PHYS 410/510 Fall 2024 Homework 2
Bret Brouse Jr. (10/09)

Problem 1 (Thresholding). *(6 pts.) Use Otsu's method (`threshold_otsu` in Python's `skimage` library; `greythresh` in MATLAB) to determine the threshold values for the images "robbert-mapplethrope-calla-lily-1984.png" and "istanbul_arch_museum_gray_crop.png." Plot the histogram of pixels for each image and indicate the threshold value on the histogram plot. Submit the histograms along with the binary thresholded images - paste those into your PDF. Comment on all results. Similarly, threshold an inset to the Calla Lily photo: "robert-mapplethrope-calla-lily-1984_CROP.png" - why does this look different than the thresholded whole image?*

Solution. First the images were imported and threshold values determined using Otsu's method. The histograms for the calla lily and the Istanbul images, as well as their images after thresholding, can be seen in Figures 1-4 below.

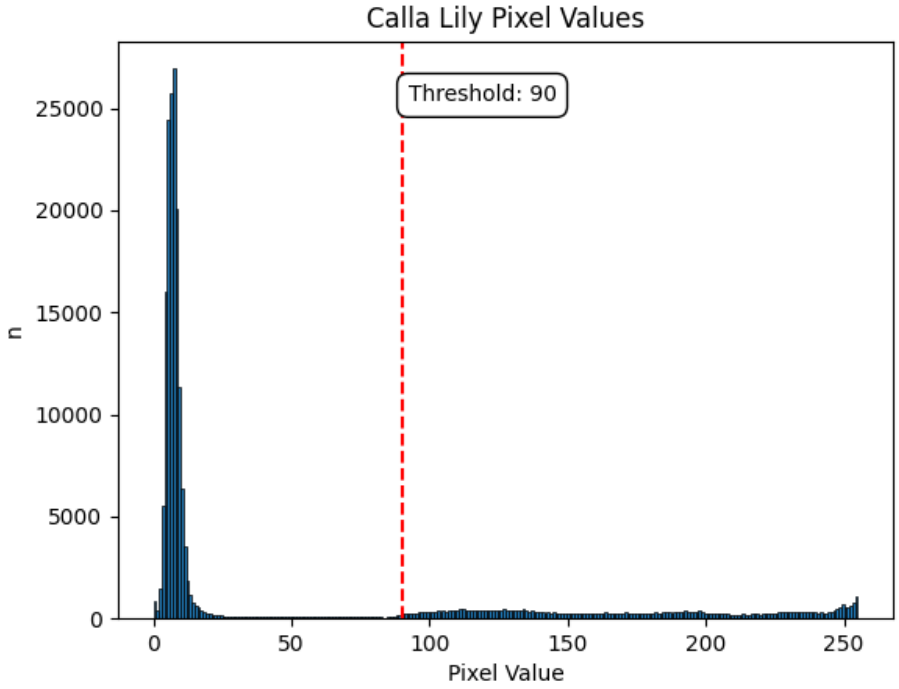

Figure 1: A histogram showing the `uint8` counts of values in the calla lily array. The threshold value using Otsu's method is highlighted.

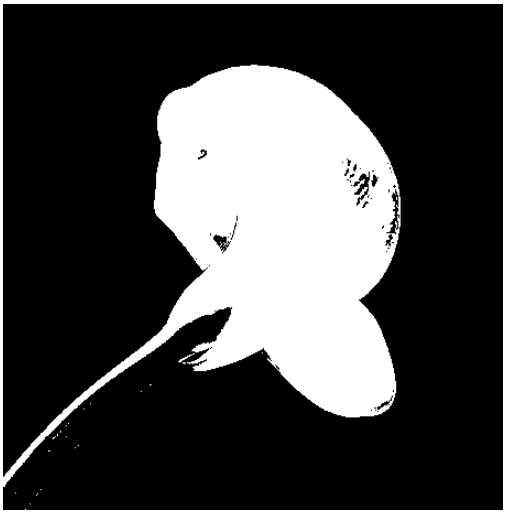

Figure 2: The original Calla Lily image after thresholding to the value 90, determined by Otsu's method.

It's interesting to note that the threshold value for the calla lily nearly completely washes it out while for the sculpture you still maintain a good bit of detail. One would most likely not be able to tell it was a calla lily (it resembles a jelly fish after applying the threshold). The bust is still distinguishable, albeit with the face next to the right jawline looking far more sinister.

