Prof. Raghuveer Parthasarathy
University of Oregon; Fall 2024

Physics 410/510 Image Analysis: Homework 3 – *partial*

NOTE: I will likely add one more exercise to this, by Friday.

Note: The first few exercises are similar in their setup, so they're not as long, in total, as they seem.

Due date: Wednesday October 16 by 11:59 pm, submitted through Canvas. You'll see in "Assignments" a place to submit a **PDF**.

Reading: (Optional) Gonzalez, Woods, and Eddins, Sections 3.4 and 3.5 – about spatial filtering and convolution. The same things we discussed in class, with more elaboration.

1 Convolution and filtering. (4 pts.) Find an image and make it grayscale (2D) if it isn't already using whatever means you like. Also make an 11×11 square array in which all the elements are the same. Perform a local averaging of the image by convolving it with the 11×11 array. In Python use `convolve` in the `scipy.ndimage` package. In MATLAB use `imfilter`; this is similar to `conv2` but allows clearer statements about what to do at the borders. (There are other subtle differences between the two.) Try at least two different choices for what the convolution does near and beyond the image borders, one of which is zero-padding. Submit a "Zoomed in" image for each choice, and comment on whether they are doing what you expect.

Suggestion: Pick an image that isn't too large (maybe 600x600?), so that you can see the effects of the filtering more clearly. **Or** you can make a larger kernel – not 11×11, but something like 51×51, but be warned that this may be slow.
Comment: The "Zoomed in" image should zoom in to the edge, of course, so you can see the difference between edge options.
Comment: Recall that "zero padding" means filling all the "new" pixels beyond the edge with zeros.

2 Gaussian filtering. (6 pts.) Find a grayscale image or make a color image grayscale however you like; choose an image that has some fairly sharp edges. We'll perform Gaussian filtering of this image. Note that a Gaussian function is

$$A \exp\left(-\frac{r^2}{2\,\sigma^2}\right)$$

where r is the distance to the center.

(a) [3 pts.] Gaussian-filter the image by explicitly creating a Gaussian kernel and convolving it with your image as in Problem 1. Show your code for making the kernel. (*Hint:* use meshgrid for x and y positions, then figure out "r".) Show the original image, an image of your kernel and the output of the convolution for two different values of the Gaussian width, σ.

(b) [1 pt.] Pick one of the σ values and Gaussian-filter the image using a built-in Gaussian filter from a MATLAB or Python toolbox, for example `imgaussfilt` in MATLAB and `skimage.filters.gaussian` in Python. Show this image.

(c) [2 pts.] Calculate the difference between the output arrays (for the same σ) in (a) and (b). Are they identical? First, pay attention to the range of the arrays: if you started with an 8-bit image, is the *output* array 0-255?! (If not, multiply by the appropriate factor, then subtract.) Show this difference image. Where does it differ? Think about why, and explain.

(d) [1 pt] Time how long each method (a and b) takes. (Use the `time` library in Python, for example, or `tic` and `toc` in MATLAB.) Why might these be so different? Read the beginning of this, https://bartwronski.com/2020/02/03/separate-your-filters-svd-and-low-rank-approximation-of-image-filters/, through "Simplest analytical filters," and briefly explain. This isn't the only reason for the speed difference, but it is the dominant one.

3 Median filtering. (5 pts.) Perform a local median filtering of the same image you used in the previous exercise. Apply a *N*x*N* median filter, ,where *N* is the same as one of the σ values of one of the Gaussian filters from the previous exercise. (Python: `skimage.filters.median`, which is very slow; MATLAB: `medfilt2`). Compare Gaussian and median filtering; zoom into the neighborhood of a sharp edge in your original image. Comment, and submit the median-filtered image, and a zoomed in region (a screenshot is fine). Also note how long the median filtering takes.

