CS 477 HW #8

Questions completed: All undergrad level questions in MATLAB

B1. Finding the magnitude of the gradient using convolution.

Below in Figure 1, the magnitude of the gradient for the climber.tiff image is shown.

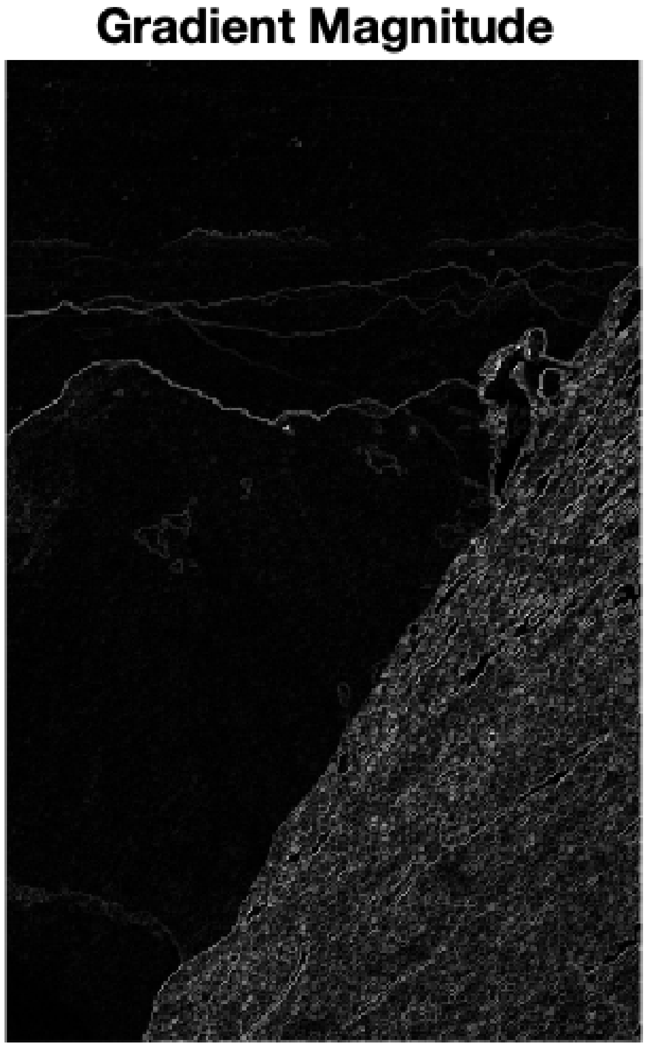

Figure 1: The gradient magnitude image was generated in MATLAB using the 'conv2()' function to find the x and y gradients. The resulting gradient matrix brightness was properly scaled to get a sensible image as shown.

First, the image was read into MATLAB, converted into grayscale, and then casted into double values. Next, the finite difference kernels for the x and y gradients were defined as ‘dx = [1 -1]’ and ‘dy = [1; -1]’, and convolution was performed doing “conv2(climber, dx, ‘same’)” (and same for dy) with the third optional argument of ‘same’ since we only want the central part of the convolution that is the size of the image. Finally, once we have our x and y gradients which are the two components of our gradient vectors, we can construct the matrix of gradient magnitudes, also known as the edge strength. Brightness scaling was done on the gradient matrix by dividing the values of the matrix by the maximum value in the matrix times 255.

B2. Finding a threshold for the gradient magnitude.

In Figure 2 below, the image of the gradient magnitude with a set threshold is shown.

