PHYS 410/510 Fall 2024 Homework 3
Bret Brouse Jr. (10/16)

Problem 1 (Convolution and Filtering).

Solution. I chose a photo of Crater Lake that I took a few years ago at sunrise (Figures 1 and 2). Since we're concerned with edge behavior, I thought this would be a good way to check since there's a significant difference between the right and left edges. I chose to

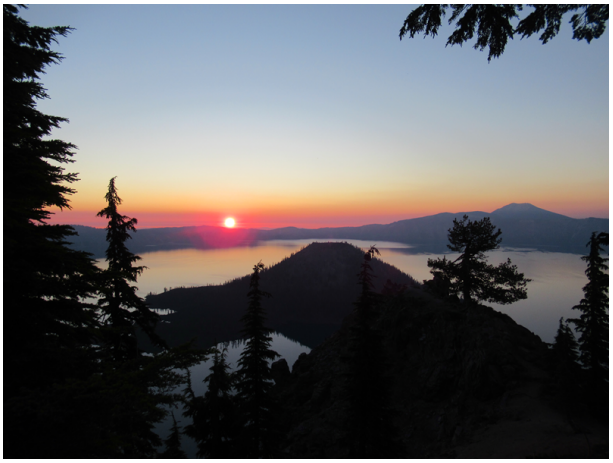

Figure 1: Crater Lake at sunrise...

Figure 2: ...in grayscale.

convolve this image with 4 different methods including zeroes, mirror, wrap, and nearest (Figure 3). When we zoom in on the right hand side (Figure 4), nearly centered vertically (pixel ranges [275:325, 750:800]), we can see that for the zeroes convolution we get a darkened edge due to the fact that 0 here would register as a black pixel and so we have a gradient as more and more zeroes are incorporated into the kernel as we move to the right. We see the same effect on the wrap convolution because the pixels on the left hand side of the image are very dark. I manually cut a section from the zeroes image and compared it to that of the wrap and we see what I was expecting: Since the pixels on the left hand side are not exactly zero, but slightly above it, the edge on the zeroes image has a slightly lighter gradient than that of the wrap convoluted image (Figure 5).

As for the nearest and mirror, they look very similar. There may be some small discrepancies but their values are very close which I would also expect as they way they interpolate the missing pixels is similar.

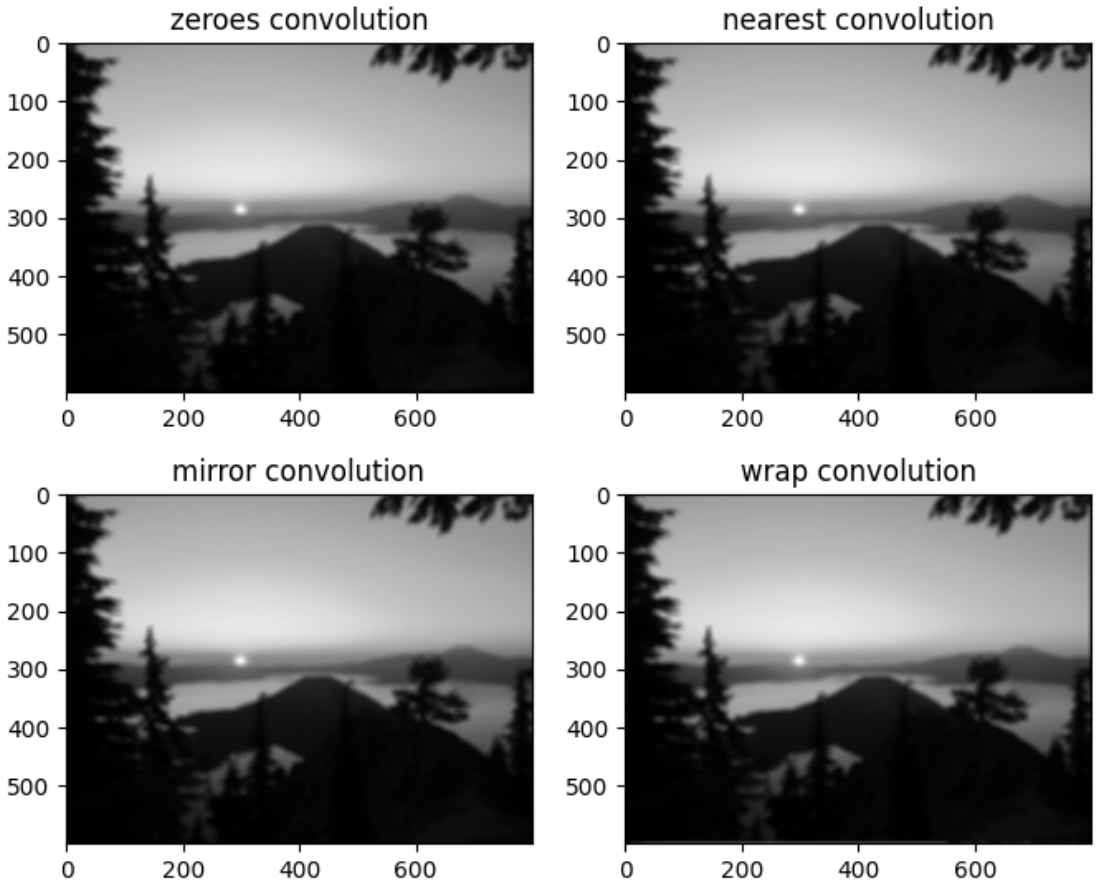

Figure 3: Convoluted images.

