CS5404 Homework 1 Report

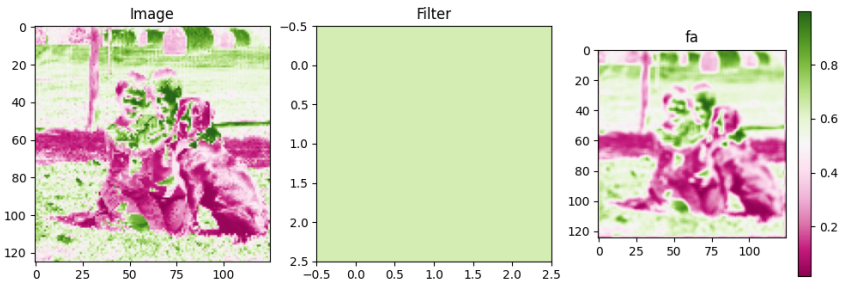
Joshua Santy

September 19, 2025

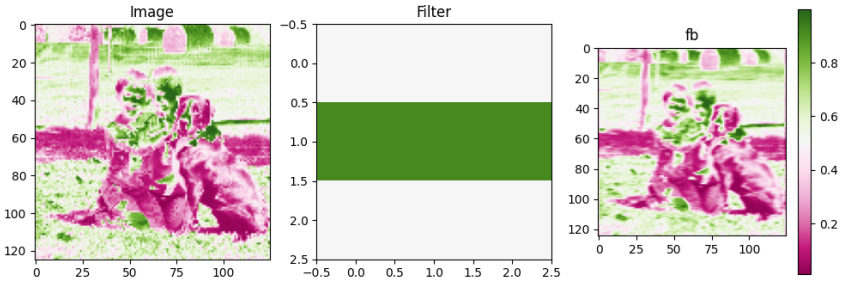
Problem 1

For all sections of this homework (except for problem 3), the following image will be used:

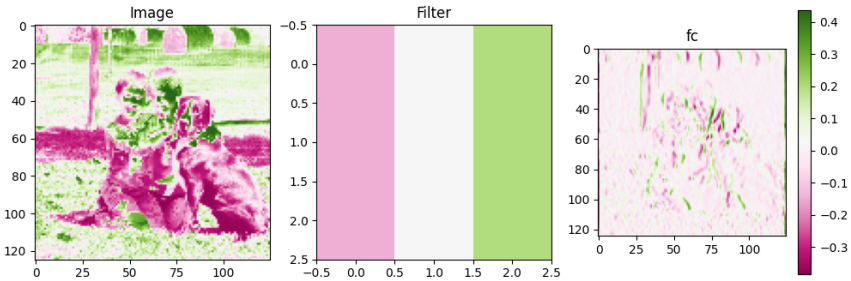
Problem 1.1.1



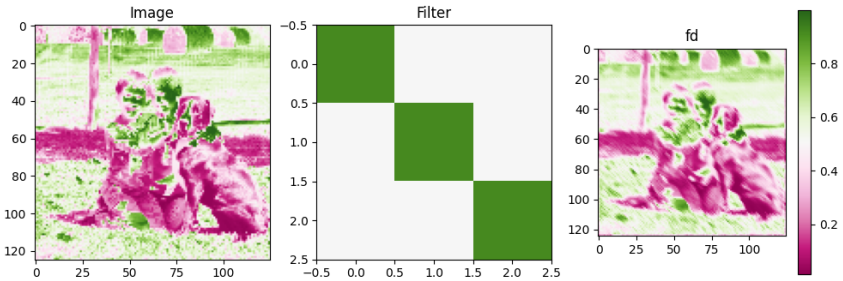
Filter A



Filter B



Filter C



Filter D

Figure 1: Original image with each filter applied

Filter A appears to be a box filter, taking the average of the equally-weighted 9 pixels.

$$\textbf{Filter A: } \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \text{ or when separated, } \frac{1}{3}\begin{bmatrix}1\\ 1\\ 1\end{bmatrix} * \frac{1}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix}$$

Filter B appears to be a horizontal blur with the middle row of 1's. It is horizontal since pixels from columns 1 and contain 3 are considered in the filtering, and zeros in the first and third row lead to the vertical component having no effect.

$$\textbf{Filter B: } \frac{1}{3}\begin{bmatrix}0 & 0 & 0\\ 1 & 1 & 1\\ 0 & 0 & 0\end{bmatrix} \text{ or when separated, } \begin{bmatrix}0\\ 1\\ 0\end{bmatrix} * \frac{1}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix}$$

Filter C appears to be edge detection (similar to the Sobel horizontal filter). It is similar in that it follows the pattern of having negative-, zero-, and positive-valued columns, but differs since the first and third columns are equally weighted, while the horizontal Sobel filter is not, containing a 2 on row 2.

$$\textbf{Filter C: } \frac{1}{6}\begin{bmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{bmatrix} \text{ or when separated, } \begin{bmatrix}1\\ 1\\ 1\end{bmatrix} * \frac{1}{6}\begin{bmatrix}-1 & 0 & 1\end{bmatrix}$$

Filter D appears to be a diagonal blur, from top left to bottom right. It cannot be separated since, being the identity matrix, it has a rank of 3, as well as being a 3x3 square matrix. A separable matrix must have a rank of at most 1.

$$\textbf{Filter D: } \frac{1}{3}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Problem 1.1.2

The horizontal Sobel filter, defined as

$$\begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix}$$

and the vertical Sobel filter, defined as

$$\begin{bmatrix}1 & 1 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix}$$

were applied to the image. Figure 2 shows the filters applied to the previously scaled-down version of the image.

