Spring 2020 - CS 477/577 - Introduction to Computer Vision

Assignment Two

Due: 11:59pm (*) Thursday, January 30.

(*) 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: Approximately 5 points

This assignment must be done individually

General instructions

You can use any language you like for this assignment, but unless you feel strongly about it, you might consider continuing with Matlab.

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 third 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 (**$**).

Learning goals

- Begin learning how to create synthetic data to test your computational process
- Solidify your understanding of spectral image formation (forward process)
- Using least squares to invert the forward process to infer sensitivity functions
- Solidify your understanding of image gamma transformation
- How real data might not match implicit assumptions in synthetic data (grad students)
- Using constrained least squares to put more knowledge into the inference (grad students)

Part A

The file in the D2L data directory, rgb_sensors.txt (also 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. Begin by providing a plot of the sensors **($)**. Then 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)`). You will need to adjust this for difference languages. Convince yourself that your experiments are repeatable by resetting the random seed.

   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 that instrument, into RGB for the camera settings used on the day of the experiment. To be specific, your generated 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 sensor responses in the range of (0,255) unless we had adjusted the sensors beforehand to make that so. Determine a single scale factor, K, that scales (multiplies) your 1600 randomly generated light spectra so that the maximum of any of the resulting (R,G,B) responses 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.

   The main deliverable for this problems is an image that visualizes all 1600 (R,G,B) at once to verify that your data makes sense. To do so, we will pretend that the 1600 spectra came from 1600 different squares laid out on a 40 x 40 grid. For example, the RGB for each of the first 40 spectra correspond to the 40 squares on the first row, the second 40 RGB correspond to the squares on 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 (you can use bigger blocks if you like---the main issues is to make them big enough so that one can see the variety of colors). Specifically, assuming 10x10 blocks, you would create a 400 by 400 color image made from 1600 10x10 blocks of uniform RGB. If you are still confused, refer to Figure 1.

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 (e.g., all white or all black), check the data types. You may have to cast the values to create an image for display. The function `imshow()` *can work with either* `uint8` *or* `double`*, but in the first case, the range is [0,255] and the second case it is [0,1].*

*Hint: You will notice that the data is biased towards blue, which is fine for this assignment. The bias is because the camera balance is for light that is biased away from blue, as is the case for indoor lighting. Light spectra that are generated the way we are doing it here are **not** very natural!*

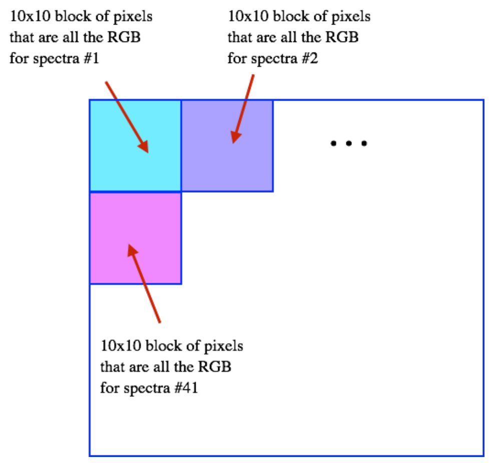
10x10 block of pixels that are all the RGB for spectra #1
10x10 block of pixels that are all the RGB for spectra #2
. . .
10x10 block of pixels that are all the RGB for spectra #41

Figure one. Visual version of the layout instructions for problem one.

2. You now have a simulated responses for 1600 light spectra given the three sensitivities. Now we will explore recovering those sensors, pretending that we do not know them until it is time to check the answer (more precisely, compute the error). Begin by using the least square method developed in class to estimate the 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 (**$**). The second error is a measure of how well your estimated sensors can reconstruct the RGB.

   For those that have forgotten (or never knew), RMS (root mean square) error between two sets of numbers is the square root of the sum of the squared differences divided by the number of them to get the mean. Note that since dividing by the number of items, as well as sqrt(), are monotonically increasing function, minimizing the sum of squared error (i.e., least squares) 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 by setting the standard deviation of the noise to 10.

   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) is forced to be in this range by setting values outside the range to the appropriate limit value. (An easy way to do this is with `min()` and `max()`). 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 (sensor and response), 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 divided by the number of them, 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.

5. Remind yourself about Gamma Calibration in Lecture 3. The idea is that when the gray apple matches the stripes viewed at a distance, this is when the raw intensity of the apple is ½ between black and white. Suppose that when the user matched the gray apple with the stripes viewed at distance, the gray value was 80 on a scale from 0 to 255. Assuming that this non-linearity can be fixed with a gamma correction, compute the value of gamma. Explain what you did and why, as well as reporting the answer ($).