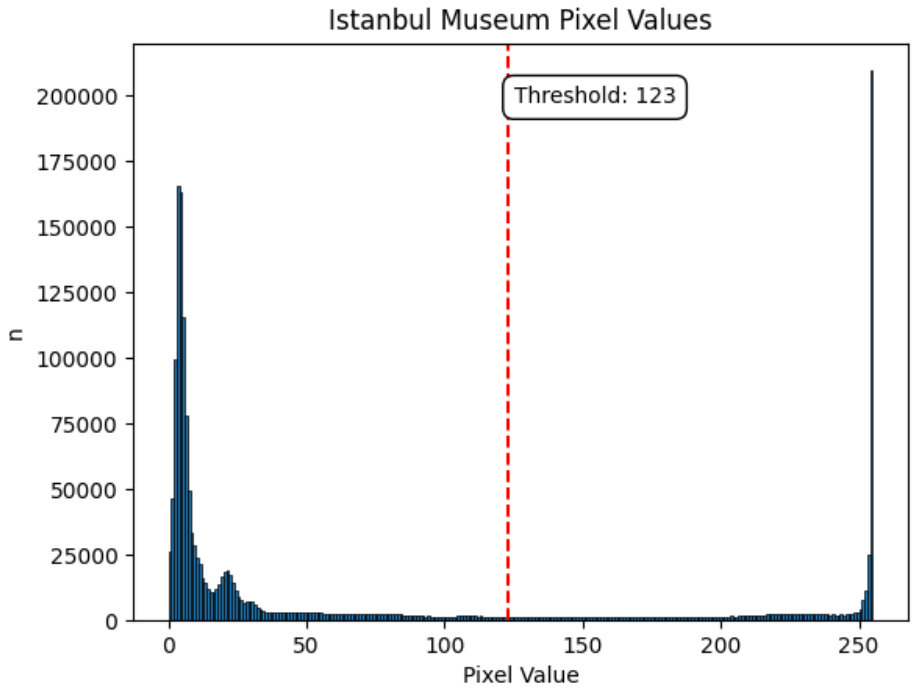

Figure 3: A histogram showing the `uint8` counts of values in the bust array. The threshold value using Otsu's method is highlighted.

Figure 4: The original bust image after thresholding to the value 123, determined by Otsu's method. The smaller face makes me uncomfortable.

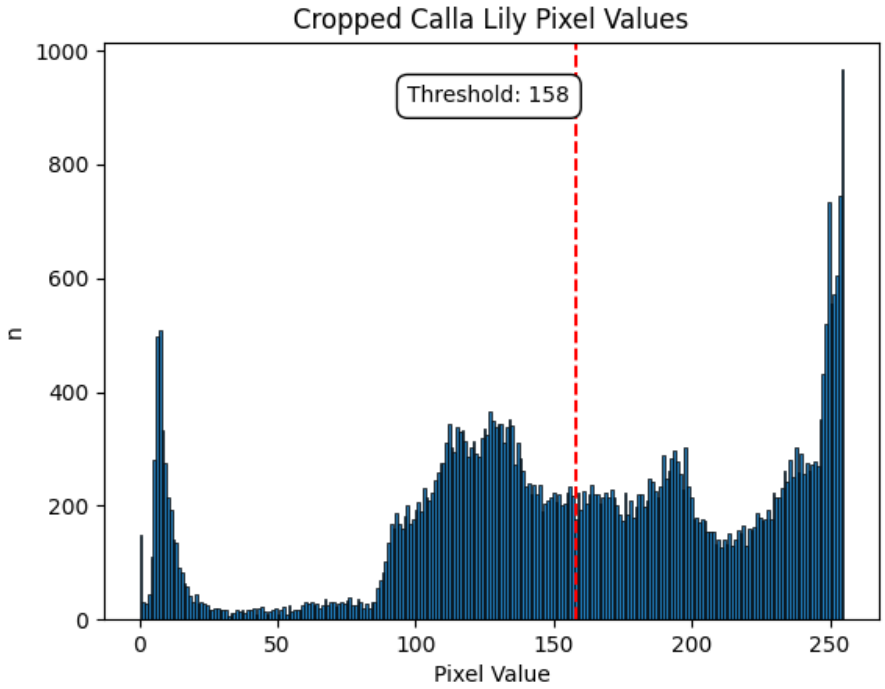

Figure 5: A histogram showing the `uint8` counts of values in the cropped calla lily array. The threshold value using Otsu's method is highlighted.