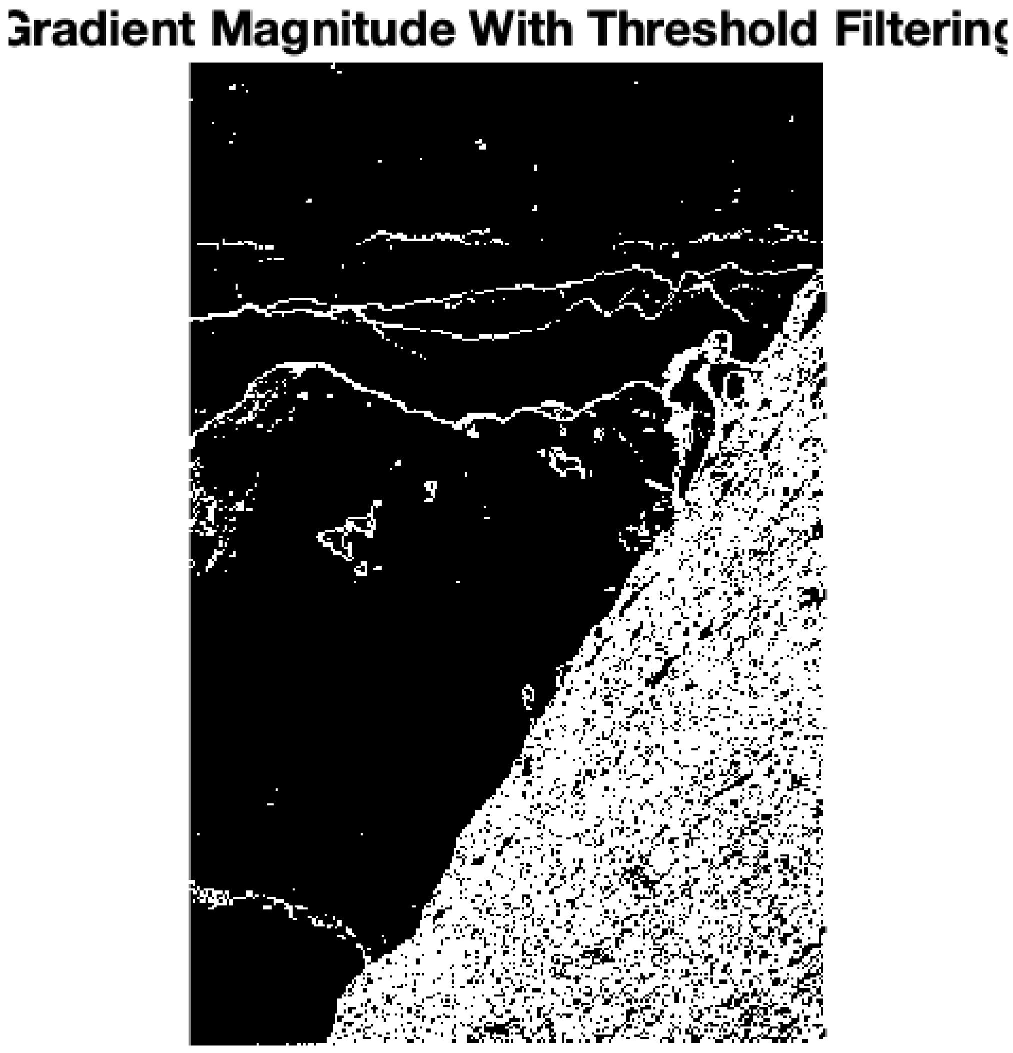

Figure 2: In MATLAB, the gradient magnitude from Figure 1 was filtered so that any gradient magnitude exceeding the threshold is set to white, otherwise set to black.

My attempt at finding a reasonable threshold to filter the gradient magnitude was first finding the max, min, mean, and median of the values in the gradient matrix so that I could get a feel for what I am working with. Upon inspecting the values, I noticed that the mean was higher than the median which suggested a right-skewed distribution. I opted to try out the mean as the threshold and decided after comparing this mean threshold to other thresholds that the mean of 18.5 for the gradient matrix threshold suffices. Anything above this mean value was set to white (255), and anything below this mean value was set to black (0) to produce Figure 2 above.

B3. Using convolution to smooth image.

Below in Figure 3, the surface map of the Gaussian kernel with $\sigma = 2$ is shown.