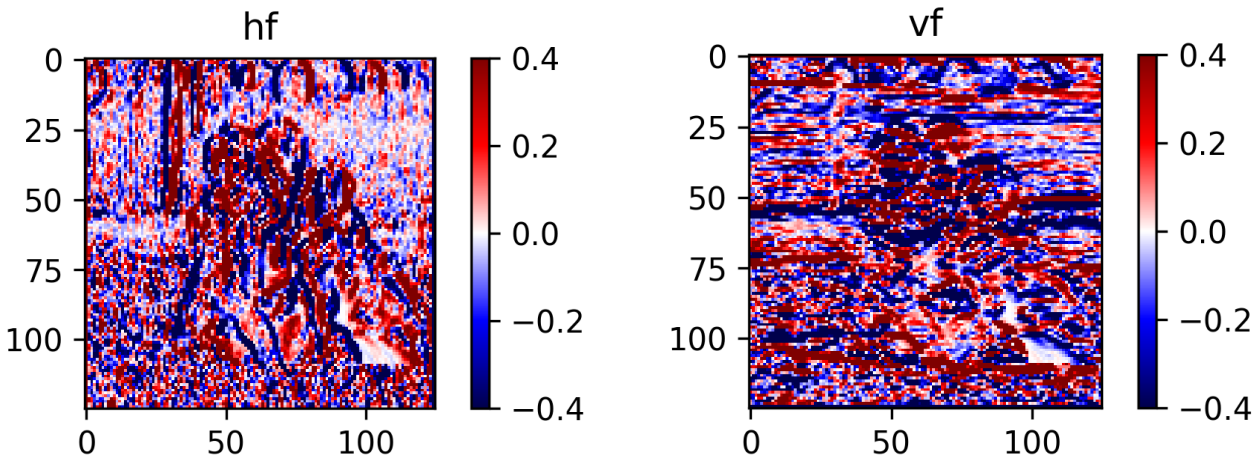


Figure 2: Original down-scaled image with horizontal (left) and vertical (right) Sobel filters applied

However, due to the composition of the image (lots of grass and many edges present), these two filtered images look messy and noisy, so the horizontal and vertical Sobel filters were reapplied to the original resolution images in Figure 3 , providing a clearer output image. In both sets, however, it can clearly be seen that the horizontal Sobel filter brings out the vertical lines, and the vertical Sobel filter brings out the horizontal lines.

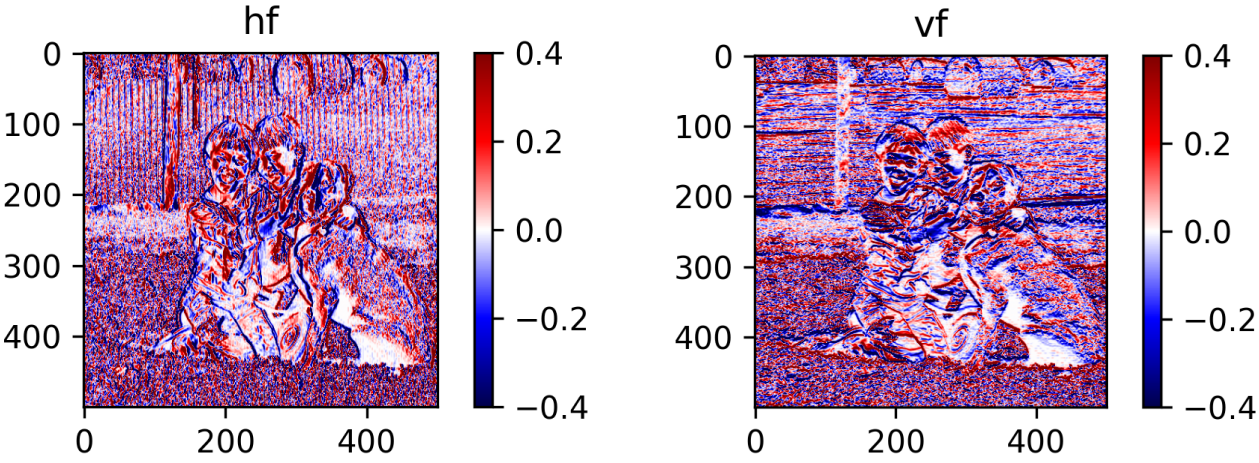


Figure 3: Original resolution image with horizontal (left) and vertical (right) Sobel filters applied

Next, the gradient amplitude and angle information of the images were plotted (Figure 4).

$$\nabla I = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right] \quad \text{Gradient}$$

$$\theta = \tan^{-1}\left(\frac{\partial I}{\partial y} / \frac{\partial I}{\partial x}\right) \quad \text{Gradient Angle}$$

$$\|\nabla f\| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2} \quad \text{Gradient Amplitude}$$