Figure 6: The cropped calla lily image after thresholding to the value 158, determined by Otsu's method.

We next apply the same technique to the cropped calla lily photo, which has a much different result, as seen in Figures 5 and 6. We find a much different result since the threshold value is much higher. While it would be hard to determine if this was a lily, some of the detail is still preserved. This is due to the fact that the threshold value is higher than before and in the full image, a huge chunk of the values are below it's original threshold of 90. In the cropped image, it seems to be a bit more of an even split, preserving some detail. It goes without saying, but the threshold is different because we're sampling a different image, even though it's a cropped version of the previous one.

Problem 2 (Thresholding, again.). *(6 pts.) Practice thresholding, both global and adaptive (or "local"), as described in the posted excerpt from the Gonzalez et al. book. Find some images; try things. You can use standard toolbox functions, such as* `threshold_local` *in Python's* `skimage` *library (use* `offset=0`*) or* `adaptthresh` *together with* `imbinarize` *in MATLAB or. For one image, submit the original and the results of applying a global threshold and a local threshold, noting what block size you used. Be ready to share your examples them with the class.*

Solution. For my image, I chose one of my favorite paintings, 'I and the Village' by Marc Chagall. It's an interesting painting with a lot of color that one doesn't normally see for things, so I thought it would be fun to threshold to interesting results. Seen in Figure 7, I performed the adaptive thresholding with a block size of 37. I actually tried a few block sizes to see how it works, noticing that if you choose the size (or bigger) than the entire image, you just get an Otsu threshold again. This implies that for adaptive, it's taking small regions based on block size and thresholding those individually. The result is an interesting take on a black and white version of the image that preserves a lot of detail.

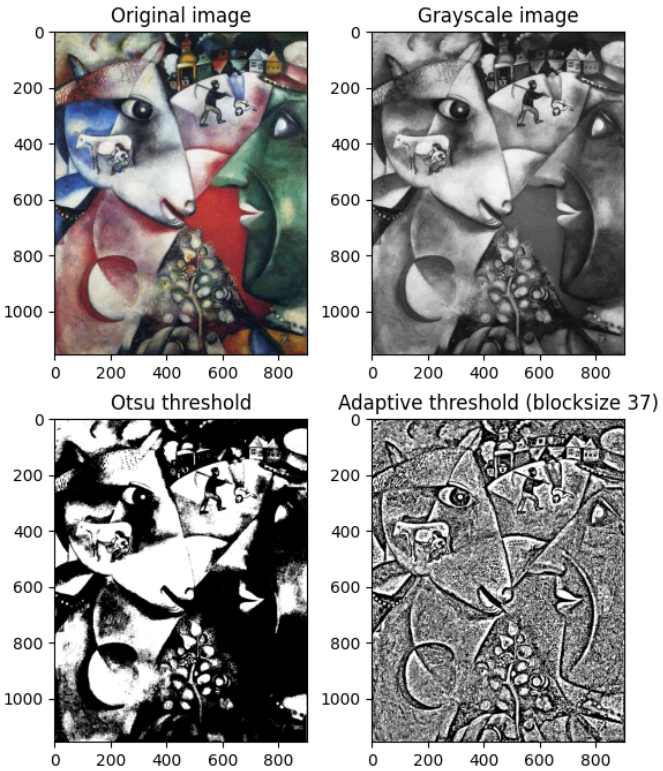

Figure 7: I and the Village by Marc Chagall including how it looks when applying an Otsu threshold and an adaptive threshold.

Problem 3 (Protein Density). *(12 pts.) Download the image "microarray_crop.png", which shows just a few fields of peptides, roughly rectangular in shape, out of over 200,000 patterned onto a chip. (Source: this paper[2].) The different intensity levels correspond to different densities of proteins bound to these peptide patches, indicating different binding strengths. Intensity is proportional to protein density. Use ImageJ (drawing boxes and selecting "Analyze Measure") or figure out how to draw/analyze regions in Python or MATLAB to measure the average intensity in different regions.*
(a) (8 pts.) The protein density in field C is ____ X that of field D. Fill in the blank and describe your process. Pay attention to the black regions! Note: you needn't calculate uncertainties.
(b) (4 pts.) What can you say about the protein density in field A relative to D?

