Project 1: Filters

Natalie Huante 03/13/2025

Grayscale Filter

Implementation

1. **Which RGB->Gray method do you prefer to use in your implementation? Why? Have you tried the other two methods? What are the visible differences you can tell from visualizations?**

   I chose the averaging method because it is faster to run, and I like how it doesn't return as bright an image as the Luma method. I decided not to use the lightness method because I feel it returns an image too dark and enhances some of the contrast too much for my liking, and I think it distorts the picture (as we can see in the examples below).

Results

Luma (1.4282 sec)

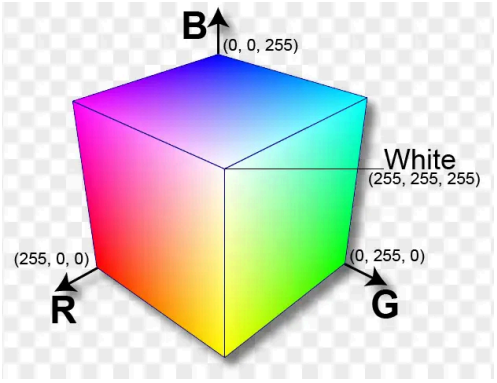
Average (1.0120 sec)

Lightness (2.1534 sec)

Invert Filter

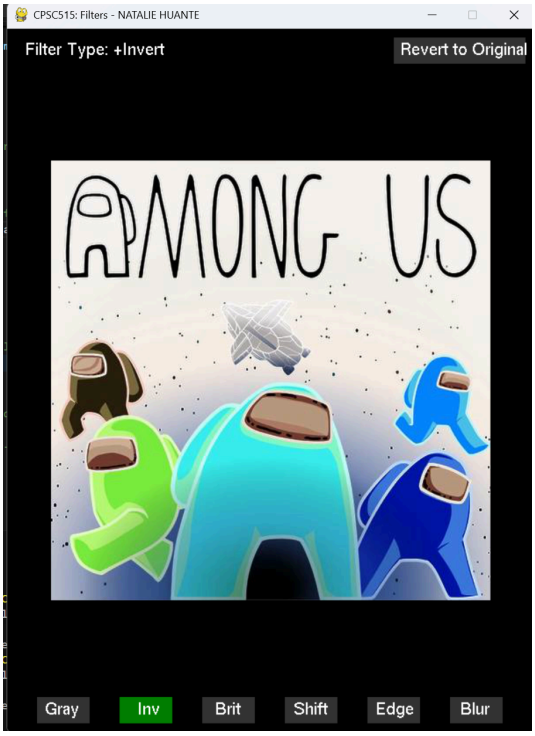
Implementation

1. **Why does applying the invert filter produce a "negative" effect on the image, causing reddish colors to appear bluish? Use your own words to explain the underlying color representation and transformation that leads to this effect.**



The inversion process is completed by taking every RGB value and subtracting it from the maximum value (which is 255). So, if the original R-value is 255 (the brightest red value), the new R-value will be on the other end of the red spectrum (no red) with a value of 0. So, if the pixel shows a bright red color, let's say its values are (255, 0, 0), which we can see on the graph on the right at the bottom-left corner of the cube, then the inversion process will result in the RGB value (0, 255, 255) which will be a bright cyan blue at the top-right corner of the cub. Therefore, reddish colors will be more blueish, and pink/purplish colors will be more greenish. You simply move to the opposite end of the color cube.

Results



Invert (0.3992 sec)