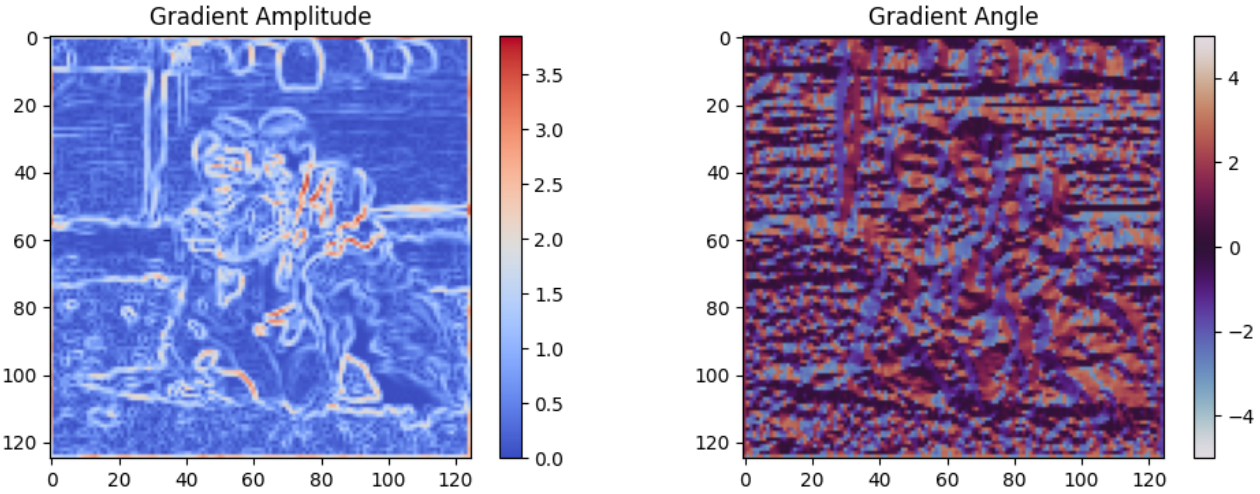


Figure 4: Gradient Amplitude (Left) and Gradient Angle (Right)

Problem 1.1.3

To derive the Laplacian Filter, we start from the definition of the derivative

$$I'(x) = \lim_{h \to 0} \frac{I(x+h) - I(x)}{h}$$

Since we are dealing with a discrete pixel grid, h can be 1 at its minimum. The second derivative can then be calculated as

$$\begin{aligned} I'(x) &= I(x+1) - I(x) \\ I'(x+1) &= I(x+2) - I(x+1) \\ I''(x) &= I'(x+1) - I'(x) \\ &= I(x+2) - I(x+1) - I(x+1) + I(x) \\ &= I(x+2) - 2I(x+1) + I(x) \end{aligned}$$

The derivative introduces a half-pixel shift and the second derivative introduces a 1-pixel shift. Therefore, we can re-center by shifting to the left one pixel; this is the same as convolving with the filter $\begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$. Expressed as a 3x3 matrix and adding its transpose (equivalent to convolving with the kernel), we get the Laplacian filter matrix:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Applying this filter yields the following image:

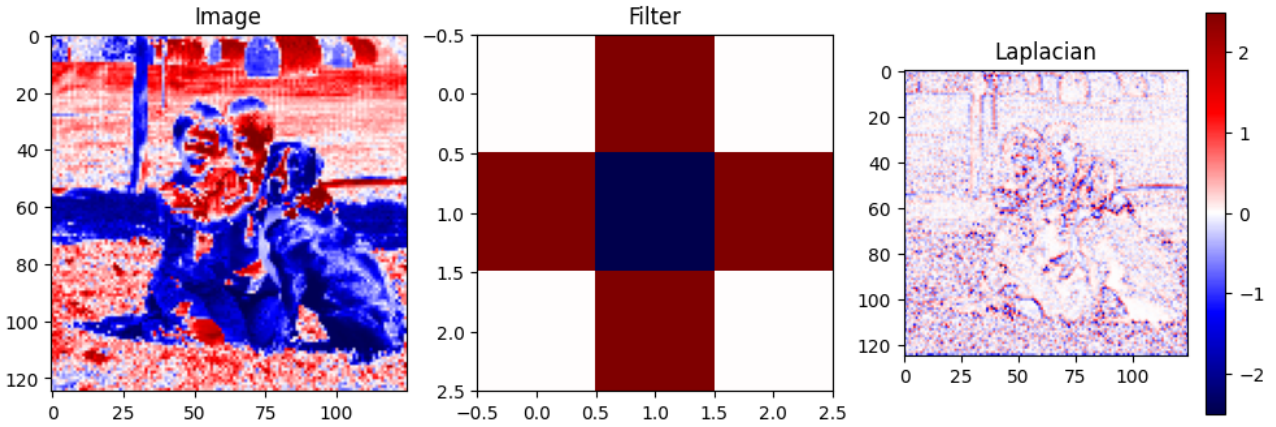


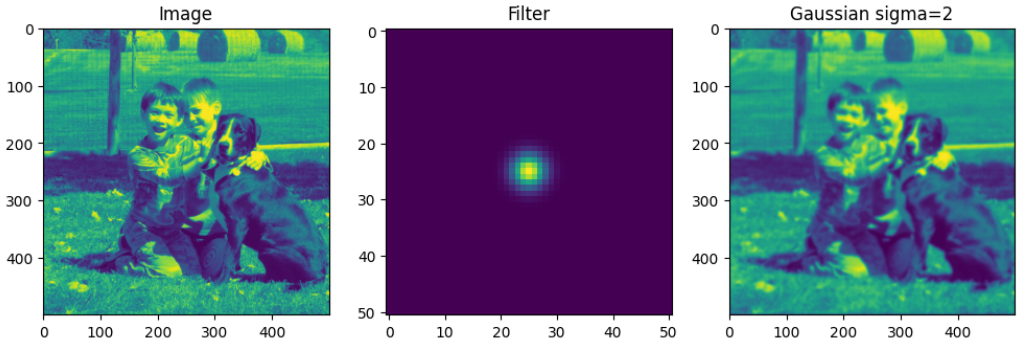
Figure 5: Original image with Laplacian filter applied