Solution. I used ImageJ to verify these but I wrote code to calculate these average intensities myself. I manually read the grid to determine the squares I wanted to use, rather than using edge detection (for the sake of time). The average intensities for both Python

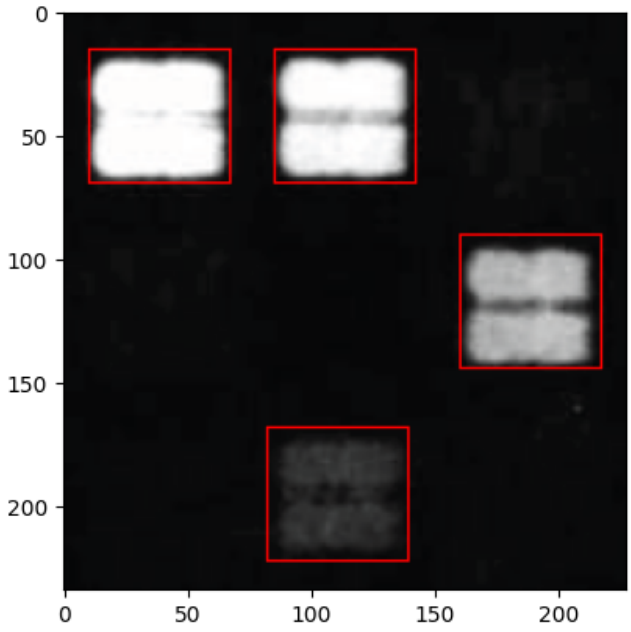

Figure 8: The original protein density image but with region squares overlaying it to show where we'll measure intensity.

and ImageJ can be found in the following table:

Region	Python	ImageJ
A	196.52	208.79
B	178.24	198.43
C	121.51	138.19
D	38.23	35.55

The ImageJ box sizes are definitely a little less consistent, but we see that they're comparable. The results of the ratio for the Python analysis end up being that $\frac{C}{D} \approx 3.178$ and $\frac{A}{D} \approx 5.140$. My method was to block off a region of pixels in the image and average their pixel values based on the squares that I defined before by eyeballing them. This could be done programmatically by edge detection but this is time consuming and I feel outside the scope of the question at hand.

Problem 4. *Decided not to copy paste this because I didn't want to format things anymore!*

Solution. (a) Plotting as a surface pointing out the spike in Figure 9:

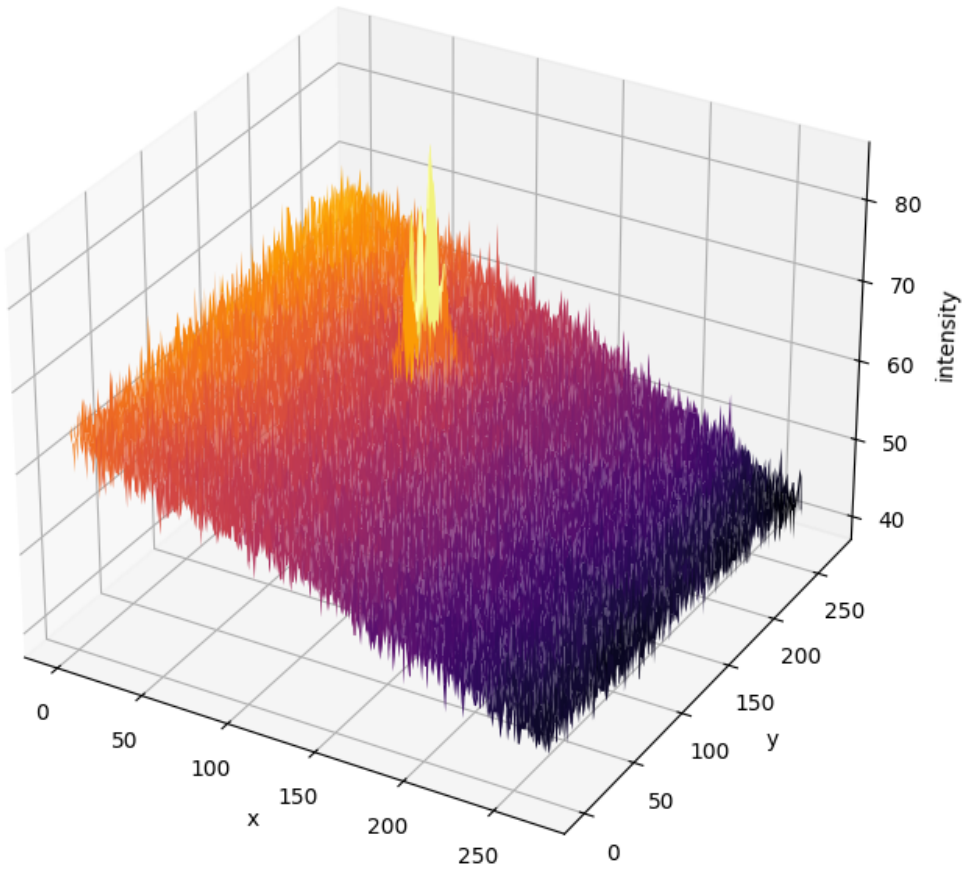

Figure 9: A 3D representation showing the height of a blip in the image.