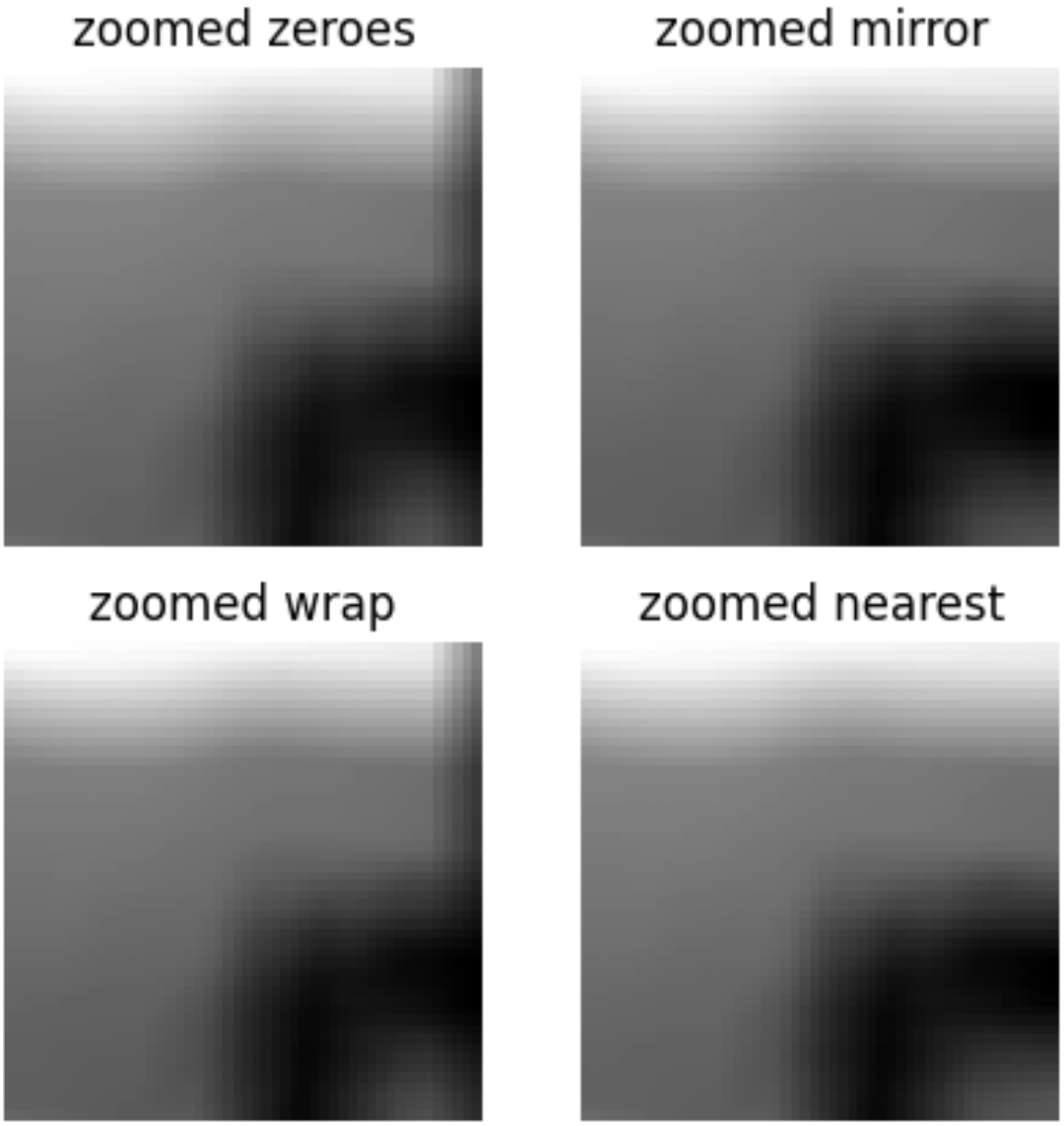

Figure 4: Zoomed sections of convoluted images.

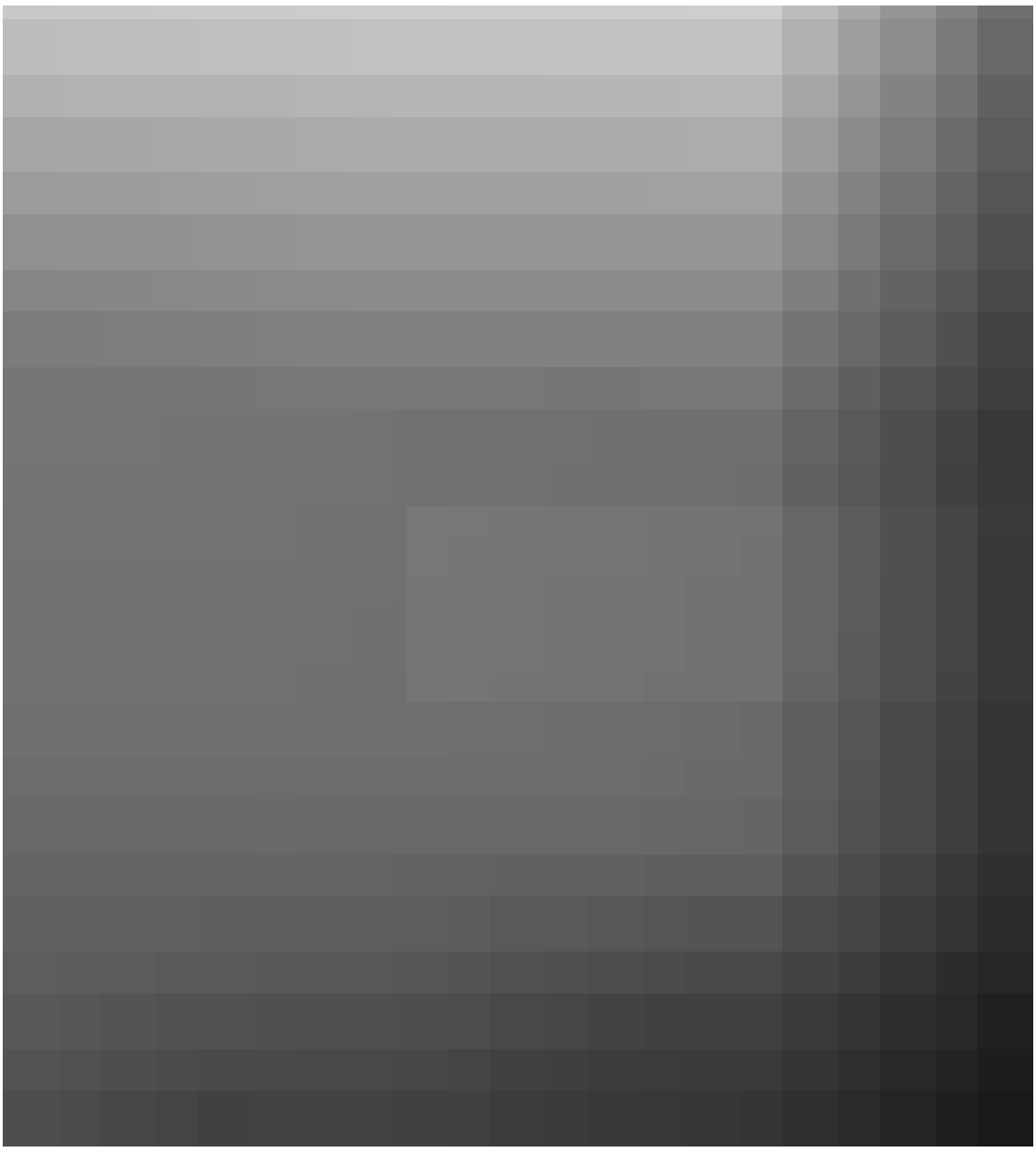

Figure 5: Mega zoomed image showing the difference between the zeroes and wrap convolution.

Problem 2 (Gaussian filtering).

