# CSE 559A: Fall 2018

Problem Set 1

Due: Sep 25, 2018. 11:59 PM

## Instructions

**Please read the late submission and collaboration policy on the course website:**
http://www.cse.wustl.edu/~ayan/courses/cse559a/

Install Anaconda for Python 3.6 from https://www.anaconda.com/download. We will test all code on this distribution. You can install it locally in a separate directory without interfering with your system install of Python.

1. Complete the code files in `code/` by filling out the required functions.
2. Run each program to generated output images in `code/outputs` directory.
3. Create a PDF report in `solution.pdf` with LaTeXby editing `solution.tex`. In particular, make sure you fill out your name/wustl key (top of the .tex file) to populate the page headers. Also fill out the final information section describing how long the problem set took you, who you collaborated / discussed the problem set with, as well as any online resources you used.
4. The main body of the report should contain responses to any math questions, and also include results and figures for the programming questions as asked for. These figures will often correspond to images generated by your python code in the `code/outputs/` directory.
5. Once you are done, "git add" the completed `solution.pdf` and your updated code files in `code/*.py`. Please do not add the generated output images, as these are already in your report (the git repo is setup to ignore those files). Then do a "git commit", and a "git push". Then, do a "git pull", and a "git log" to verify the timestamp of your submission and the files included. These instructions are also explained in the "problem-sets" section of the course website.

As a general guideline for all problem sets: Write efficient code. While most of the points are for writing code that is correct, some points are allocated to efficiency. Above all, try to minimize the total number of multiplies / adds. For the same number of underlying operations, try to keep the use of `for` loops to a minimum (i.e., over a minimum number of indices). Instead, use convolution, element-wise operations over large arrays, calls to matrix multiply, etc.

## PROBLEM 1 **(Total: 10 points)**

(a) Recall from Lecture 2 that the observed image—ignoring clipping and rounding—is given by:

$$I = gI^0 + \epsilon,$$

where $I^0$ the ideal noise-free image, $g$ is the amplification factor, and $\epsilon$ is zero-mean Gaussian noise that captures contributions from shot noise and (pre- and post-amplification) additive noise. What is the variance of $\epsilon$, in terms of the amplification factor $g$, ideal intensity $I^0$, and noise parameters $\sigma_{2a}^2$ and $\sigma_{2b}^2$. (You can refer to the slides). **(3 points)**

(b) Let's say we already have $g$ as the optimal value of amplification (for clipping/rounding) for ideal intensity $I^0$ which corresponds to an exposure time of $T$. Assuming the scene is static, $I^0$ scales linearly with $T$. So if we chose to have a shorter exposure time $T/k$, the corresponding ideal intensity would also be $I^0/k$. And let us say in that case, we would scale up the amplification factor to $g \times k$ to compensate so that the noise-free version of $I$ would be $gI^0$. But in this case, what is the expected variance of noise $\epsilon$ now ? **(3 points)**

(c) Now, let's say we took $k$ such images $I_1, I_2, \ldots I_k$ (assuming $k$ is an integer) with the shorter exposure times $T/k$. And then, simply set $I$ to be the average of these. Notice that the idea value for this average is still $gI^0$ (since the ideal value for each $I_i$ is again $gI^0$). What is the noise variance in this case ? **(3 points)**

(d) If we had a time budget of $T$, what does this say about the noise trade-off between taking a single shot with exposure time $T$, and $k$ shots with exposure time $T/k$ ? Which is preferable ? **(1 point)**

## PROBLEM 2 **(Total: 15 points)**

Implement histogram equalization. Fill out the `histeq` function in `prob2.py` that takes as input 8-bit grayscale images (i.e., single channel images where intensities are integers between 0 and 255), and outputs an equalized image of the same size (with integer intensities between 0 and 255). Run the program (from the `code` directory) as `python prob2.py`. This will load the image `code/inputs/p2_inp.jpg`, run your function on it, and store the result as `code/outputs/prob2.jpg`. Include this output in the report.

**Important:** Do not use any in-built or third-party code for histogram equalization. Implement it yourself. *Hint:* Look at (and use) the `np.unique` function in numpy.

## PROBLEM 3 **(Total: 15 points)**

(a) Fill out the `grads` function in `prob3.py` to return a map of gradient strength / magnitude $H[n]$ and orientation $\Theta[n]$ for an input grayscale image $X[n]$. Use the Sobel x- and y-derivative kernels:

$$D_x = \begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline 2 & 0 & -2 \\ \hline 1 & 0 & -1 \\ \hline \end{array}, D_y = \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}.$$

The support code already imports the 2D convolution function `scipy.signal.convolve2d` as the `conv2` function. Use this for carrying out *same* convolution: the size of your output image should be the same as that of your input image. We recommend using "symmetric" padding. Once you fill out the function and run `prob3.py`, it should generate

`code/outputs/prob3_a.jpg` showing gradient magnitudes $H$ for the image in `code/inputs/p3_inp.jpg`. Include that output image in your report. **(5 points)**

(b) In addition to the gradient magnitude, `prob3.py` produces three edge images `code/outputs/prob3_b_X.jpg`, for X = 0, 1, and 2. These correspond to three threshold values in the code (variables `T0,T1,T2`). Thresholding is done simply by comparing the $H$ image to these thresholds, which returns a boolean image, and casting it to `float32`, which maps `False` to 0 and `True` to 1. You will notice that while a high threshold misses a lot of edges, a low threshold leads to "thick" edge lines.