(b) Now we show both graphs of intensity vs x and y values respectively in Figure 10.

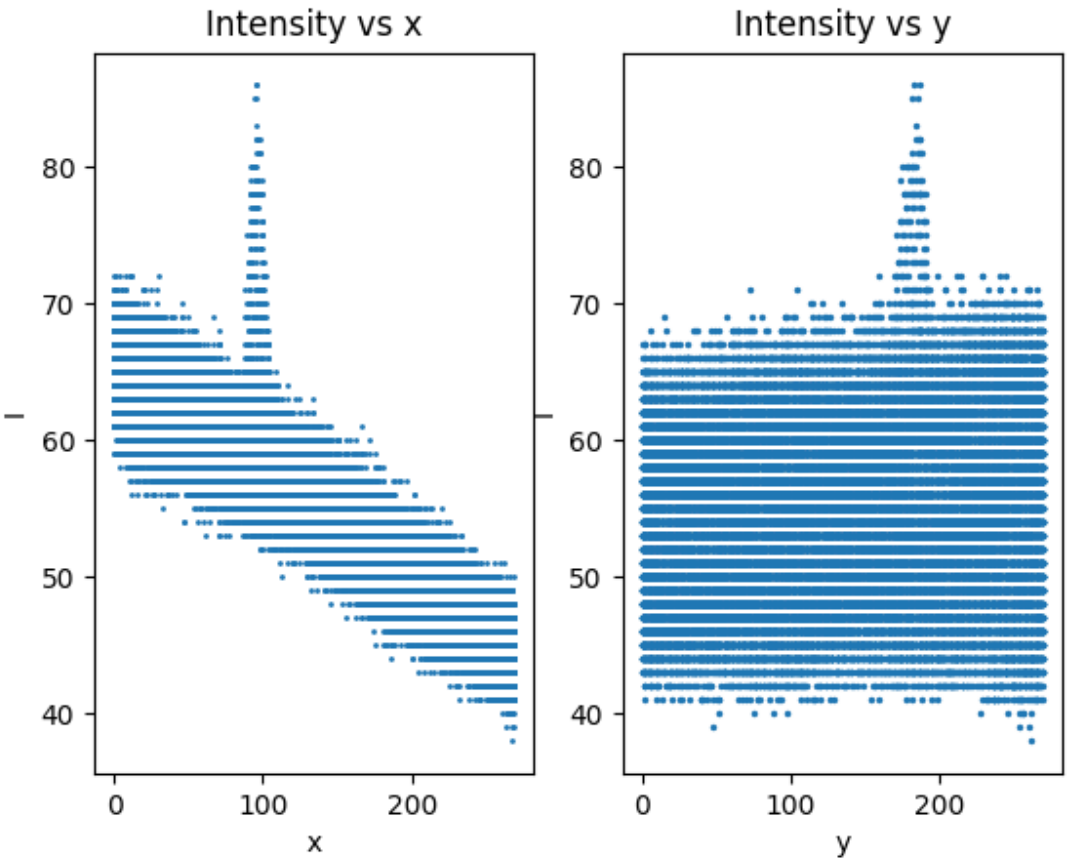

Figure 10: Intensity plots vs x and y values.

(c) Next we use a quadratic fit to subtract some background noise from the image. First I'll display the original and the image with a subtracted background, then I'll paste the code. The coeefficients of the polynomial fit are given at the bottom of the page. Figure 12 shows the code used to accomplish this.