Solution. Fighting with LaTeX a lot lately but I've included the part this applies to in the figure captions.

```python
# Function to make a gauss kernel
# May refactor this to take in any function?
def gauss_kernel(size, sig):
    # Need half size to center gaussian
    half_size = size // 2
    lsp = np.linspace(-half_size, half_size, size)

    # make evenly spaced fellers centered on the origin and grid them
    x, y = lsp, lsp
    x, y = np.meshgrid(x, y)

    # Keep r what it is so the final calculation looks normal
    r = np.sqrt(x**2 + y**2)

    # make the kernel and return normalized version
    kern = np.exp(-r**2 / (2 * sig**2))

    return kern / kern.sum()
```

Figure 6: Gauss kernel function code. (a)

Figure 7: Original image (grayscale) of Three-Fingered Jack (my own photo). (a)

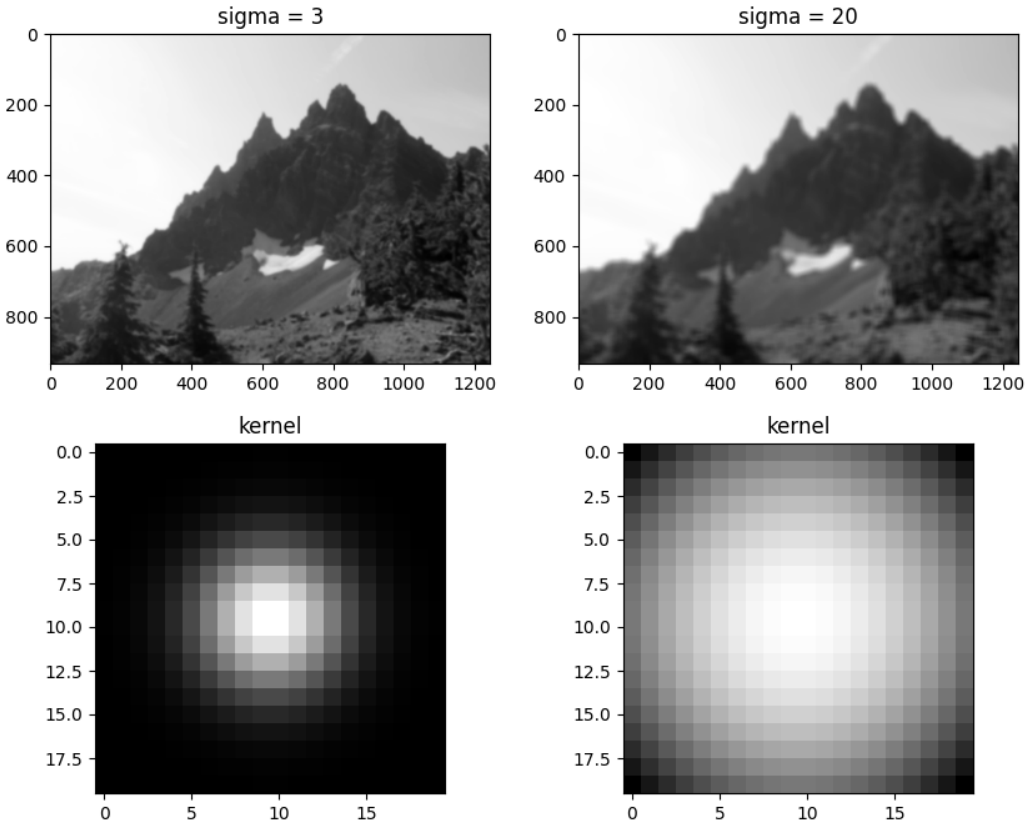

Figure 8: Gaussian convolved images of Three-Fingered Jack with three different σ values. (a)

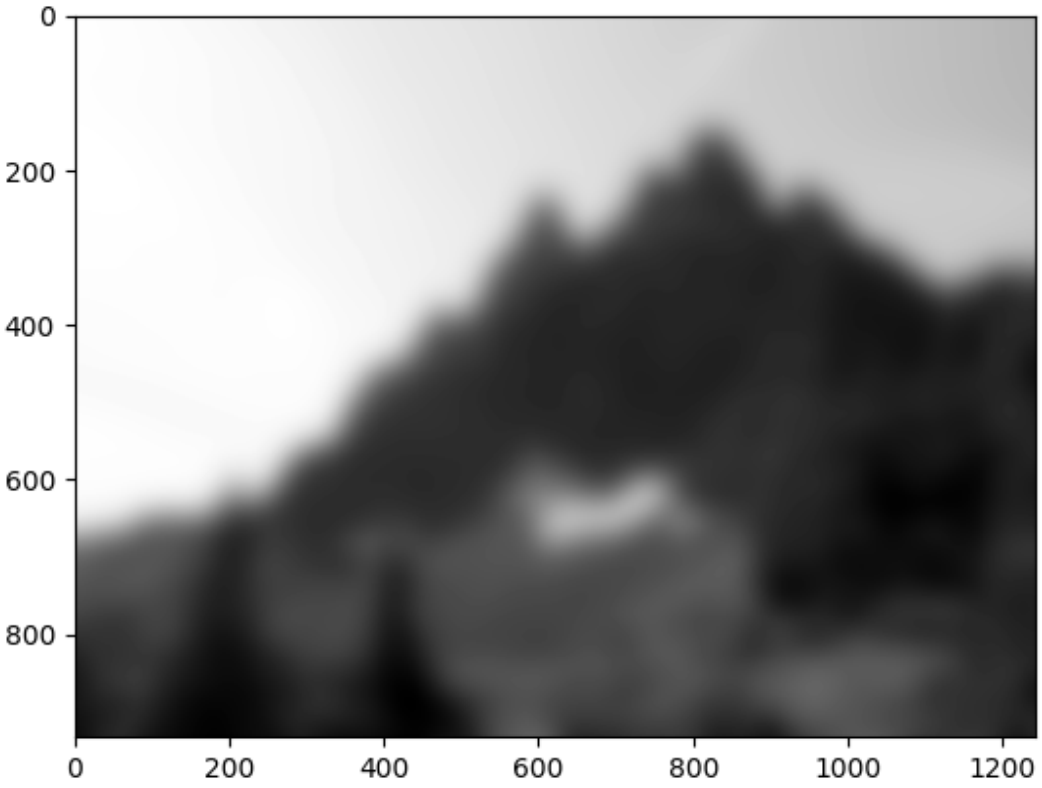

Figure 9: Gaussian convolved image of Three-Fingered Jack using the built-in Gaussian function where $\sigma = 20$. (b)

(c) The output array seems to differ most at edges with varying values. For instance, in the top left of the image, most things are consistent so we're not going to see many changes as it's mostly just high values so they cancel out. However in the bottom right, there's some variation (at the corner) so we see bigger (albeit not large) differences. There must be something going on in the built-in version that accounts for nearby pixels or something whereas how we account for edges is reduced to a few simple choices.

(d) Times:
manual ≈ 0.43576s
builtin ≈ 0.18308s
I did need to change the kernel size to get the expected result. The reason why the builtin function is faster is because it's separable. What they do is break down an M x N matrix of values for the convolution kernel into 1D arrays, such that it is now only passed as an M + N operation. One is applied in the horizontal direction and one in the vertical, usually in that order. This makes the number of calculations O(n) instead of $O(n^2)$.