Comment: Since `scikitimage's` documentation about the median filter is not very good, I'll provide some elaboration:
The first input to `skimage.filters.median` is the image to filter.
The second input is a 2D array whose size is of the size of the neighborhood that you want to evaluate the median over. The elements of this array are "1" for every location that contributes to the median and "0" at every location that will be ignored.
So, for example, to apply a 3x3 median filter, you'd make an array

```
1 1 1
1 1 1
1 1 1
```

and input this as the second argument to `skimage.filters.median` . (You can make the array of ones with `numpy.ones`, e.g. `numpy.ones((3,3))`.
You may be wondering: why would I ever have a neighborhood array that's not all ones? This array:

```
0 0 1 0 0
0 1 1 1 0
1 1 1 1 1
0 1 1 1 0
0 0 1 0 0
```

would median filter in a diamond-shaped neighborhood around each image pixel.

4 Filtering and thresholding. (4 pts.) Apply (separately) a Gaussian and a median filter to the photo "MakeUp_RichardPrince_1983_gray.png" (on Canvas), and then threshold the resulting images using Otsu's method. Use a similar median filter size as the Gaussian width. Comment on the appearance of the thresholded images.

5 High-pass filtering. (5 pts.) As noted in class, a Gaussian filter is a "low pass" filter, retaining gently-varying (low spatial frequency) changes in intensity and smoothing close-together changes (high spatial frequency). Any image is a combination of low- and high-frequency components. (We can be more exact about this using the Fourier analysis; we may revisit this later.) Therefore, if we *subtract* a Gaussian-filtered image from the original image, we're left with the high-frequency components. This subtraction (the "identity filter" minus a Gaussian filter) is a high-pass filter. Download "Elevator_to_the_gallows.png" and create a high-pass-filtered image with Gaussian width 21. Submit the image and comment on its appearance. Note the intensity range: if you're subtracting double-precision numbers, you may wish to set negative values equal to zero.

Caution: Pay attention to the data types and to the range of values in your original and the Gaussian filtered images!

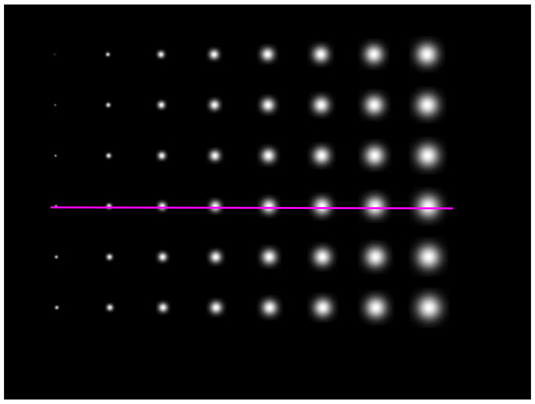
Note: As a result of the subtraction, you'll end up with an array that has negative values. Add some number (what?) to the array so that its minimum is 0 and save it as a regular 8-bit image file. (*Optional:* Scale the max to be 255. Or scale some other value to be 255.)

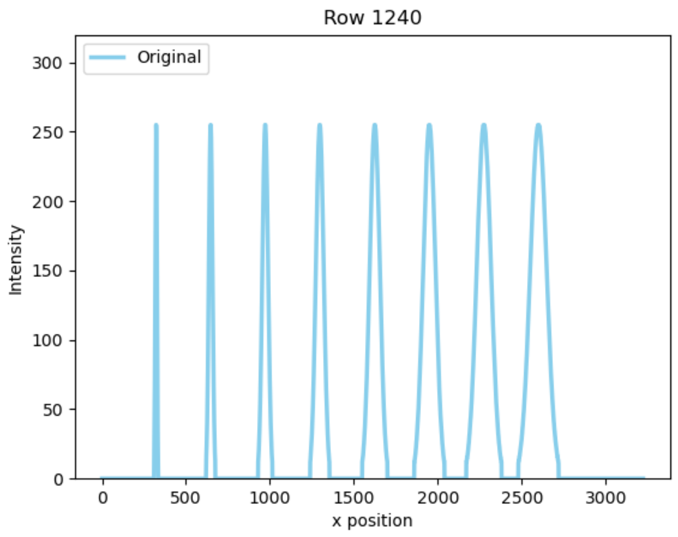
By the way: The image is a still from the excellent 1958 film "Elevator to the Gallows." It's available free to stream through UO's subscription to the Kanopy service: https://uoregon.kanopy.com/video/elevator-gallows ; see more at https://uoregon.kanopy.com/