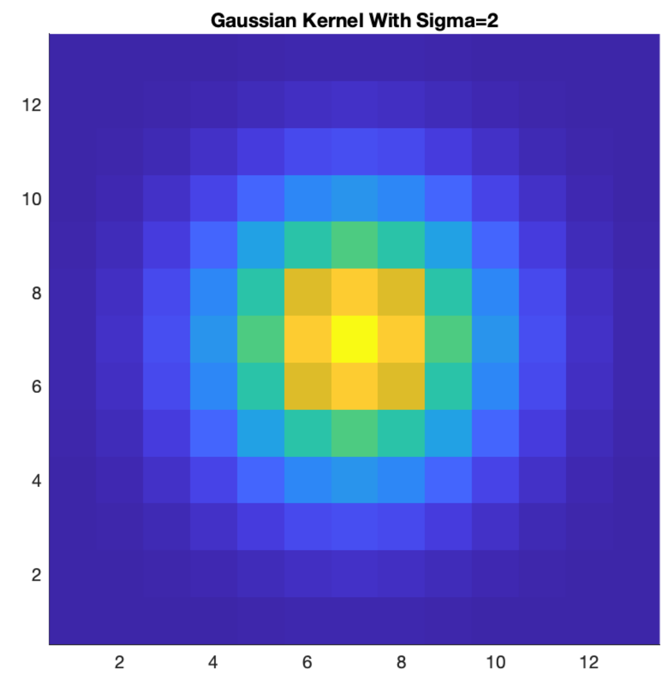

Figure 3: In MATLAB, a 13x13 Gaussian kernel with $\sigma = 2$ was created to use in convolution. The Gaussian kernel is displayed using the 'imagesc()' function which automatically scales up and color codes our Gaussian kernel matrix.

The Gaussian kernel defined as $e^{-\frac{x^2+y^2}{2\sigma^2}}$ in the lecture was created with sigma set at 2 pixels. Accordingly, the dimensions of the kernel was computed as $DIM = 2*3*\sigma+1 = 13$ to get a 13x13 kernel. This ensures that the distribution is centered within the kernel and that we are capturing 99% of the distribution in both directions. Truncating the kernel at 3σ maintains a balance of capturing most (99%) of the Gaussian spread while at the same time limiting the kernel size in order to increase computational efficiency (smaller kernel size is better). The Gaussian kernel was then scaled by dividing its values by the sum of its values in order to get a kernel whose values summed up to

1. This scaling is done so that when convolution is used with the Gaussian kernel to smooth the image, the output image is displayable as shown next.

Below in Figure 4, Gaussian smoothing of the original climber.tiff image is shown.