   This question is not meant to be particularly long or time consuming.

Part C (only required for grad students).

Simulation is not reality. The file in D2L light_spectra.txt (also, 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 responses.txt (also, http://kobus.ca/teaching/cs477/data/responses.txt) are corresponding real (R,G,B). For comparing estimated sensors, we will assume that the sensors are the ones used in Part A, even though they are not perfect, as they were also estimated from data.

6. Estimate the camera sensitivities using this data. Again, plot the real sensors and the estimated ones on the same plot **($).** Again, report the two (sensor and response) overall RMS error measures (**$**). 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 dimensionality, and only appear to have dimensionality 101 due to noise. For example, you can reconstruct them from a small set of 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 an identically sized random one. Or you might use PCA (see *assignment one) on the light spectra matrix and a random one, and tell a story about how the diagonal of the covariance matrix differs.*

7. We can make things better by doing **constrained** least squares. For example we can insist that the solution vector is non-negative. 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. The file quadprog_example.m in D2l/Codo (also
   http://kobus.ca/teaching/cs477/examples/quadprog_example.m)
   is an example that may help if you are having trouble. Again, plot the results **($)** together with the real sensors. Also, again report the two (sensor and response) overall RMS error measures and comment on the changes from the previous experiment (**$**).

   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).

8. Hopefully you will now have non-negative 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 computing 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 **S** by

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

   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***S**. 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 **($)**. Make sure that you have two plots for lambdas that you consider too small, and two for lambdas that you consider too large. The third plot should be a value of lambda that you think is pretty good. Note that the curve for blue (leftmost, covering the smaller wavelengths) cannot be fit particularly well. Don't worry about this. All plots should have both the real sensors and the estimated ones on them. The plots should have the value of lambda in the title. Also, again report the two (sensor and response) overall RMS error measures and comment on the changes from the previous experiment (**$**).

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.

9. (**, i.e., this is an optional problem that will be considered for modest extra credit). Inspection is one way to decide how to set lambda, but it is unsatisfying because there is no principle that you can explain to someone (e.g., a reviewer) about how to set lambda. One solution is some form of cross-validation. Here you divide your data in K disjoint subsets, and use K-1 one of them for "training" and the "held-out" one for "validation". For each training set, you step through lambda to get sensors of increasing smoothness, and for each, compute the RGB error on the validation set, and remember both the lambda for minimal error, and the resulting error. Each split provides one value of lambda, and the average of them (or their median, or even their distribution) is what you would use on truly new data, and the standard error of the mean over the K validation tests would estimate your error[1].

You might notice that the error computed in this way is not a completely fair estimate of the error because it was reported for data that was used to tune it. So, while the lambda is suitable for using on new data, you cannot use the error estimate to compare to another method in a paper, and if you did so in a paper, the reviewers might declare foul. The fail-safe method, which all machine learning and computer vision researchers ***must understand*** *is to divide your data into training and* ***test****, and then further divide training data into initial training and validation sets. Then, the process above for computing lambda above, explained in terms of initial training and validation, can be evaluated on data that was not involved in any way in computing it. If you have time, and you have never done this before, you are encouraged to use improve approach instead of the above.*

Notice that if you use K splits each time, your complexity is K^2. It is almost as good, and still valid, to use only one split in the inner loop. This is the standard way to set hyper-parameters, such as the number of epochs (iterations) when training a neural network.

Experiment with one of the two alternatives above, or perhaps a different strategy involving cross-validation on this task, using K=10 for the number of folds used for computing the error estimate.

As is common for extra problems, I have not actually implemented this myself on this data, so I am curious to see what those that try it out find.

[1] To calculate this error you compute $\sigma/\sqrt{K}$ where σ is the standard deviation of the K errors computed on the test set. One should always be shy about reporting any number without an error estimate, and this is one way to get one, but you also need to give the reader a hint of how the estimate was made and perhaps relevant assumptions.

What to hand in

As usual, the main deliverable will be PDF document hw02.pdf 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: You should provide a Matlab program named hw02.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 Python: You will provide a Python program named hw02.py as well any additional dot py 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 should provide a Makefile that builds a program named hw02, as well as the code. The grader will type:

```
make    ./hw02
```

You can also hand in hw02-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.