The standard random number generator provides values between zero and one, but if you make it into a light spectra vector, you are pretending that someone measured these values. But what are the units? You should understand that there is an arbitrary scale factor implicit in your light spectra because of the arbitrary range of [0,1]. So, you can pretend that your light spectra are some constant, K, times real spectra in physical units. In practice we often ignore this scale, as the absolute intensity of light is often somewhat arbitrary anyway (e.g., its units are a consequence of the photographer adjusting the aperture), but you should understand that it is there.

Similarly, the provided sensitivity curve has implied units. They come from a calibration experiment for a particular spectrophotometer, and it converts spectra, as measured by the by that instrument, into RGB for the camera settings used on the day of the experiment. To be specific, the numbers would have to be of the order of 1e-4, not the order of 1.0 as given by rand(), to have RGB values in the range (0,255).

The bottom line is that you do not necessarily expect to get sensor responses in the range of (0,255) unless we had adjusted the sensors before hand to make that so. Determine a scale factor, K, that scales (multiplies) your 1600 (R,G,B) so that the maximum of any of the (R,G,B) is 255. (You do not need to report this value).

Multiply the randomly generated light spectra by K, and regenerate the (R,G,B). Verify that the max R, G, or B is now 255.

To visualize the (R,G,B) so the instructor can easily see the data, we will pretend that the 1600 spectra came from the squares on a 40 x 40 grid. The first 40 spectra correspond to the first row, the second 40 to the second row, and so on. We will create an image that we would expect if each of the squares occupied 10 x 10 pixels.

Specifically, create a 400 by 400 color image made from 1600 10x10 blocks of uniform RGB. The RGB of the 40 blocks in the first row should be the first 40 RGB values you have generated, the next row should have the next 40 values, and so on.

Your computer program should display this image and you should put it into your PDF as the answer to this question with caption and any needed context. **($)**

> *Hint: If your image is not what you expect, check the data types. You may have to cast the values to create an image for display.*

2. You now have a simulation of responses with no noise. Now use the least square method developed in class to estimate the camera sensitivities. Plot the "real" sensors and the estimated ones on the same plot **($)**. Now compute the following two different kinds of errors. First compute the RMS error between each of the three estimated sensors and the actual ones (provided in rgb_sensors.txt) and report the values (**$**). Also compute the RMS error between the R, G, B as calculated using the actual sensors and the estimated ones (**$**). This is a measure of how well your estimated sensors can (re)construct RGB.

   > *For those that have forgotten (or never know), RMS (root mean square) error between two sets of numbers is the square root of the sum of the squared differences. Note that since sqrt() is a monotonically increasing function, minimizing the sum of squared error is that same as minimizing the RMS.*

3. Now simulate measurement error by adding a different amount of normally distributed (Gaussian) noise to each of the (R,G,B) values for each of the 1600 colors. Start with noise of the order 10 by setting the standard deviation of the noise to 10. Since you scaled the data to (0,255), this is about 2%.

   Hint: The easiest way to get the noise values is using the Matlab function `randn()`.

   Comment: I am suggesting normally distributed noise because this is the assumption that the least squares solver makes, and I would like you to understand how to put this assumption into computational experiments using synthetically generated data. If you are interested, you might want to investigate some other noise models such as uniform, or either normal or uniform with some outliers. Doing so might be featured in subsequent assignments.

   You now have a simulation of a camera calibration experiment. You have light spectra, and you have a bunch of responses with some noise. Now use the least square method developed in class to estimate the camera sensitivities. Plot the "real" sensors and the estimated ones on the same plot **($)**.

   As described, if we were to increase the level of noise, then we will start to get lots of negative values and values beyond 255. This may or may not be the simulation we are interested in, as real data from sensors will be "clipped" to be in the range of [0,255]. So, implement a second case where the resulting RGB level (not the noise) cannot be negative or more than 255. Again plot the real sensors and the estimated ones on the same plot **($)**.

   As above, compute the RMS error between each of the three estimated sensors and the actual ones (provided in rgb_sensors.txt) and report the values (**$**). Also compute the RMS error between the R, G, B as calculated using the actual sensors and the estimated ones (**$**).

4. Now, let us set the amount of noise (standard deviation) to *i*10* where *i* will start at *0* and go up to *10*. The previous questions provide most of the results for i=0 and i=1, and it is OK if your final implementation does those cases as part of this one, but you still want to organize your report as implied by the question numbers. For this part, instead of computing three numbers for each error type, compute a single overall RMS error for each error type.

   The overall RMS is the square root of the sum of all the errors across R, G, and B, and it may be helpful to convince yourself that this is the same as the RMS of the RMS values for R, G, and B.

   In your PDF, provide the plots for the sensors for *i=5* and *i=10* **($)**. Also provide the 11 results for both types of error, with and without clipping. (An 11-by-2 table would be one way to report your results) **($)**. Comment on the results, considering both the effect of modeling clipping, and the increasing noise level **($)**.

Part B (only required for grad students).

Simulation is not reality. The file

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

is a file of 598 real light energy spectra. Note that wavelength is now across columns (opposite to rgb_sensors.txt). The file

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

are corresponding real (R,G,B).

5. Estimate the camera sensitivities using this data. Again plot the real sensors and the estimated ones on the same plot. Again report the two overall RMS error measures (**$**). Hopefully you will find (and report) that your sensors are terrible! Can you explain this? ($).