Problem 3 (Median filtering).

Solution. The gaussian filter tends to do an overall blur whereas the median preserves edges (while still bluring non edges). One can see that in Figure 10, whereas in Figure 11, we can see it in fine detail along with the difference, emphasizing the edges. The median time took 0.153s. Here, $\sigma = 20$.

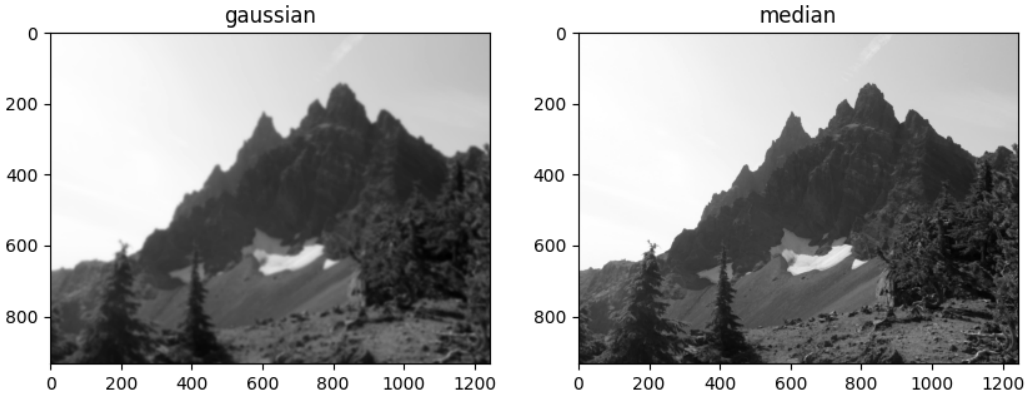

Figure 10: A Gaussian and median filtered version of the photo of Three-fingered Jack.

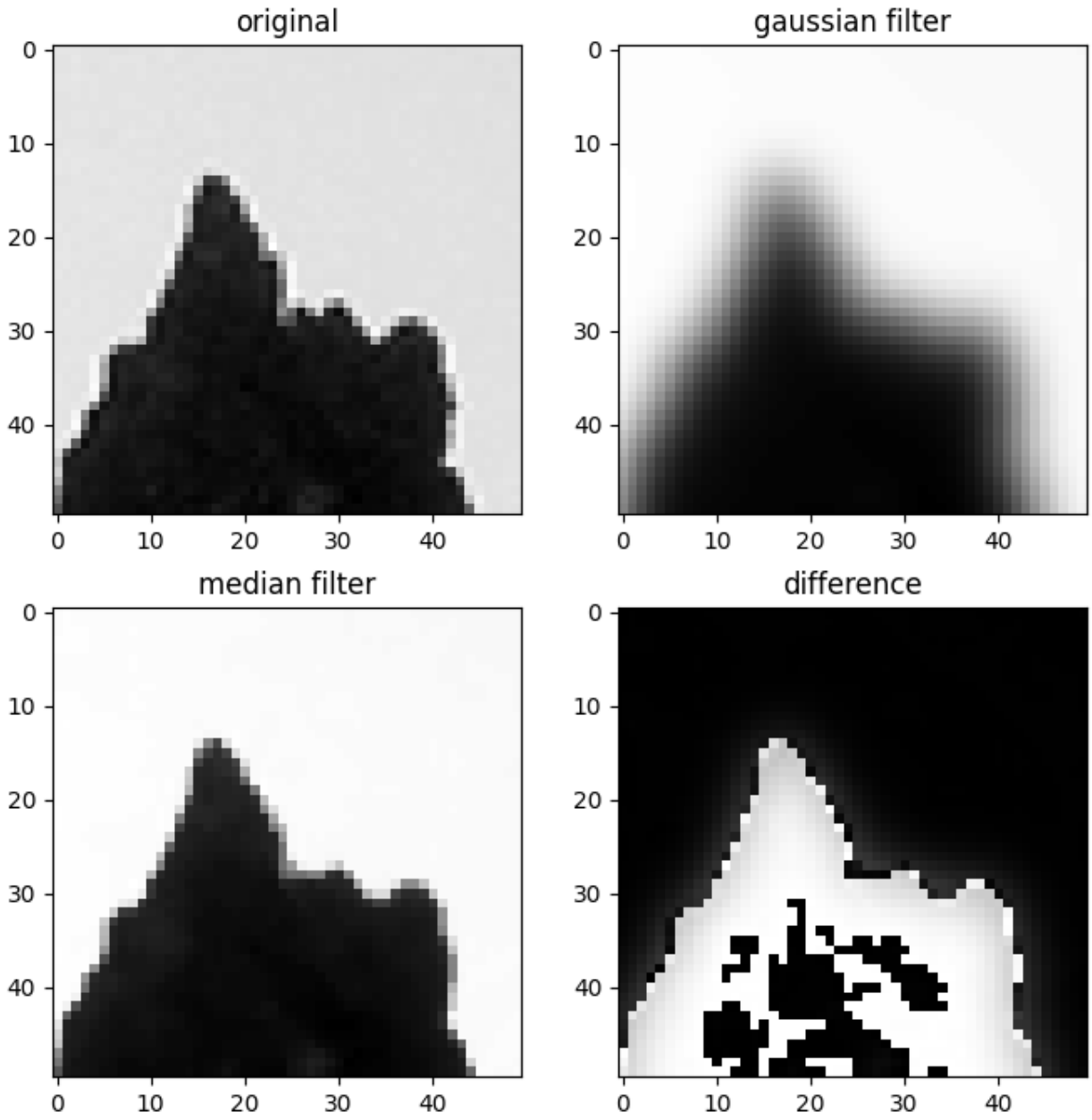

Figure 11: A zoom-in on one of the peaks of Three-fingered Jack, showing the differences between the filtering methods.

Problem 4 (Filtering and thresholding).

Solution. Since the median filter recognizes edges better, we see more defined edges in the Otsu threshold. However, the Gaussian filter doesn't consider edges as strongly, so we see less of the edge details.