To mitigate the latter, implement non-maxima suppression (NMS) by filling out the function `nms`, which takes as input a binarized edge image $E[n]$, and the original magnitude and orientation images $H[n]$ and $\Theta[n]$. Your code should return a new edge image $E^+[n]$, where an edge pixel $n$ in $E[n]$ (i.e., with value 1) is set to 0 if corresponding magnitude value in $H[n]$ is not higher than that of its two neighbors in the direction of the maximum gradient (from $\Theta[n]$), i.e., orthogonal to the edge direction. To keep things simple, only consider the horizontal, vertical, and two diagonal directions by rounding off $\Theta[n]$ appropriately. *Hint:* Use the numpy function `np.where` and `np.logical_and` to get a list of oriented edge pixels and do boolean operations on arrays.

Running `prob3.py` will also call `nms` and produce the thinned edge maps as `code/outputs/prob3_b_nmsX.jpg`. Try different values of the three thresholds that you think cover the range of acceptable edge images (after non-maxima suppression), and include the produced edge images (before and after NMS) in the report. **(10 points)**

**Note:** For this question, it doesn't matter what convention you use for $\Theta[n]$ (i.e., $0^\circ$ could mean aligned with $\pm$ x-axis or y-axis, positive values can mean clockwise or counter-clockwise). All that you have to ensure is that the convention that you use is consistent between parts (a) and (b)—i.e., your `nms` function thins along the correct orientation based on the output of your `grads` function.

## PROBLEM 4 **(Total: 15 points)**

Implement Bilateral filtering by filling out the `bfilt` function in `prob4.py`. The function takes in a color image $X$ and parameters $\sigma_S$ and $\sigma_I$ (spatial and intensity standard deviation), and neighborhood size $K$. It should return an output image $Y[n]$ of the same size as $X[n]$ where:

$$Y[n_2] = \sum_{n_1} B[n_1, n_2] X[n_1], \quad B[n_1, n_2] \propto \exp\left( -\frac{|n_1 - n_2|^2}{2\sigma_s^2} - \frac{|X[n_1] - X[n_2]|^2}{2\sigma_I^2} \right),$$

where $B[n_1, n_2]$ is normalized such that $\sum_{n_1} B[n_1, n_2] = 1$. The support of the filter should be $(2K+1) \times (2K+1)$, i.e., the summation over $n_1$ should be from $n_2 - [K, K]$ to $n_2 + [K, K]$. For locations outside the image boundary, you should set the contribution weights $B$ to 0 (and normalize accordingly). Also note that the weights $B$ are scalar—both $\|n_1 - n_2\|^2$ and $\|X[n_1] - X[n_2]\|^2$ imply summing the squared differences across $(x, y)$ co-ordinates and the three color channels respectively. However, the same weight is applied while averaging for all three color channels.

`prob4.py` calls the `bfilt` function with different parameters on two noisy images `code/inputs/p4_nz*.jpg` to create a number of output files `code/outputs/prob4_*.jpg` (most of them with the first image), with `code/outputs/prob4_*rep.jpg` corresponding to repeated bilateral filtering. Include these generated images in your report. (You may optionally also want to try different parameters to see if you can get better results.)

## PROBLEM 5 **(Total: 15 points)**

(a) Let $F[u, v]$ be the Fourier transform of a real-valued image $X[n_x, n_y]$ of width $W_X$, height $H_X$. Treating the imaginary and real parts of a complex number as two separate scalars, show you only need to store $W_X H_X$ distinct scalars from $F[u, v]$. Explain which scalars you will be storing, and how you can recover the full $F[u, v]$ from them. You may need to do this separately for when both $W_X, H_X$ are even, when both are odd, and when one is odd and one even. **(10 points)**

(b) `prob5.py` tries to implement convolution (same convolution with circular padding) in the Fourier domain. Fill out the `kernpad` function that takes a smaller kernel $k$ and the size of the image $X[n]$, and returns a zero-padded and circularly-rotated kernel of the same size as the image, so that the center of the original kernel is at the top-left corner. You can assume that both the height and width of $k$ are odd. Running the script will create a file `code/outputs/prob5.jpg` that contains three columns: original image, output from calling regular convolution, and output from doing it in the Fourier domain with `kernpad`. The second and third column should look identical. Include this image in your report. **(5 points)**

## PROBLEM 6 **(Total: 30 points)**

(a) Implement Harr wavelet decomposition by completing the `im2wv` function in `prob6.py`. The function takes a grayscale image and number of levels $N$ as input. The output is a python list with $N + 1$ elements: `[V1,V2,...,VN, X(N+1)]`. All elements of this list, except for the last one, are lists themselves—each containing the three detail images $[H_1, H_2, H_3]$—for the corresponding level, with the first element `V1` being the finest level (for which the three images have width and height half of that of the input image). The last element of the list returned by the function will be an image, corresponding to the coarse "scaling" coefficients at the top of the wavelet pyramid. `prob6.py` contains code to visualize the pyramid and store the result as `code/outputs/prob6a_*.jpg`. Include these in the report. **(15 points)**

(b) Complete the `wv2im` function in `prob6.py`. This should take the pyramid list produced by `wv2im` and reconstruct the original image. You should infer the number of levels from the length of the list. Running `prob6.py` also calls this reconstruction function, on the original pyramid and on versions with the finest levels zeroed out. These are all saved as `code/outputs/prob6b_*.jpg`. Include these outputs in your report. **(15 points)**

**Important:** Do not use any wavelet toolbox / libraries for this problem.