Original Image

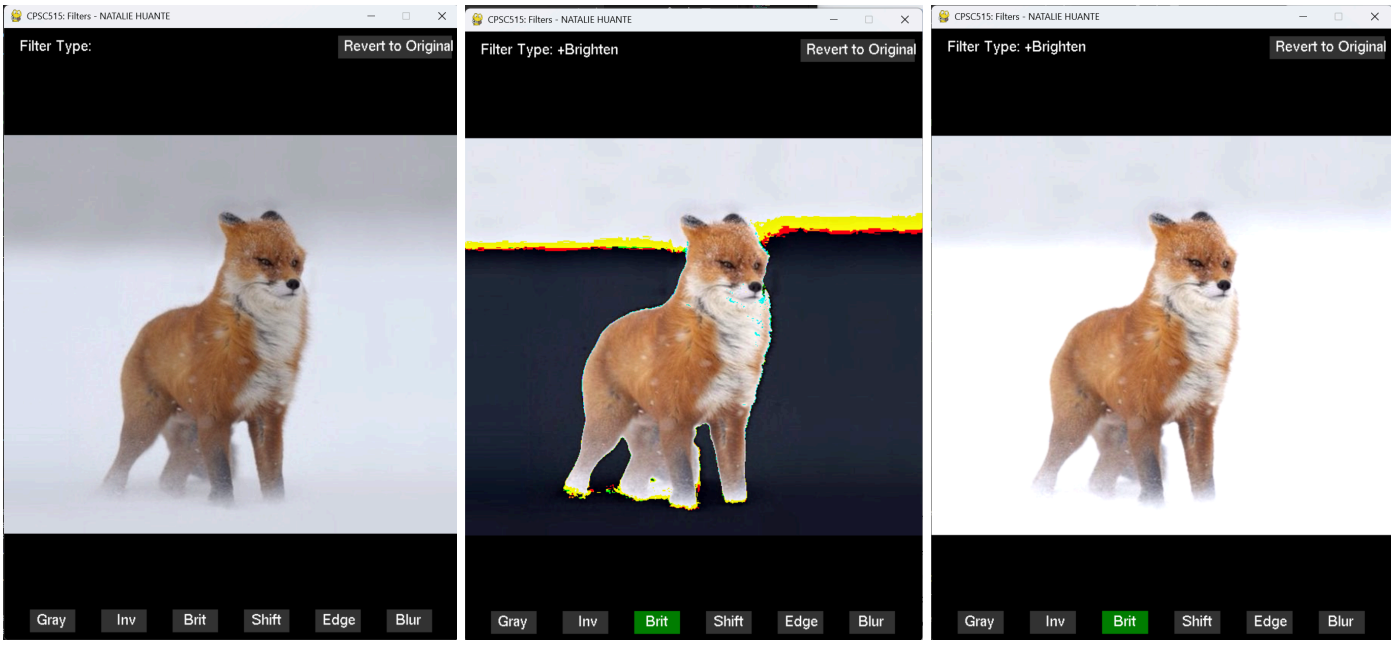
Brighten Filter

Implementation

1. **When simply increasing the intensity values of the image, we may end up with "weird" artifacts caused by integer overflow. Explain how you resolved the issue in your implementation.**

This filter returns "weird" values for our pixels when we do not handle integer overflow. This means that for colors that are close to the 255 max value, brightening them more will cause them to go past the 255 value and essentially "wrap around" to the lower values in the range. Therefore, if a pixel should be closer to white but overflows, it will be returned as a lower value and essentially one closer to black than white. This is what we see below. To fix this, we should check if this new value is greater than 255 when we calculate the brightened value. If it is, we cap it and simply return 255. This way, the pixels will never "wrap around" and simply stop at the brightest possible color, pure white (255, 255, 255).

Results



Original | Before Handling Overflow | After Handling Overflow

Shift Filter

Implementation

1. **Use your own words to explain your approach to creating a left-shift filter based on different pixel-shifting numbers.**

   1. When the shift kernel is created, its size depends on how many pixels we should shift to the left. All values in the kernel should be 0, and the only 1 value should be placed on the left-most element of the center row. We want each pixel of the resulting image matrix to depend entirely on the pixel n-columns to its right, where n is the number of pixels to shift.

For example, if we shift 1 pixel to the left. Each resulting pixel should get its value from the pixel to the left in the original image. If we shift 2 pixels to the left, the value is grabbed from the pixel 2 columns to the left in the original image, and so on.