6 Band-pass filtering. (7 pts.) We've now seen low-pass and high-pass filtering. We can combine these to make a band-pass filter which, as you might guess, keeps middle-frequency components of an image. Load the image "gaussians_s2_to_s50_px.tif," which is a grid of Gaussians with varying width, from about σ = 2 to 50 pixels. Figure out a series of low or high pass filtering and subtraction steps that will diminish the intensity of the "big" and the "small" dots, leaving the "medium" dots relatively intact. ("*Relatively*" is important: you won't perfectly reject or keep dots, but the middle columns should be attenuated less than the outer ones.) Describe your procedure in words, include your code. Submit an image of the final band-pass filtered image. Also submit a plot of intensity vs. column number (i.e. "x") for row 1240, as indicated below, including both the original and the band-pass-filtered images.

Note: As a result of the subtraction, you'll end up with an array that has negative values. Add some number (what?) to the array so that its minimum is 0, and save it as a regular 8-bit image file. (Optional: scale the max to be 255.)

Plot the intensity of all the pixels in Row 1240, approximately where this magenta line is...





7 Signal to Noise Ratio (14 pts). The first part of this problem is a useful first step towards next week's simulated single molecule images, and the second part revisits the notion of the signal-to-noise ratio. Suppose your background (photons plus dark noise) is on average ***b*** = 2 intensity units per millisecond at each pixel, and readout noise is negligible. The signal you care about gives, on average, ***r*** intensity units per millisecond, all of which go to the center pixel of a 27 x 27 pixel sensor array. The signal is also variable and is also Poisson distributed; in other words, if ***r*** = 4, we expect in *t* = 2 ms an average intensity of ***r t*** = 8, but the number is actually a random number drawn from a Poisson distribution:

$$Probability(intensity = k) = \frac{(rt)^k e^{-rt}}{k!}$$

This signal intensity adds onto the background intensity. Write a function that creates a simulated image as a function of ***r***. Note that you'll have to generate Poisson-distributed random numbers. In MATLAB, use `poissrnd`; in Python `numpy.random.poisson` .

(a) (4 pts.) Show one example each of the output for r = 2, 6, and 10 units/ms and an exposure time of 1 millisecond. (You can show them each scaled to the full range of display intensity, which is the default for `matplotlib.imshow` in Python or keep the same range for all by setting `vmin` and `vmax`.) .

(b) (10 pts.) Make a series of simulated images for ***r*** = 0.5 units/ms for a series of exposure times ***t*** = 1, 2, 3, ..., 300 milliseconds. Just make one simulated image for each t; we'll deal with averaging over images later. From the actual pixel values of the simulated images, calculate the signal-to-noise ratio (SNR). You can crudely use the intensity of the center pixel as an estimate of "signal" and the standard deviation of all the other pixels as your estimate of "noise"" Make a scatter plot of SNR vs t, and comment on whether it has the expected shape. (*Hint:* you might want to make your image a 1D array to more easily calculate the standard deviation of "all the other pixels." If you had a 3x3 image, you'd calculate the standard deviation of pixels 0, 1, 2, 3, 5, 6, 7, 8; why?)

NOTE: We'll discuss this problem on Thursday. If you have trouble with it, submit a description of your approach and what did/didn't work, and I'll (probably) allow you to revise this in Homework 4 for full credit.

8 Probably a new exercise.

9 Comments (2 pts.) Roughly how long did this assignment, not including the readings, take? Which parts took the most time?