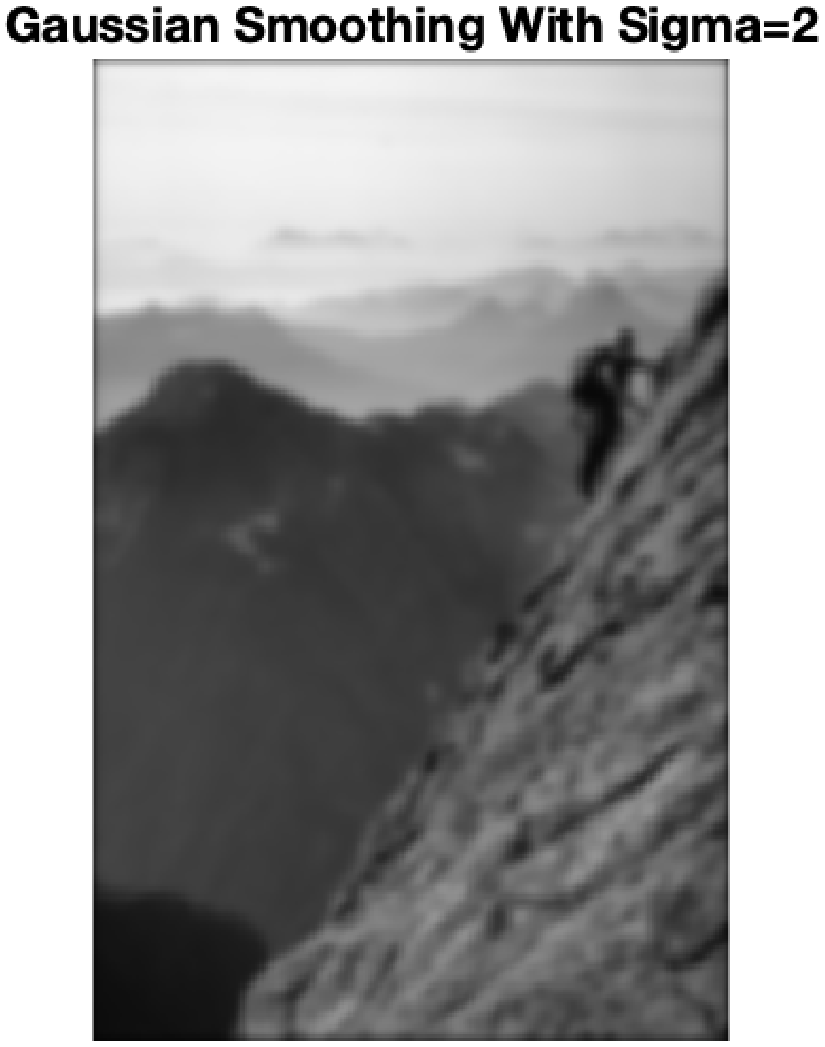

Figure 4: In MATLAB, the Gaussian kernel as previously described was used to perform convolution using 'conv2()' to smooth the original climber.tiff image.

To perform the smoothing through convolution with a Gaussian kernel as seen above in Figure 4, the function 'conv2()' was used with the inputs being the original climber.tiff image, our Gaussian kernel, and again passing ‘same’ as the third parameter. We can see in Figure 4 that our image indeed has been smoothed.

B4. Blurring followed by edge detection.

Below in Figure 5, edge detection and binary threshold filtering were performed on the blurred image as shown.