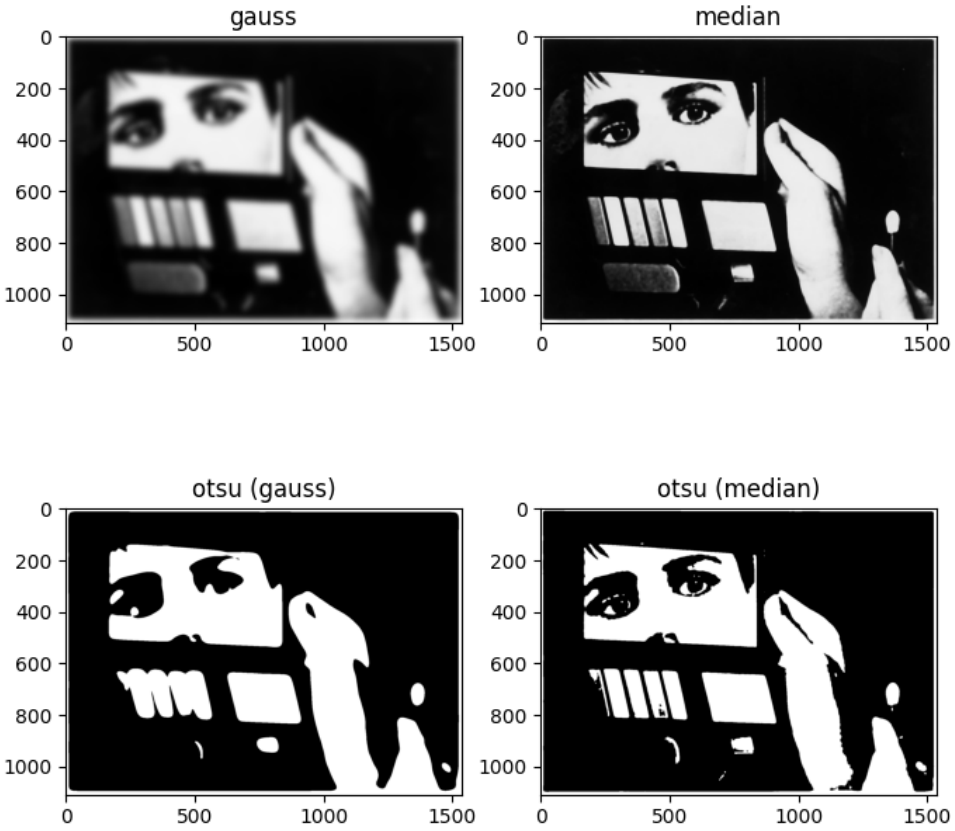

Figure 12: Different filters applied to the 'makeup' image and then thresholded using the Otsu method.

Problem 5 (High-pass filtering.).

Solution.

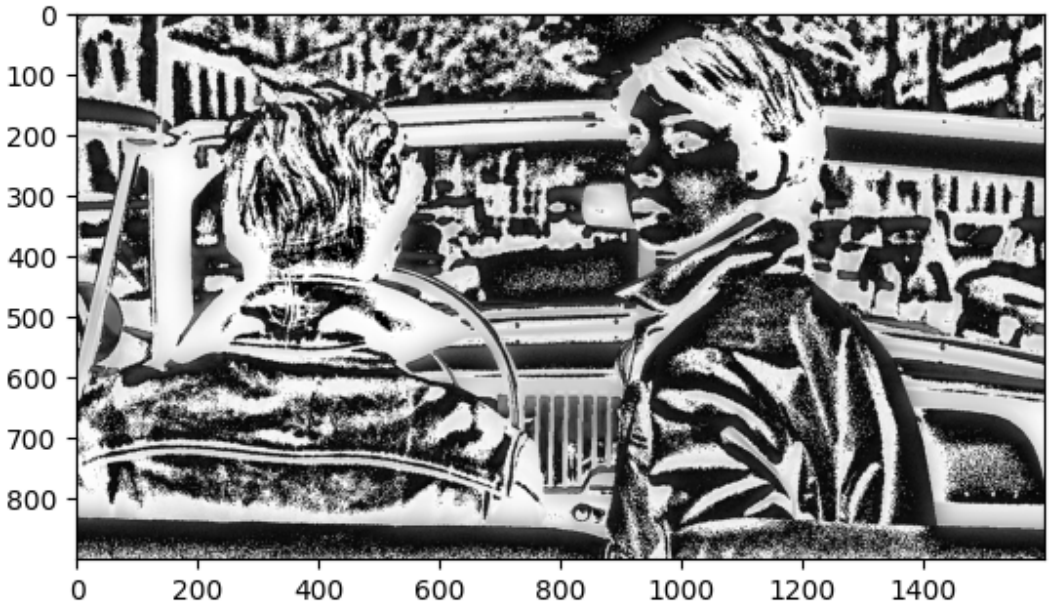

Figure 13: The 'Elevator to the Gallows' image sent through a high pass filter. We see that it preserves and even emphasizes sharp edges with big changes in intensity, producing a sort of 'negative' effect in this case. The ares without sharp edges are generally smoother.

Problem 6 (Band-pass filtering.).

Solution. The process was to take the original image, and send it through a low-pass filter with a relatively high σ value with respect to the widths given in the image. I think also created a high pass version by sending it through a filter using a smaller σ value. I then took the original image, and subtracted both the high-pass filtered and low-pass filtered images and called that the band-pass. This only had a very small effect on the original image (probably due to needing to tweak the sigma values as best as possible) so I ran it through the band-pass function a few times to accentuate that it eliminates the low and high ends while preserving the center. We can see how this affects it in Figures 16 and 17. In Figure 16 we see that the small dots are somewhat washed out while the large dots are significantly washed out, and the center columns are less affected.

```
def shift_points(array):
    array = array - np.min(array)
    scaled_array = ((array / np.max(array))*255).astype(np.uint8)

    return scaled_array

def band_pass(orig, high, low, iter=1):
    for i in range(iter):
        orig = orig - high - low

    return orig
```

Figure 14: Functions defined for the band-pass.

```
# Load up that image!
dots_tif = (io.imread('images/homework_3/gaussians_s2_to_s50_px.tif')).astype(np.float32)

dots_low = (filters.gaussian(dots_tif, sigma=40)*255).astype(np.float32)
dots_low = shift_points(dots_low)

# Try a high pass to isolate the larger dots
# dots_high = (dots_tif - dots_low).astype(np.float32)
dots_high = dots_tif - (filters.gaussian(dots_tif, sigma=15)*255).astype(np.float32)
dots_high = shift_points(dots_high)

# Now if I run it through the band filter multiple times I can isolate the middle dots even better!
band_test = band_pass(dots_tif, dots_high, dots_low, 8)
band_test = shift_points(band_test)

fig, axes = plt.subplots(2, 2, figsize=(10, 8))
ax = axes.ravel()

ax[0].imshow(dots_tif, cmap='gray')
ax[0].set_title('original')

ax[1].imshow(dots_low, cmap='gray')
ax[1].set_title('low pass')

ax[2].imshow(dots_high, cmap='gray')
ax[2].set_title('high pass')

ax[3].imshow(band_test, cmap='gray')
ax[3].set_title('band pass')

# Try plotting the row 1240 results
orig_row_1240 = dots_tif[1240, :]
band_row_1240 = band_test[1240, :]


plt.figure(figsize=(10, 6))

plt.plot(orig_row_1240, label='original row 1240')
plt.plot(band_row_1240, label='band passed row 1240')
plt.title('Intensity vs Column Number for Row 1240')
plt.xlabel('Column number')
plt.ylabel('Intensity')
plt.legend()

plt.show()
```

Figure 15: Calculation and plotting for band-pass.