Problem 1.1.4

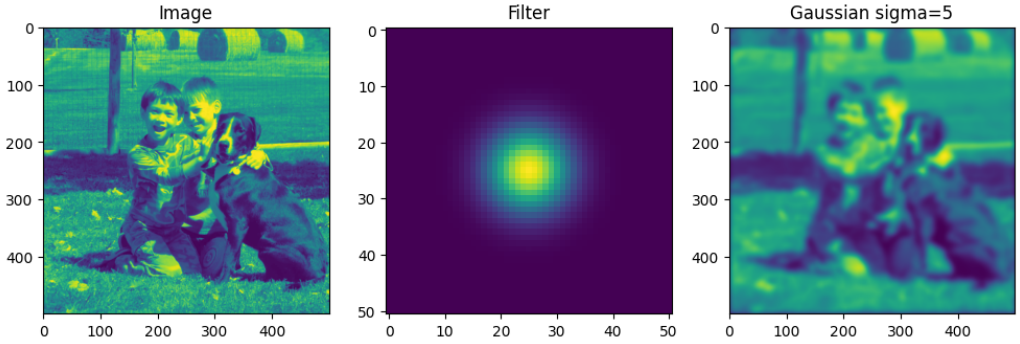
The Gaussian Filter is given by

$$G(x, y) = a \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

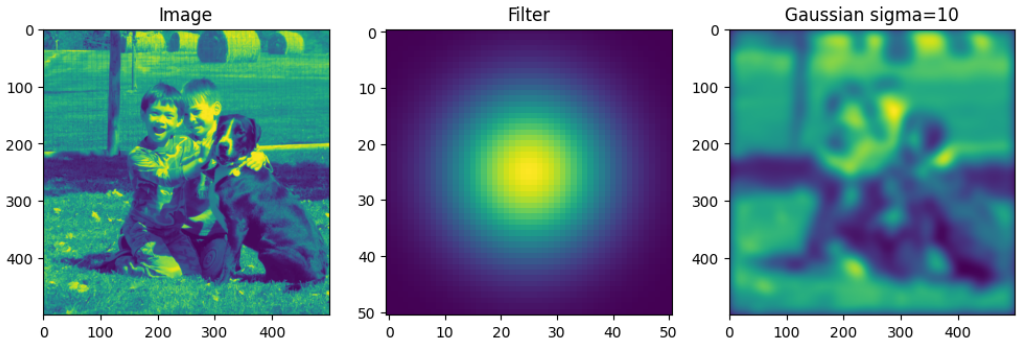
where x and y are distance (in pixels) from the center of the image, σ is the standard deviation, and a is a parameter. Three σ values (2, 5, and 10) were used with a 51×51 Gaussian filter on the original image. The results are shown in Figure 6.



Gaussian Filter, $\sigma = 2$



Gaussian Filter, $\sigma = 5$



Gaussian Filter, $\sigma = 10$

Figure 6: Original image with each Gaussian filter applied

If σ is too large or if the filter size is too small, then the diameter of the Gaussian "blur" is too large for the filter and bleeds out the sides of the filter, shown in Figure 7

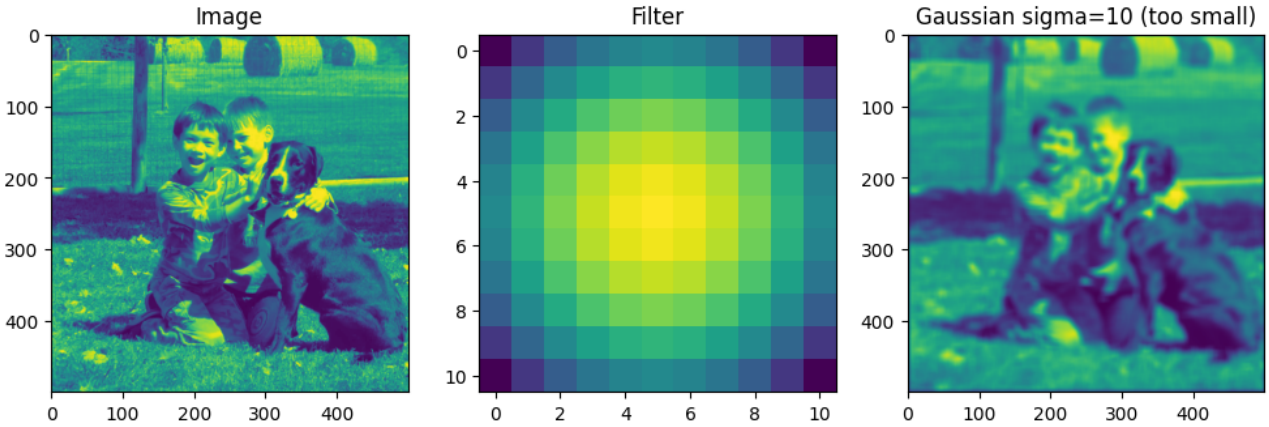


Figure 7: Gaussian with size of 11, $\sigma = 10$

For the Gaussian Filter

$$g = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

its σ value is $\sqrt{-\frac{1}{2\ln(1/2)}} \approx 0.8493$. This value was derived by finding the value of **a** using the center pixel, then using pixel (1,0) to determine σ as follows:

$$g = \begin{bmatrix} 1/16 & 2/16 & 1/16 \\ 2/16 & 4/16 & 2/16 \\ 1/16 & 2/16 & 1/16 \end{bmatrix}$$

$$G(0,0) = a = \frac{1}{4}$$

$$G(1,0) = \frac{1}{8} = \frac{1}{4}e^{-\frac{1}{2\sigma^2}}$$

$$\ln\frac{1}{2} = -\frac{1}{2\sigma^2}$$

$$\sigma = \sqrt{-\frac{1}{2\ln(1/2)}}$$

Problem 1.1.5

Lastly, for problem 1, it was shown that taking the derivative of a filtered image is the same as taking the derivative of the filter, then applying it to the image. More formally, using the definition of the convolution,

$$\frac{\partial}{\partial x}(f * g) = \frac{\partial f}{\partial x} * g$$

To show this equivalence, the box filter was used, and convolutions with the horizontal Sobel filter provided the derivative. Each order of applying differentiation and filtering was plotted, as well as their difference. Since their difference is an empty image, this helps empirically prove this relation.