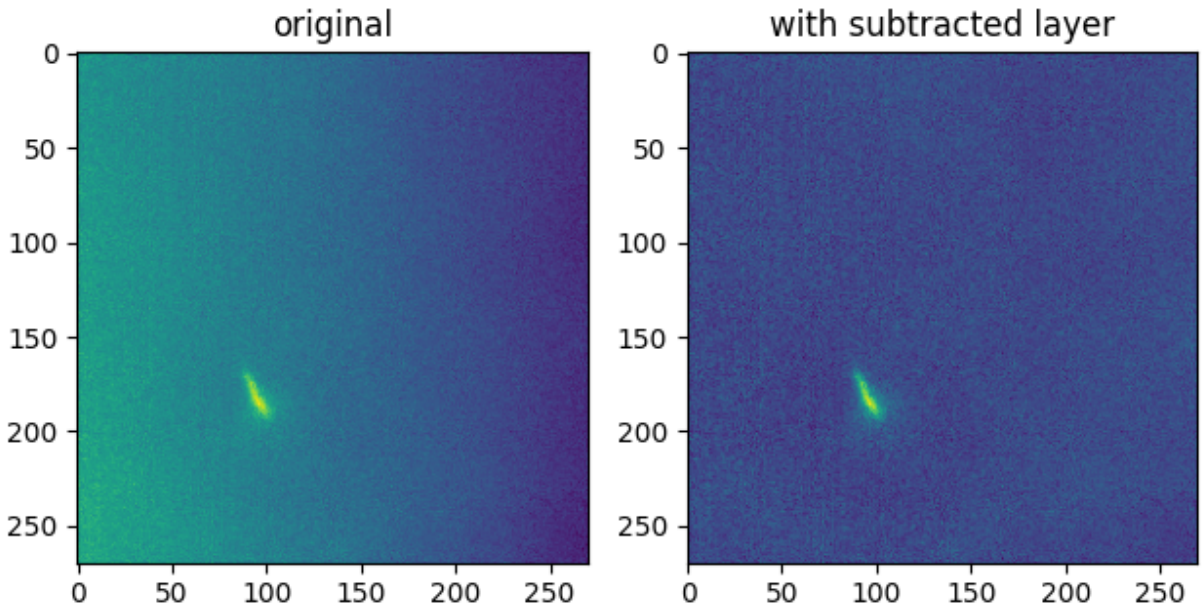

Figure 11: Original bead image and the same image with after substracting backround noise using a polynomial shape fit filter.

(d) The biggest issue is that a subtractive layer 'assumes' a mostly gradient amount of background noise. This could be common depending on image capturing set up, but if there are other anomalies in the image (peaks), then this would fall apart. This is because when you apply a polynomial fit over the image, it's considering the whole image. You could fix this by doing some sort of 'adaptive' subtraction I think, similar to the adaptive thresholding. Another issue with the current setup is that they're looking for peaks which might be fine for 1-3, if they're distinct, but otherwise this could very much be very hard to work with. This could become messy very quickly. Instead, we get a nice 'deletion' of background noise from the average while preserving the total background noise. I think I saw this in my comp sci class as well. If you process this as audio, it's neat to hear as a blip.

P_5	P_4	P_3	P_2	P_1	P_0
-4.43e-5	-5.49e-5	3.23e-6	-5.75-2	1.50e-2	6.27e1

Lastly, here's my code snippet. There's a lot that went into it, but this is how I did the regression.

```
# (c)
# For a quadratic fit, we need to define a matrix to do the least squares fit
x_sq = flat_x ** 2
y_sq = flat_y ** 2
xy = flat_x * flat_y

# assemble that matrix
mat = np.column_stack([x_sq, xy, y_sq, flat_x, flat_y, np.ones(flat_x.shape)])

# Use least squares to find the best fit surface
coeff = np.linalg.lstsq(mat, flat_i, rcond=None)[0]

# make the fit surface
fit_surf = (coeff[0] * x**2) + (coeff[1] * x*y) + (coeff[2] * y**2) + (coeff[3] * x) + (coeff[4] * y) + coeff[5]

# subtract the new surface
nobg_bead = bead_2d - fit_surf

# plot that!
plt.figure()

# Plot original
plt.subplot(1, 2, 1)
plt.imshow(bead_2d)
plt.title('original')

# Plot image with subtracted layer
plt.subplot(1, 2, 2)
plt.imshow(nobg_bead)
plt.title('with subtracted layer')

plt.tight_layout()
plt.savefig('images/homework_2/xyi_plot.png')
plt.show()
print(coeff)
```

Figure 12: Code used to do the background subtraction.

Problem 5 (Comments).

Solution. The problems themselves took 4-5 hours, but I'm more fluent in Julia rather than Python so some of it was just syntax re-familiarization. I also have been writing my solutions in LaTeX which take some time but I don't consider that part of my homework, just a preference. Overall, if I had to judge the homework time this actually took up, I'd say this was roughly 5-5.5 hours to be worthy of submission.