   Hints. Some ideas to address this follow. You might consider that the real light spectra came from a limited number of sources, through a limited number of filters, hitting a limited number of

surfaces. Further, the reflectance spectra of most surfaces is smooth, which implies that they have limited dimension. For example, you can reconstruct them from a small set basis functions such as a handful of sines and cosines (Fourier series). Or you could explore computing the condition number (Matlab cond()) of the light spectra matrix and a similar sized random one. Or you might use PCA (see assignment one) on the light spectra matrix and a similar sized random one, and tell a story about how the diagonal of the covariance matrix differs (if you find the difference to be substantive).

6. We can make things better by doing **constrained** least squares. For example we can insist that the solution vector is positive. Implement this. If you are doing this in Matlab, you will need to use a function like **quadprog.** (The way quadprog() is set up is a little different than what you might expect. Here is a link to a simple example:
   http://kobus.ca/teaching/cs477/examples/quadprog_example.m
   which may help if you are having trouble. Again, plot the results ($) together with the real sensors.

   Tip: So far you might have taken the convenience fitting the sensors together. However, for this part and the next you will find it easier to set up the call to quadprog() if you fit them independently. (IE, red, green, then blue, perhaps in a loop).

7. Hopefully you will now have positive sensors, but they are still weird. The problem is that they are not smooth. We can promote smoothness by pushing the derivative of the sensor curve towards zero. In this paradigm, this amounts to introducing equations that set the derivative to zero, and thus increasing deviations from perfectly smooth lead to greater error that is traded off with the error of fitting (which we are already using). Before reading on, consider how you might arrange this.

   OK, I am assuming that you have thought about this, and want to check your ideas.

   Consider a matrix, M, that implements a derivative operator, which, when using vectors to represent functions, can be approximated by a vector **D** of successive differences. Consider first, a row of that matrix $\mathbf{m}_i^T$ that computes the difference between the *i* and *i+1* element of a vector **R** by

   $$D_i = R_i - R_{i+1} = \mathbf{m}_i^T * \mathbf{R}$$

   Work out for yourself what $\mathbf{m}_i^T$ must be to compute that difference. This should enable you to construct a "differencing" matrix M that computes a vector **D** of successive differences by D=M*****R**.** You should ignore fence post problems (i.e., you can compute 100 differences for a 101 element spectra). Further, you should introduce a scalar multiple of M which we will refer to as λ (lambda). If we set $(\lambda M) * \mathbf{R} \cong 0$ in a least squares setup, we can promote smoothness on **R**. The value of lambda will modulate the amount of smoothness.

   Augment your light_spectra matrix with another 100 rows that is the differencing matrix. Augment your response vectors (R, G, and B) to have the desired result (zero).

   Verify for yourself that tweaking lambda adjusts the balance of fit and smoothness. You should be able to produce very smooth curves that do **not** resemble your sensors, and curves approaching the ones you found in the previous part (sensors with positivity), where $\lambda = 0$ should give exactly the same sensors as before.

   Provide plots for 5 different ascending values of lambda to illustrate the control you have on the output **($)**. Make sure that you have two plots for lambdas that you consider too small, and two for lambdas that you consider too large. The third plot should be a value of lambda that you think is pretty good. Note that the curve for blue (leftmost, covering the smaller wavelengths) cannot be fit

particularly well. Don't worry about this. All plots should have both the real sensors and the estimated ones on them. The plots should have the value of lambda in the title.

Clarifying comment: This smoothing approach works both with regular least squares and constrained least squares. For this question, do the second, keeping the sensors positive.

Note: Lambda implements the desired effect because in least squares any row can be weighted by simply multiplying both the row and the response by a scalar. This is a very useful thing to have in your toolbox. Think about the error function and make sure you understand why this works.

Part C

Below are links to two images. Dump these into a into a drawing program, and draw enough lines over the image to make a case that the image is either approximately in perspective or not. (Have a look at the building examples in the lecture notes if this is not making sense). You should understand and state your assumptions, and explain your reasoning. To get you started: You can assume that that the chandelier is perfectly symmetric.

Your deliverables for this part of the assignment is entirely within the your PDF (no code). You will provide images with lines drawn on them and an explanation of what you conclude and why **($)**. For the chandelier image, you must also put small circles (i.e., "dots") that make it clear where the lines come from. In the case of the building, it will be generally clear how you drew the line, but in the case of the chandelier image this will not be the case. Help the grader by showing the points that you are drawing lines through. If you color code the points and/or lines, this will help provide points of reference in your explanation.

http://kobus.ca/teaching/cs477/data/building.jpeg
http://kobus.ca/teaching/cs477/data/chandelier.tiff

Part D (Optional challenge problems)

Challenge problems are not for credit. They are for students who are especially interested in the subject, and who are comfortable with their understanding of the basics. They can be difficult, and I recommend being careful about spending too much time on them.

In class we learned that under perspective projection, parallel lines (generally) converge to a point. Can you prove this?

We also learned that under perspective projection, the vanishing points for sets of coplanar parallel lines are collinear. Can you prove this?

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 hw2.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 leas one week before the due date: You should provide a Makefile that builds a program named hw2, as well as the code. The grader will type:

```
make    ./hw2
```

You can also hand in hw2-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 leas one week before the due date): Hand in a script named hw2 and any supporting files. The grader will type:

```
./hw2
```