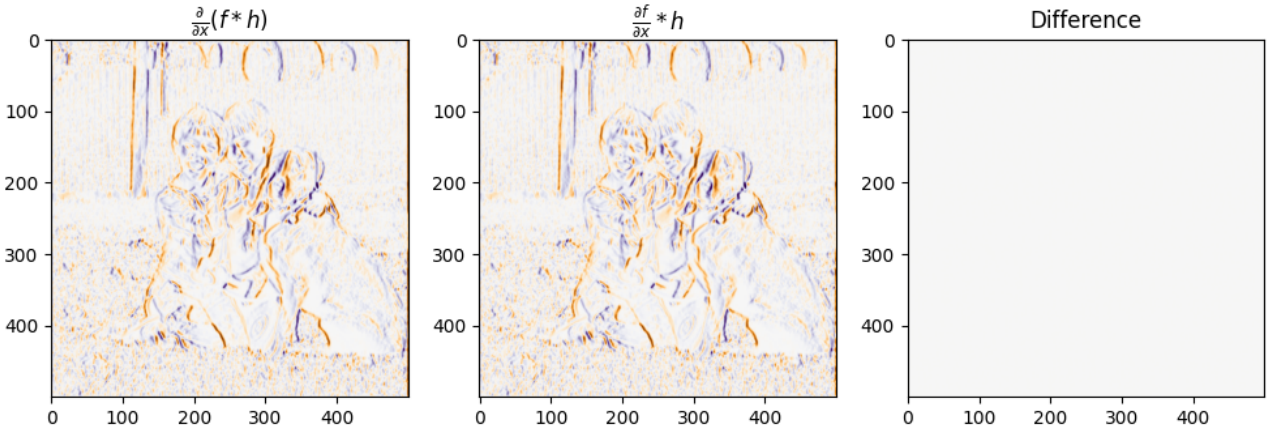


Figure 8: Differentiation and Filter Relation Plots

Problem 2

Problem 1.1.2

Problem 1.2.1 starts with 1D Upsampling Kernels. Given a generated signal and a new target value, three different interpolation techniques were implemented: nearest neighbor, linear, and cubic interpolation. The Python code for linear and cubic are shown here:

```
#Linear Interpolation Implementation:
def interp_linear(signal, new_x):
x = new_x
x0 = int(np.floor(new_x))
x1 = x0 + 1

#Upper Bound
if x1 >= len(signal):
    return signal[-1] #avoiding divide by zero if x1=x0

y0 = signal[x0]
y1 = signal[x1]

return y0+ (x-x0)*(y1-y0)/(x1-x0)
```

```
#Cubic Interpolation Implementation:
def interp_cubic(signal, new_x):

def cubicFunction(x, a=-0.5):
    ax = abs(x)
    if ax < 1:
        return 1 - (a+3)*ax**2 + (a+2)*ax**3
    elif ax < 2:
        return a * (ax-1) * (ax-2)**2
    else:
        return 0.0

def getBase(signal, idx):
    if 0 <= idx < len(signal):
        return signal[idx]
    else:
```

```
            return 0.0

    #Need to get 4 values - 2 to the left and 2 to the right
    x0 = int(np.floor(new_x))-1 #2 to the left
    x1 = int(np.floor(new_x)) #1 to the left
    x2 = int(np.floor(new_x))+1 #1 to the right
    x3 = int(np.floor(new_x))+2 #1 to the right

    #Check bounds:
    base_0 = getBase(signal, x0)
    base_1 = getBase(signal, x1)
    base_2 = getBase(signal, x2)
    base_3 = getBase(signal, x3)

    x0Dist = new_x - x0
    x1Dist = new_x - x1
    x2Dist = new_x - x2
    x3Dist = new_x - x3

    cubic0 = cubicFunction(x0Dist, -0.5)
    cubic1 = cubicFunction(x1Dist, -0.5)
    cubic2 = cubicFunction(x2Dist, -0.5)
    cubic3 = cubicFunction(x3Dist, -0.5)

    return cubic0 * base_0 + cubic1 * base_1 + cubic2 * base_2 + cubic3 * base_3
```

Figure 9 shows the original signal and the three implemented interpolation methods. Nearest neighbor interpolation solely uses the value of the nearest point, and no other neighboring values. This method leads to a piece-wise interpolation with straight, vertical lines connecting flat, horizontal lines marking the value of the point. Linear interpolation uses two points (a left and right point) to determine the y value of a point at any given x-value of interest, giving the equation for the linear line intersecting those three points. The equation for the y value, given known points (x_0, y_0) and (x_1, y_1), and the new value of x, is given by

$$y = \frac{y_0(x_1 - x) + y_1(x - x_0)}{x_1 - x_0}$$

Lastly, cubic interpolation uses 4 points (2 to the left, 2 to the right of the target x-value). It uses the following piecewise equations to determine a value:

$$h(x) = \begin{cases} 1 - (a+3)x^2 + (a+2)|x|^3 & |x| < 1 \\ a(|x| - 1)(|x| - 2)^2 & 1 \leq |x| < 2 \\ 0 & \text{otherwise} \end{cases}$$

Nearest neighbor provides a blocky fit, linear interpolation does better with the long vertical lines but suffers at the end points, and cubic fits the most closely of the three methods.