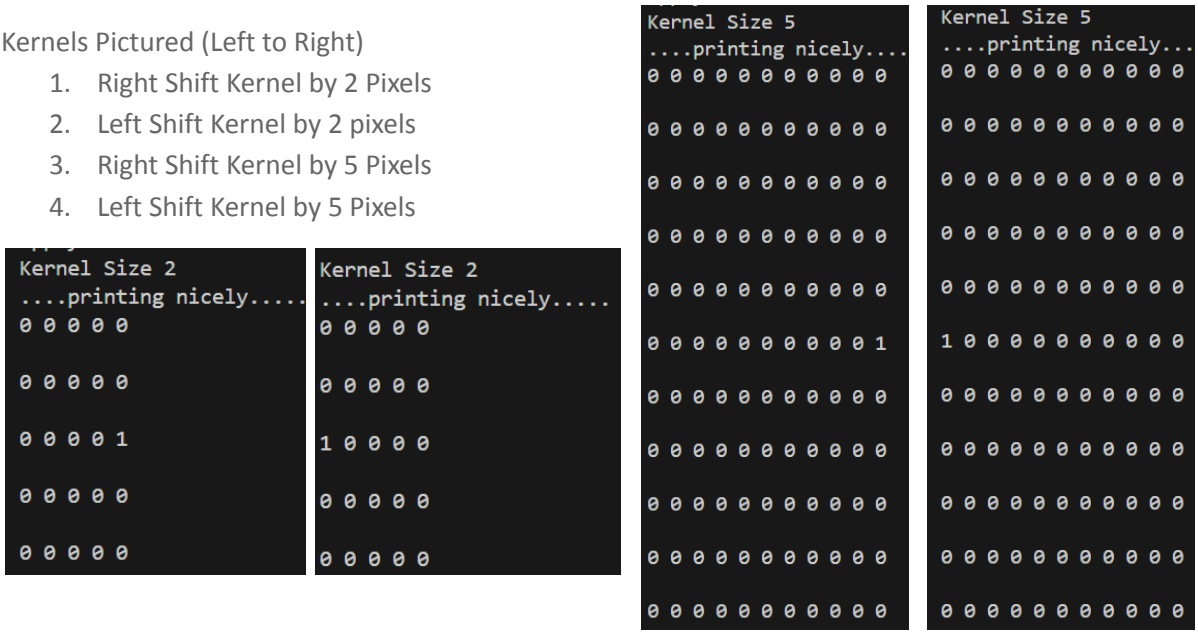
So, to create the correct kernel, we must have n rows on the left and right of the center pixel since our kernel must be odd and square (we assume this). We calculate this width of `2n + 1`, initialize a 2d array of size `2n+1 * 2n+1` with all elements as 0, and update the center row's left-most column to 1.

2. Apply the shift kernel we created to the image using 2d convolution.

2. **What do your left-shift filter kernels look like for shifting 2 and 5 pixels at a time?**

Kernels Pictured (Left to Right)

1. Right Shift Kernel by 2 Pixels
2. Left Shift Kernel by 2 pixels
3. Right Shift Kernel by 5 Pixels
4. Left Shift Kernel by 5 Pixels



3. **Have you noticed the performance dropdown when applying a shift kernel with 2D convolution? Use your own words to explain why it'd happen. Is there any performance difference in shifting 2 and 5 pixels at a time? How about the result in the shifted image? Explain why.**

When using 2D convolution, the image processing takes longer than the pixel-by-pixel calculations we were computing before. This is due to the significant increase in the number of calculations needed for the convolution.

Looking back at the inverse filter, we only needed to perform `width x height` subtractions where width and height refer to the image's dimensions. We then update each pixel as we do its subtraction. For 2D convolution, however, we now need to perform nine multiplications and eight additions for each pixel before we can update its value (assuming the kernel is 3x3). So, if the image is 10x10 pixels and the kernel is 3x3, the number of calculations for the inverse filter is 100, while the 2d convolution requires 17,000. As we can predict, this increases exponentially as the number of pixels

in our image increases or the size of the kernel increases. We see this reflected below in the time each filter takes to execute.

Using a 500x500 image (Capybara)