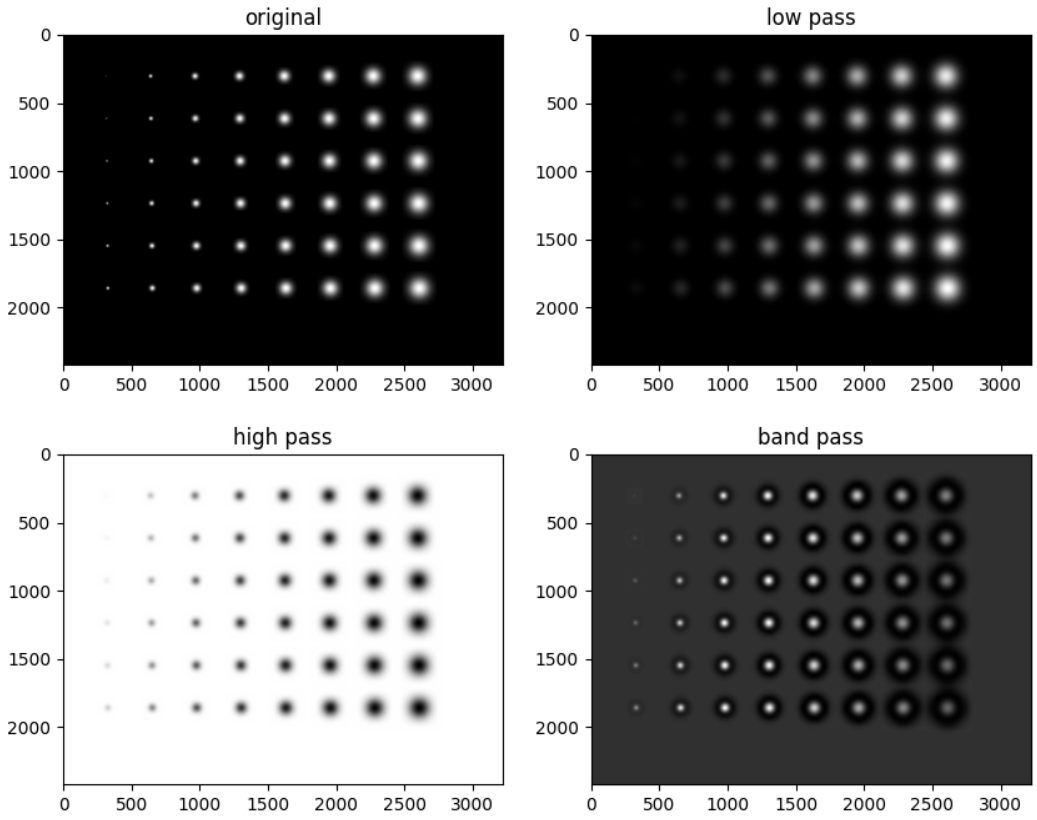

Figure 16: Results of filters.

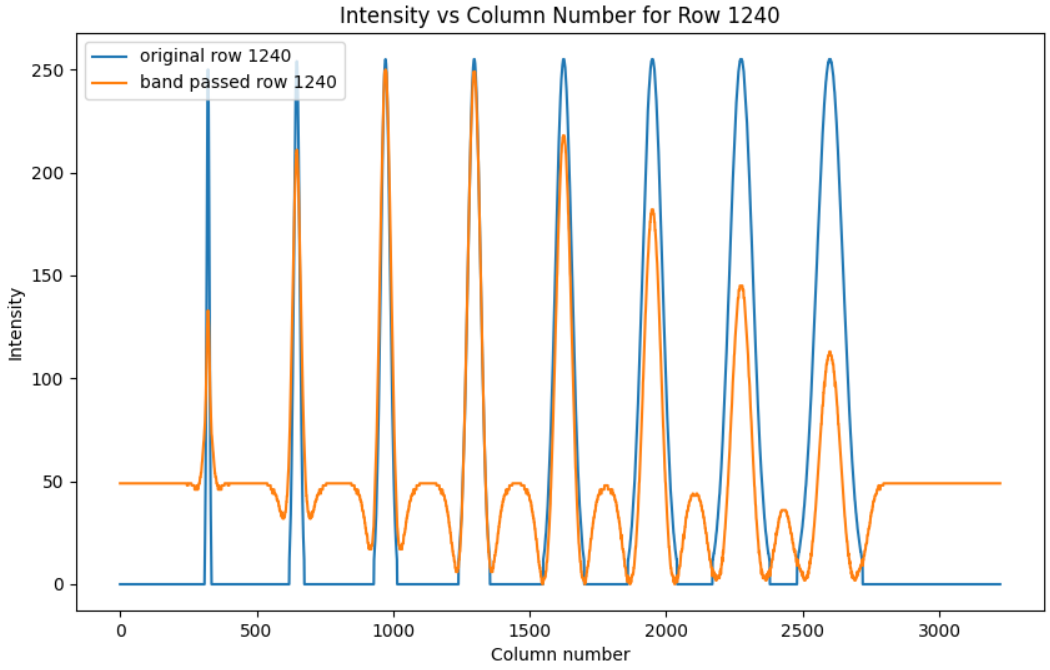

Figure 17: Intensity vs column number graph for the band-pass filtered image after 8 iterations.

Problem 7 (Signal to noise ratio).

Solution. (a) and (b): We learned in class that the signal to noise ratio is proportional to $t/\sqrt{t} = \sqrt{t}$ which we see in Figure 20 that it indeed does fall off as a function of the square root of time, t.

```python
def gen_poisson(r, t, b=2, size=(27, 27)):
    # random bg
    img = np.random.poisson(lam=b * t, size=size)

    # random pixel signal
    signal_output = np.random.poisson(lam=r * t)

    # calc center
    cent = (size[0] // 2, size[1] // 2)

    # put signal at center
    img[cent] += signal_output

    return img
```

Figure 18: Function written to generate random noise around a pixel at center also randomly chosen using the Poisson distribution.