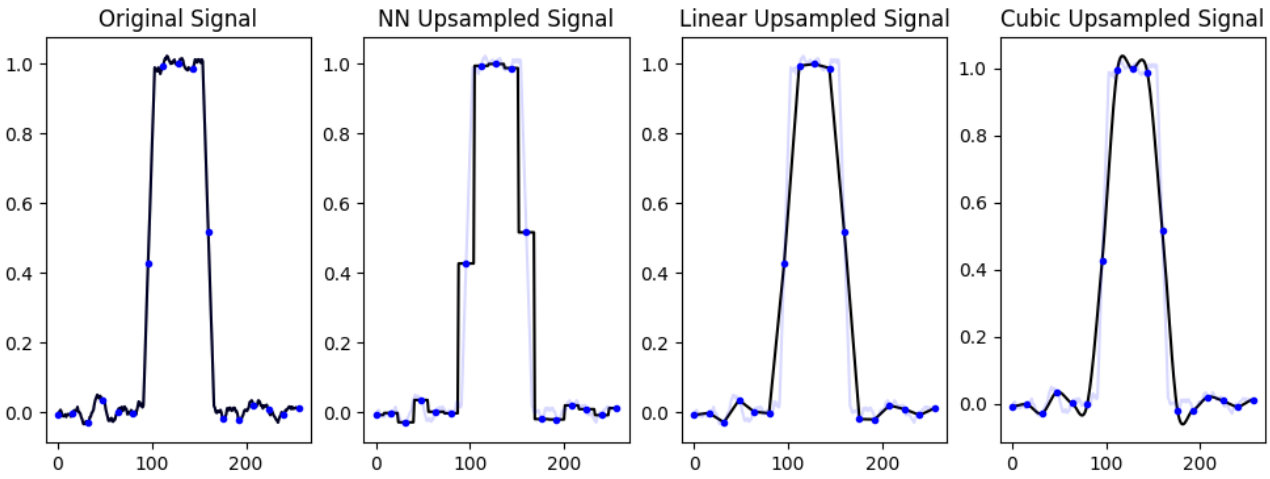


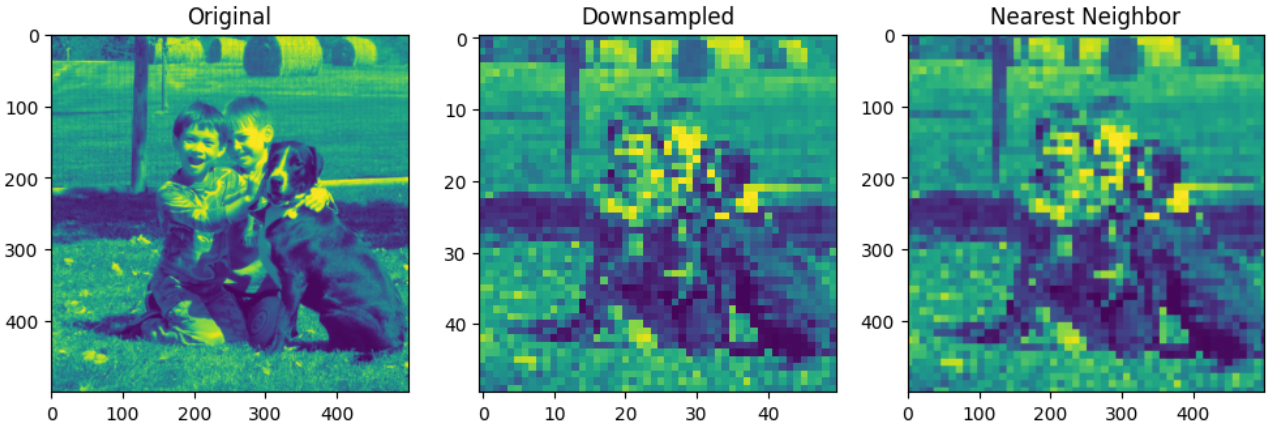
Figure 9: Original Signal and Nearest Neighbor, Linear, and Cubic Interpolation

Problem 1.2.2

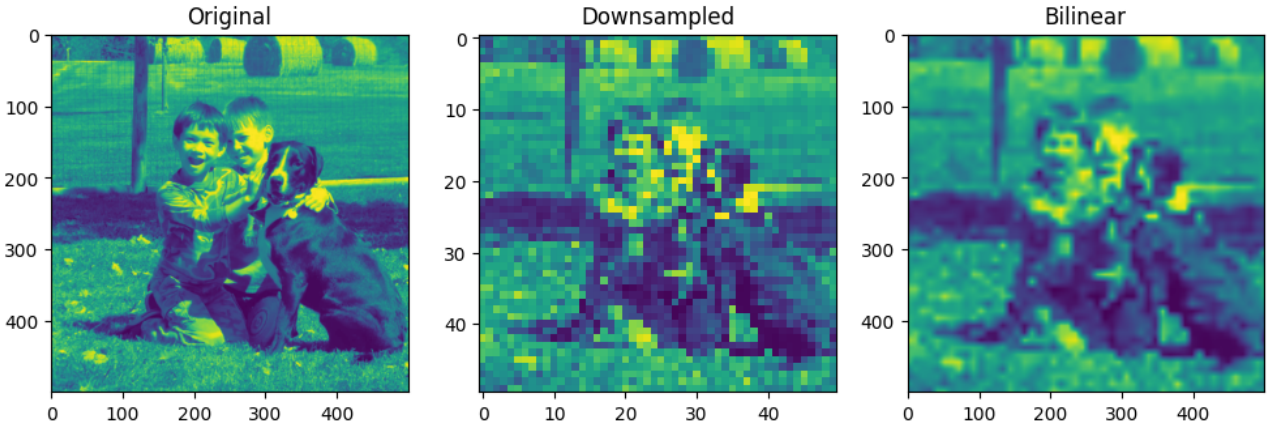
Next, 2D nearest neighbor, bilinear, and bicubic interpolations were implemented and applied to the image. The implementations are as follows, and the interpolated images are shown in Figure 10.

```
#Bilinear Interpolation
def interp_bilinear(image, new_x, new_y):
#Interp along x, then y:

height = len(image)#rows
width = len(image[0]) #columns

x = new_x
y = new_y

x0 = int(np.floor(x))
x1 = min(x0 + 1, width - 1)
y0 = int(np.floor(y))
y1 = min(y0 + 1, height - 1)

q00 = image[y0][x0]
q01 = image[y1][x0]
q10 = image[y0][x1]
q11 = image[y1][x1]

y = q00+(x-x0)*(q10-q00)+(y-y0)*((q01+(x-x0)*(q11-q01))-(q00+(x-x0)*(q10-q00)))

return y
```

