CSci 4270 and 6270
Computational Vision,
Spring Semester, 2022
Lecture 07 Exercise
Due: Thursday, February 10, 2022 at 5 pm EST

I'm giving you a full week here because of the overlap with the due date for HW 2. There **will not** be a new exercise assigned associated with Lecture 8 on Monday February 7.

Problem

As discussed in class, the first few steps of edge detection involve smoothing, differentiation, and calculation of the magnitude and direction. In this exercise, you will implement these — borrowing heavily from the Jupyter Notebook set up for class — and then threshold and visualize the result. In particular, given an input image, which has been converted to grayscale, you need to implement the following steps:

1. Smooth the image with a Gaussian
2. Compute the partial derivatives in the x and y directions (call these I_x and I_y). Use whatever differentiation convolution kernels you'd like.
3. Compute the gradient magnitude image I_g
4. Normalize I_x and I_y so that together I_x and I_y form a unit vector at each pixel.
5. Encode I_g, I_x and I_y into an L^*a^*b image, with I_g forming L, I_x forming a and I_y forming b.
6. Convert the resulting L*a*b image to BGR and save the result.

Start from the code provided.

Notes

A few small issues require your attention. They are easily handled once you are aware of them:

- In step 4 it is possible that I_x and I_y are both 0 at a particular pixel, making I_g 0 as well. This means that the normalization step could involve dividing by 0. To address this, add a very small value, such as 0.00001, to the gradient magnitude at each pixel before the division. In computer vision programming you should constantly be alert for issues like this.
- We haven't discussed L*a*b images that much, but the L value is the "brightness" and the a and b values, together represent color. Thus, gradient magnitudes will be bright and the a,b colors will encode their direction.
- In converting to a and a in step 5, note that the values of I_x and I_y will be floats in the range $[-1, 1]$, but the a and b values must be integers in the range $[0, 255]$ (be very careful that they don't go outside this range). Therefore you must scale and shift the values of I_x and I_y when converting to a and b. As examples, -1 should map to 0 and 1 should map to 255.

- To better visualize the gradients, we sometimes threshold small gradients — mapping them to 0 — and scale up larger gradients. We'll do this here, using two parameters g_0 and g_1 and the following steps. First, modify the gradient image by (a) mapping any gradient smaller than g_0 to 0, and (b) mapping any gradient greater than g_1 to g_1. Then multiply the resulting image (all pixels) by $255/g_1$ and round the result. This should produce the final values that are used as L. If you struggle with the details of this step, refer back the Example 5 from Lecture 2.

The overall command line will now look like

```
python sol.py in_img out_img sigma g0 g1
```

You should need to borrow about 10 lines of code from previous problems and write 10-12 new lines using methods you already know.

No text output is required for this problem. We will judge your results visually — and they don't have to be perfect.

Finally, you should probably add an explicit step at the end – after converting to BGR — to map any pixel whose L is 0 (original gradient was less than or equal to g_0) to 0. This is because having an L value of 0 does not force the resulting BGR values to 0, and I found some small ghosting in some results without this step. To help you out, here is one simple way to do this

```
im_bgr[im_lab[:,:,0] == 0] = np.array([0, 0, 0])
```