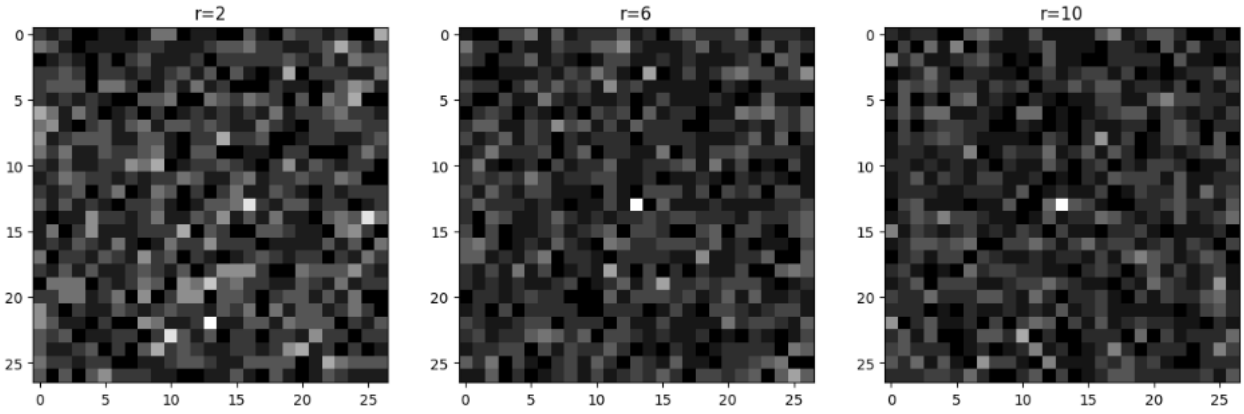

Figure 19: Three examples using different values for r intensity units/ms.

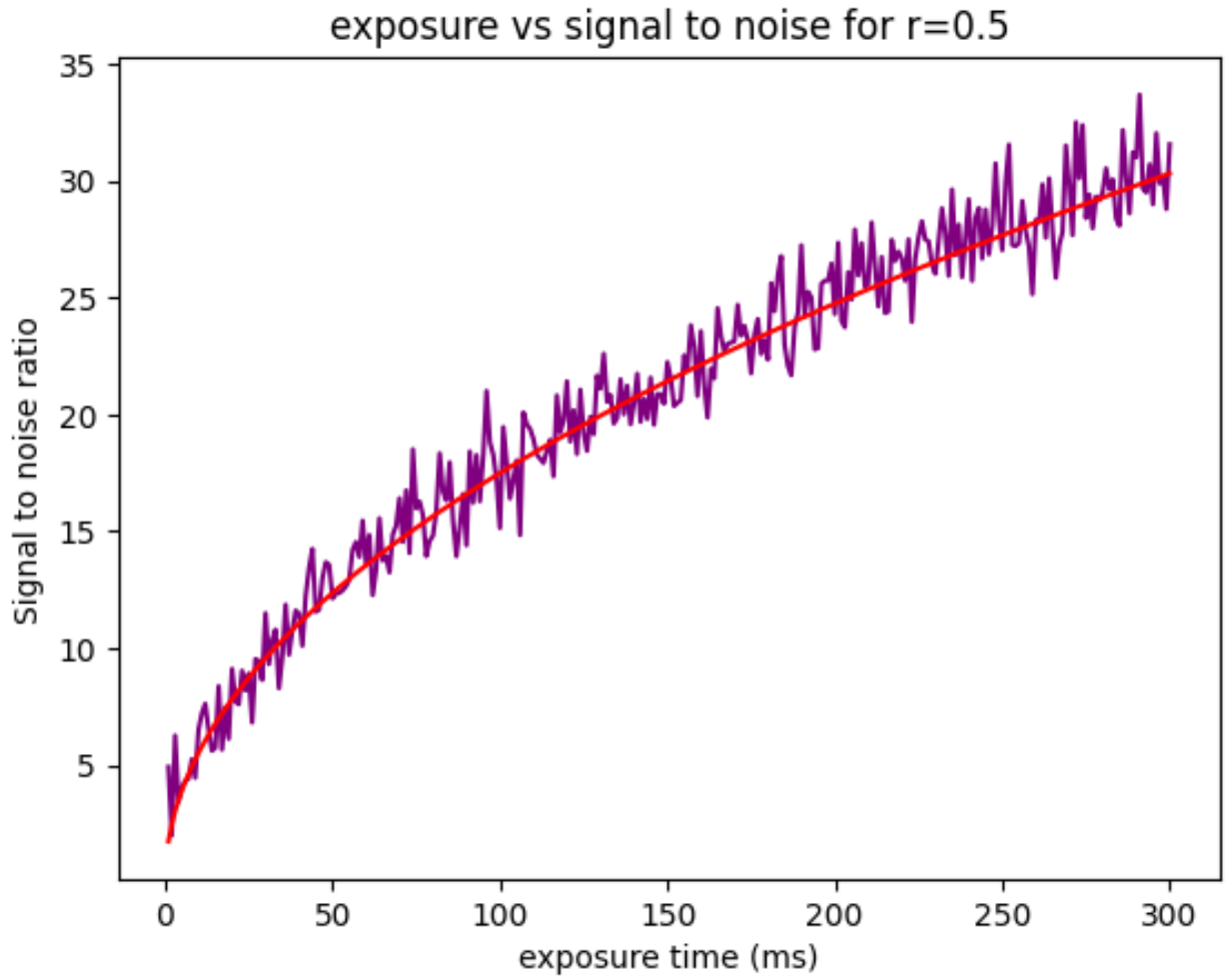

Figure 20: Signal to noise ratio vs time of exposure plot. Fit with an amplitude $A = 1.75$ times a square root function to show that it does indeed fall off as a square root as expected.

Problem 8 (A little bit of Fourier transforming).

Solution. The output of the code given seems to remove the the pillars, which are the most frequent/repetitive feature of the image, seen in Figure 21. After the highest frequency features of the image are zeroed out and the image is reconstructed, we see issues with the pillars either disappearing or being blurred out. We can see abberations of this on the sides where the zeroing has affected areas where pillars didn't exist.