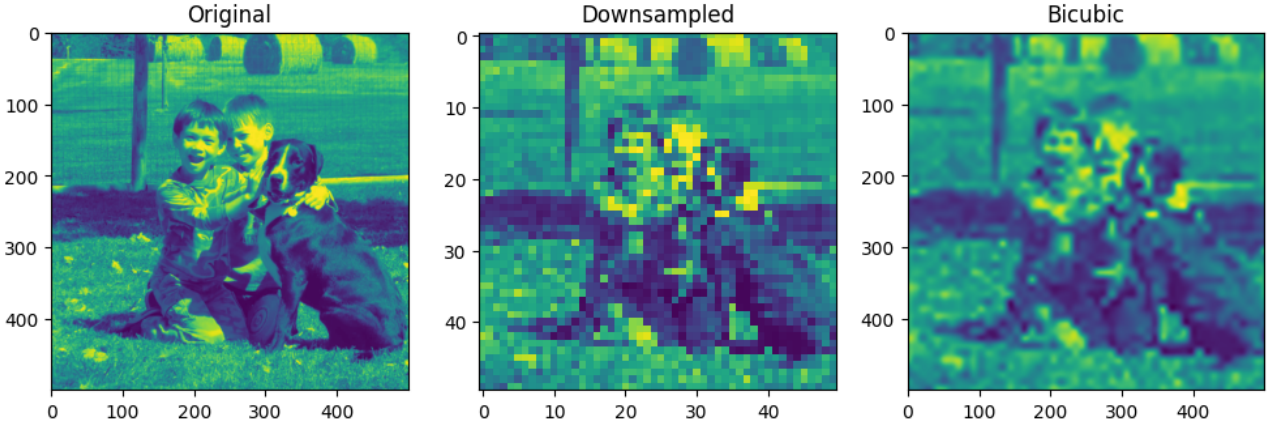
```
#Bicubic Interpolation
def interp_bicubic(image, new_x, new_y):
#Get the 4 neighbors:
x = new_x
y = new_y
y0 = int(np.floor(y))-1
y1 = int(np.floor(y))
y2 = int(np.floor(y))+1
y3 = int(np.floor(y))+2

points = [y0, y1, y2, y3]
image_result = []
#Interp with respect to x
# print(image.shape[0])
for point in points:
    if 0 <= point < image.shape[1]:
        image_result.append(interp_cubic(image[point, :], x))
    else:
        image_result.append(0)
#interp with respect to y:
return interp_cubic(image_result, y-y0)
#!! have to use y-y0 since y is the global y, but we need the new y after
#interpolating in the x direction, which is y-y0
```



2D Nearest Neighbor Interpolation



2D Bilinear Interpolation



2D Bicubic Interpolation

Figure 10: 2D Nearest Neighbor, Bilinear, and Bicubic Interpolation

Problem 1.2.3

Finally, Fourier Transform upsampling, also known as zero padding, was used. The frequency space of each interpolation method was analyzed, and Figure 11 shows it for nearest neighbor interpolation.