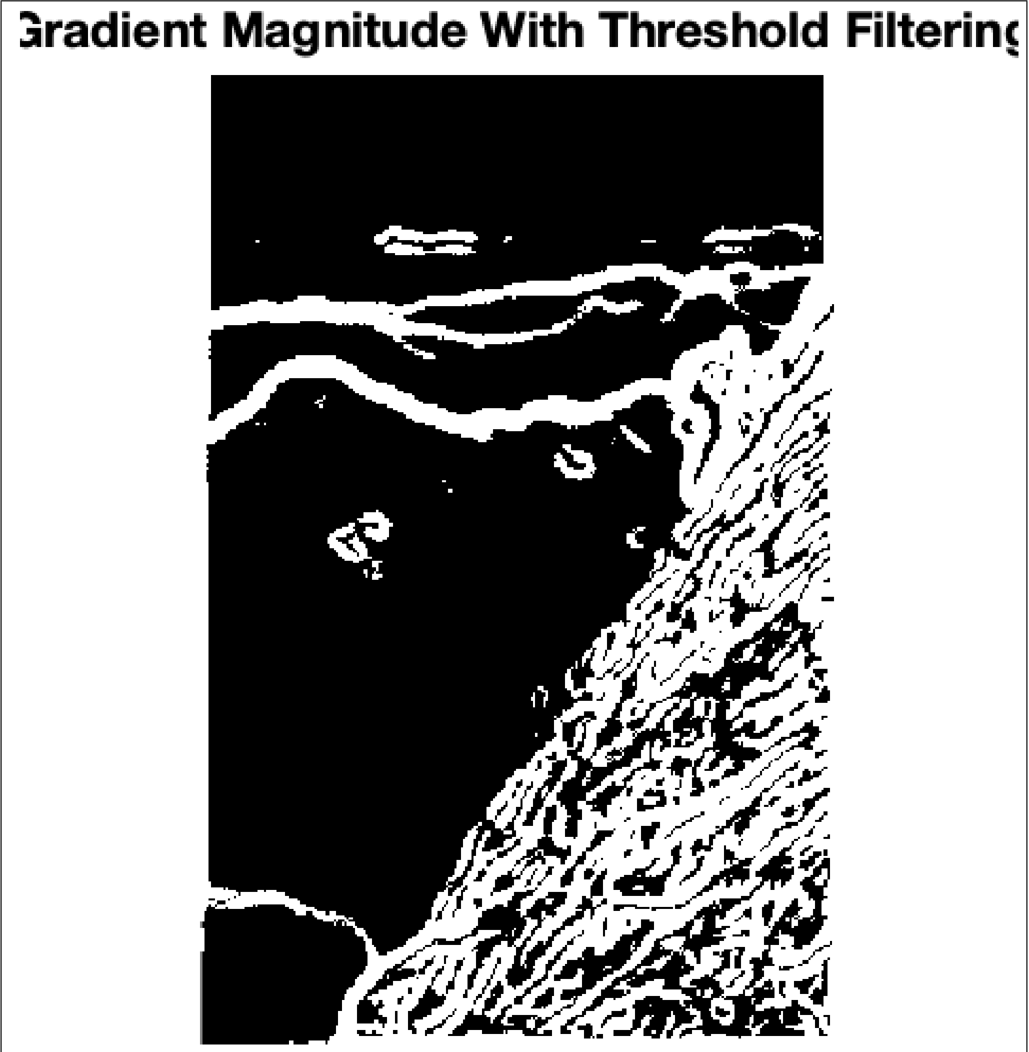

Figure 5: In MATLAB, edge detection and threshold filtering was done for our newly smoothed image from Figure 4. The same MATLAB operations were as described in parts B1 and B2 were used.

When comparing the edge detection with threshold on the blurred image compared to the edge detection with threshold on the original image as seen in B2 Figure 2, it appears that the edge lines are much thicker than before. Whether that improved the edge detection quality or not, I cannot say, but the edge detection produced in Figure 2 seemed like it sufficed while this new edge detection seen in Figure 5 seems overkill in terms of the thickness of edge boundaries.

B5. Combining blurring and edge detection into one filter.

In Figure 6 below, the climber.tiff image is displayed after going through a combined blurring and edge detection filter.