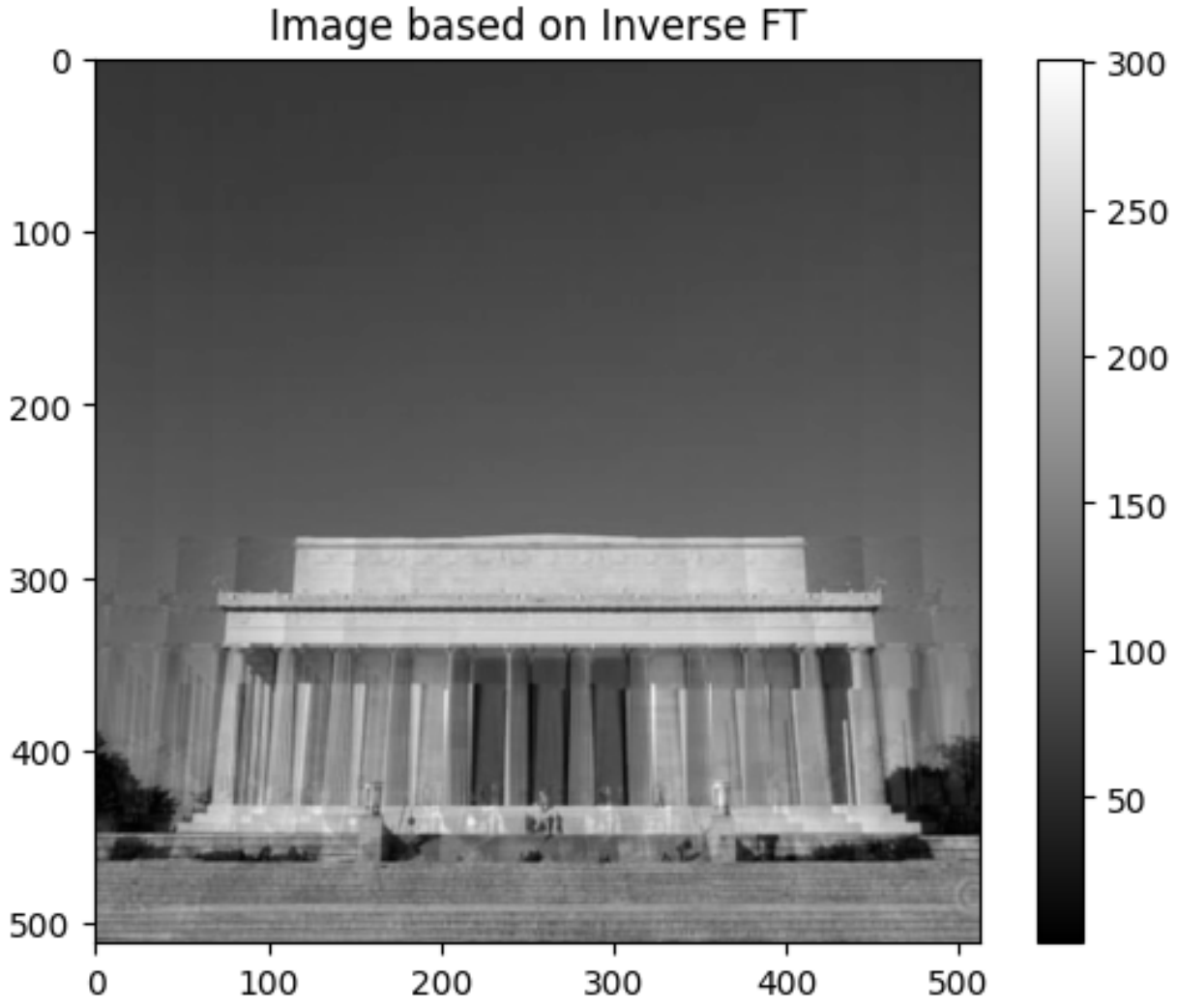

Figure 21: A photo of the Lincoln Memorial Building after undergoing a Fourier transform, a filter that zeroes out specific frequencies, then attempts to reconstruct the image via a reverse Fourier transform.

We can fix this by adding one line just before the "new-amplitude" calculation that inverts the mask: mask = 1-mask. By adding this, we effectively filter out all of the less frequent things (basically all but the pillar) and preserving them giving us the image in Figure 22, which is primarily comprised of pillars.

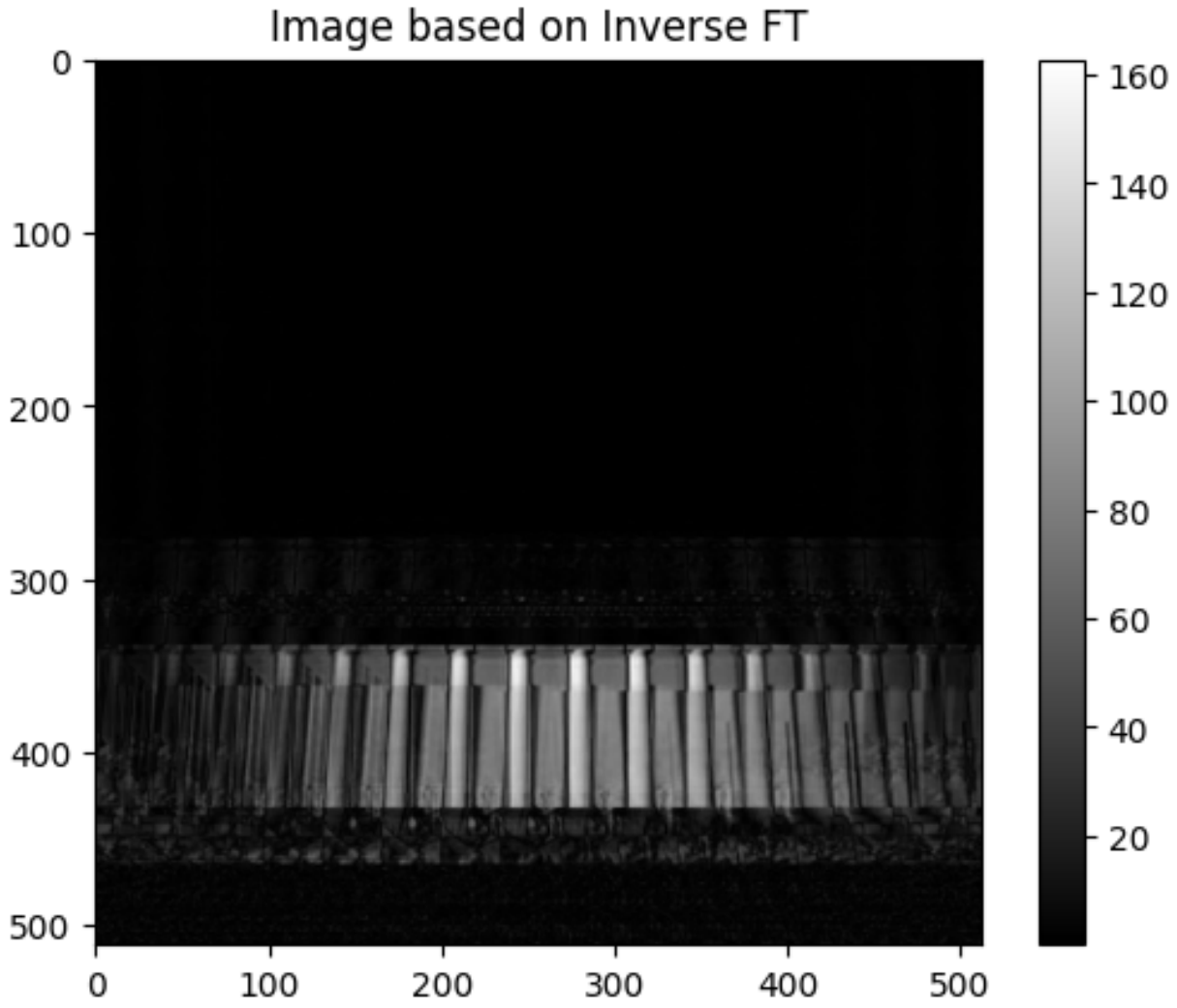

Figure 22: The effects of looking at the opposite 'filter' after applying the Fourier transform and then reconstructing the image using an inverse Fourier transform.