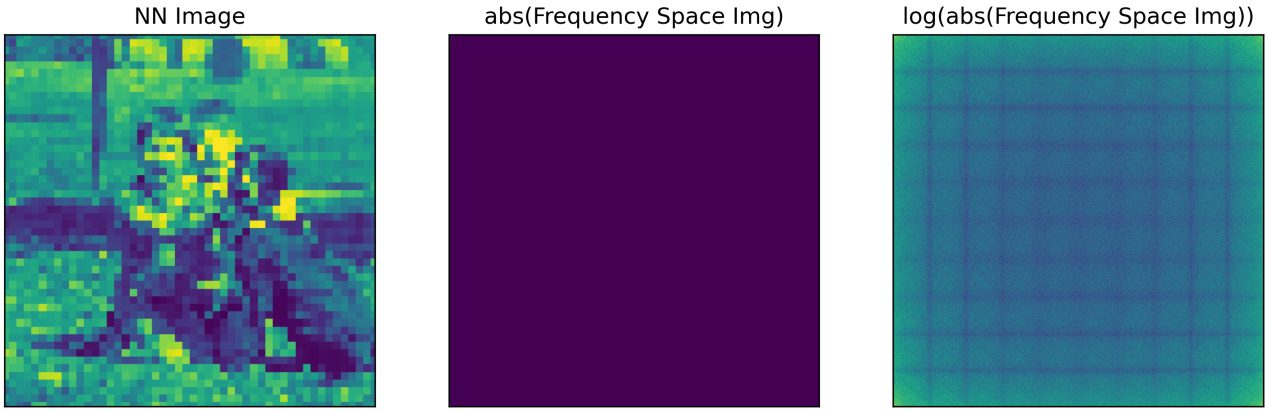


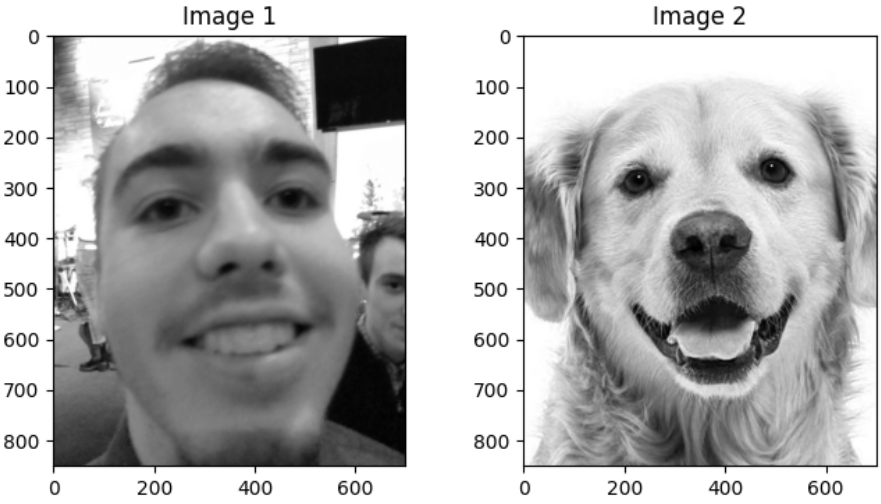
Figure 11: Nearest Neighbor Interpolation Frequency Space

Since the nearest neighbor algorithm interpolates with the closest point to the one in interest, it creates repeated blocks of the original images' pixels. This can clearly be seen in the log of the absolute value of the frequency space, in which the low-frequency spectrum creates a repeated grid-like structure.

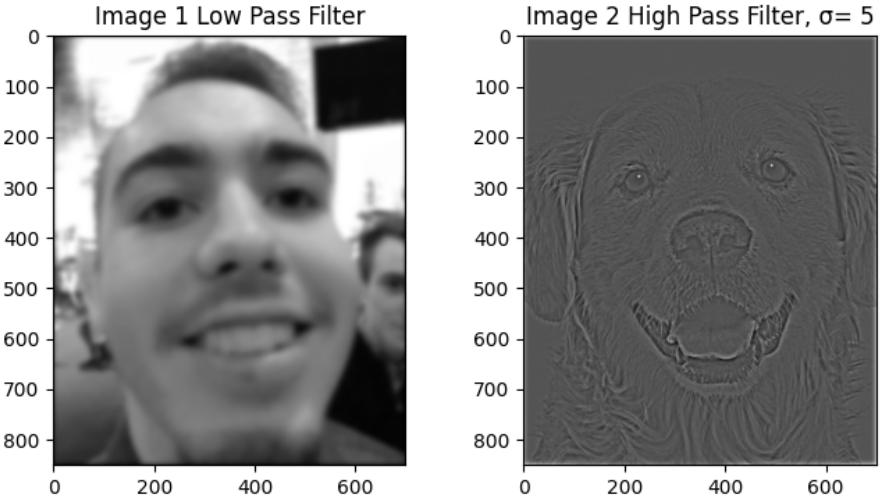
Nearest-neighbor interpolation upsampling of the image has more and stronger high frequency signals in its Fourier space, since the abrupt blocky transitions between pixels introduce rapid oscillations in the normal image space. On the other hand, bilinear interpolation smooths across pixels, so its Fourier space plot has weaker high-frequency terms with more lower/mid frequencies. This is why nearest neighbor upsampled edges look sharper and more jagged, while bilinear produces smoother, less sharp edges.

Problem 3

Problem 3 involves taking two images, applying a high-pass filter to one and a low-pass filter to the other, and combining the two filtered images to create a hybrid image. In this case, the image of the man (Caleb) was convolved with a low-pass filter and the image of the dog was convolved with a high-pass filter. Figure 12 shows that up close, the dog can be seen better, but moving further away, the man can be seen more clearly. Different σ values were used and balanced to try to create the best looking hybrid image. A Gaussian Pyramid was also created to see the two images more clearly (Figure 13).



Original Images A and B



Images with low-pass filter (Left) and high-pass filter (Right) applied

Hybrid Image

Figure 12: Original, Filtered, and Hybrid Image

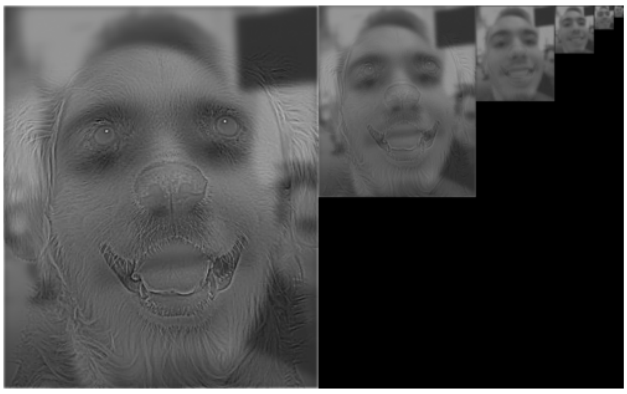


Figure 13: Gaussian Pyramid of the hybrid image