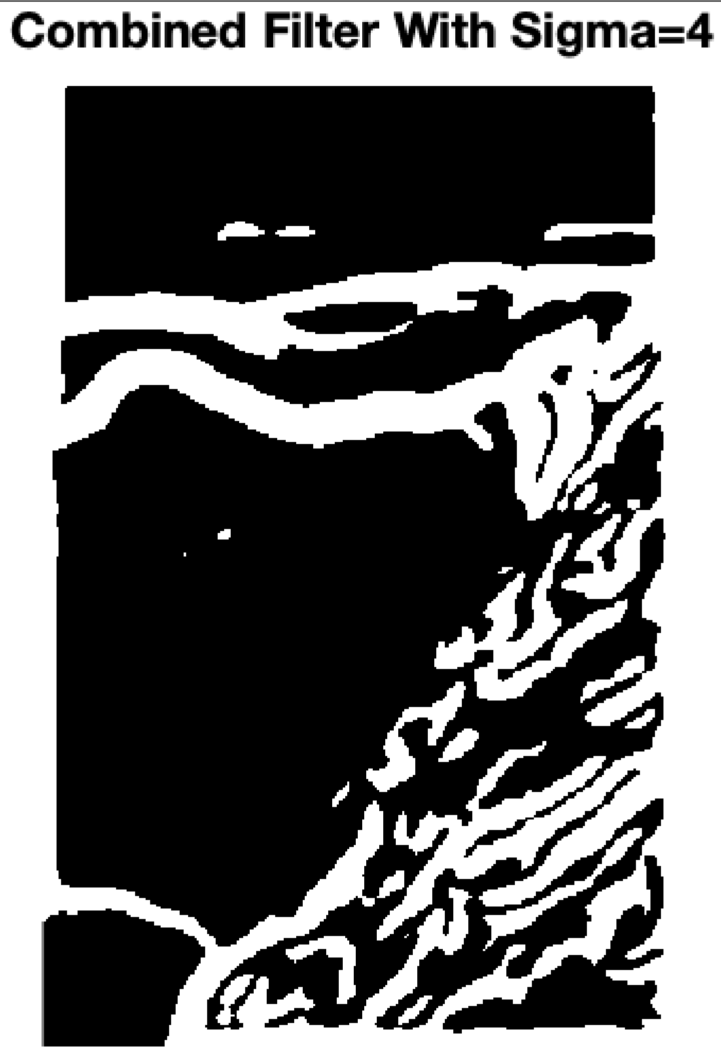

Figure 6: In MATLAB, a combined blurring and edge detection kernel is applied to the climber.tiff image. The Gaussian component for the blurring was set to $\sigma = 4$.

The Gaussian kernel and the simple finite difference edge detection kernels (one for x and one for y) were combined through convolution using 'conv2(gauss, edge, 'same')' so that we have two newly combined kernels for the x and y directions. The gradient matrix was computed and threshold filtered the same way as in described in B1. The blurring and edge detection convolutions were also performed separately to compare to our combined filter, and the resulting image matched well with our combined filter result with the observation that the combined filter had slightly better detection of edges using the same threshold, so perhaps combining kernels is a more streamlined process.

B6. Re-implementing smoothing using 1D convolution.

To prove that the Gaussian blurring function is separable:

$$f(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$f(x,y) = e^{\frac{-x^2-y^2}{2\sigma^2}}$$

$$f(x,y) = e^{-\frac{x^2}{2\sigma^2}-\frac{y^2}{2\sigma^2}}$$

$$f(x,y) = e^{-\frac{x^2}{2\sigma^2}}e^{-\frac{y^2}{2\sigma^2}}$$

$$Let\ g(x) = e^{-\frac{x^2}{2\sigma^2}},\ h(y) = e^{-\frac{y^2}{2\sigma^2}}$$

$$Then,\ f(x,y) = g(x)h(y)$$

Thus, we have proven that Gaussian blurring is a separable function. This means that convolution with $f(x,y)$ is the same as a 1D convolution by $g(x)$ followed by a 1D convolution by $h(y)$.

Following this method, separate 1D Gaussian vectors were created for the x and y direction with $\sigma = 2$ and applied to the climber.tiff using "conv2(X, Y, image, 'same')" where X and Y are the Gaussian vectors, image is climber.tiff, and 'same' is the parameter as described previously. The resulting smoothing was the same result as Figure 4 in B3.

I expect two 1D Gaussian vector convolutions to be faster than the single Gaussian distribution matrix that we have been using because the size of the kernel directly reflects the amount of computation that is required for convolution of an image. Each convolution operation involves ‘sliding’ the kernel over the image and performing the computations at each pixel, so I am guessing that the computation complexity is $O(i * j * k^2)$ for an image with i rows, j columns, and a $k * k$ Gaussian kernel. Since our Gaussian kernel is separable as we have proven above, we can do two 1D convolutions which reduces the computation complexity to $O(i * j * 2 * k) = O(i * j * k)$.

The speeds were tested by putting the two different convolutions in a loop of 10,000 iterations each. Separating the Gaussian kernel approach took 1.82 seconds while the Gaussian matrix took 7.04 seconds. It is hard to say from this whether the exact time complexities I listed above holds for these two approaches as there is an unknown constant number of operations and overhead in calling and using the conv2() function, but the basic theory holds that convolutions are faster for smaller kernels.