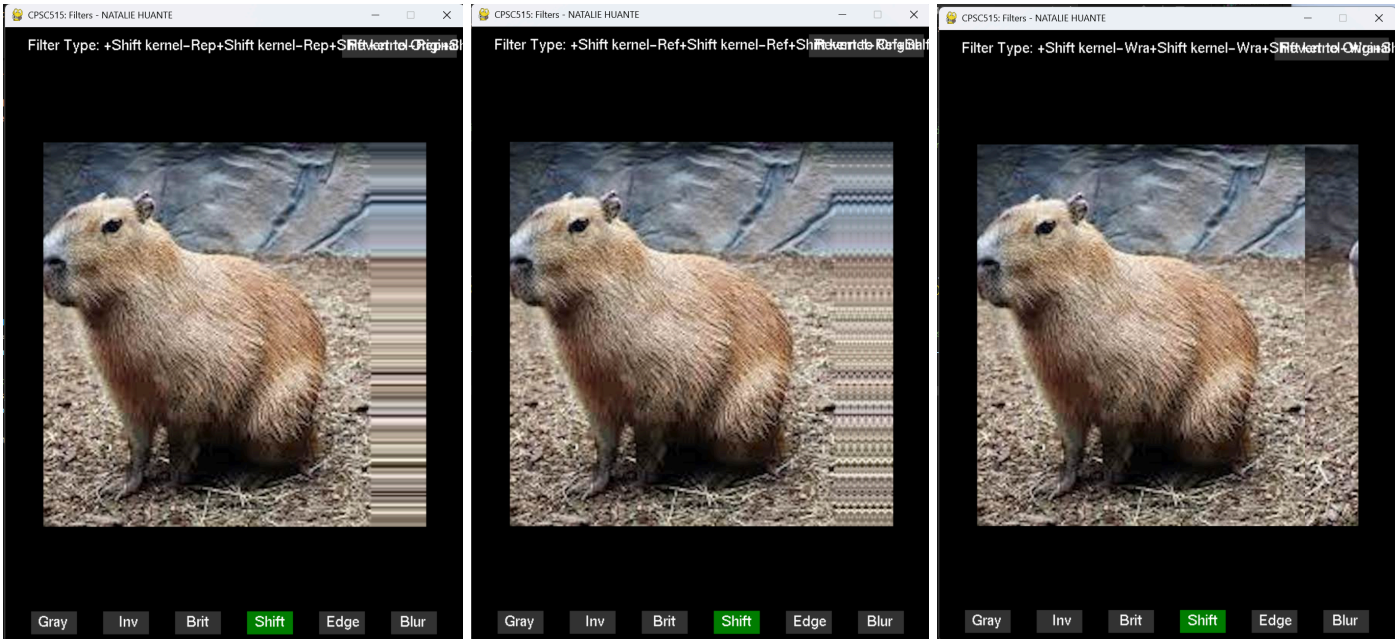
- Inverse Filter 0.4211 sec
- Left Shift 2 Pixels 2.2929 sec
- Left Shift 5 Pixels 3.1236 sec
- Left Shift 10 Pixels 6.2243 sec

Using a 450x671 image (Mona Lisa)

- Left Shift 10 Pixels 7.5902 sec

Results

All three outputs below are the result of 14 left-shift operations by 5 pixels in each shift.



Repeated Reflected Wrapped

Sobel Edge Detection

Implementation

1. **What are the original Sobel-x kernel and Sobel-y kernel, before flipping? Explain how to flip a 2d kernel by 180 degrees in implementation through indexing.**

The provided Sobel kernels have already been flipped, but the kernels before flipping would look like these matrices →

Sobel X Kernel	Sobel Y Kernel
[1 0 -1]	[1 2 1]
[2 0 -2]	[0 0 0]
[1 0 -1]	[-1 -2 -1]

However, if were were to plug these un-flipped kernels into the 2D convolution, we would have to flip by 180 degrees before applying to the image. I use indexing to handle this through the following steps:

1. Convert the row-major index to the corresponding row and column in the 2d kernel
2. Handle the horizontal flip when updating the row value.
    a. We start at the index of the right-most column `kernel_width - 1`
    b. Move one column to the left for every column away from column 0 the current element is at. This will result in elements originating from column 0 staying at the right-most column and elements originating from the right-most column moving left to column 0.
3. Handle the vertical flip when updating the column value.
    a. Repeat the same process as step 2, but start at the top column and move down. Elements from column 0 will stay the the top column, and elements from the top column will move down to column 0.
4. Using the new row and column values, convert to row-major form and store in our temp array.
5. Once iterating over kernel elements is done, copy over the temp array to the kernel array.

2. Use your own words (do not copy your code here) to explain your approach to respectively sliding a row kernel and column kernel on the input image.

We will use 1D convolution for edge detection, so slide the row kernel first and then the column kernel using the following steps. I do not include smaller steps, such as the conversions from row-major to 2d index, but rather the more essential steps.

Sliding The Row Kernel

1. Create a result array of the size of the original image
2. For each pixel in the original image, we will calculate a sum by doing the following:
    a. For each value in the kernel, calculate its distance from the center element in the kernel. The center element has a distance of 0. The element to the immediate left of the center -1. The element to the immediate right of the center +1. And so on. To get the corresponding pixel in the original image to the current value within the kernel we are on, we use our pixelEdgeMethods. We pass in the current row we are iterating on since we are sliding horizontally, and the current column + distance calculated. Therefore, the center element will correspond to the target pixel, while the value in the kernel to the immediate left of the center will correspond to the pixel in the same row but one column less than the center (one to the left). Given the corresponding pixel, we multiply the weight by the pixel's RGB values and add it to the sum. Once the sum has accounted for each kernel element, assign the new RGB values to the result array (clamping between 0-255 when necessary).

Sliding the Column Kernel

1. Repeat the same process as described above for the column kernel with the following differences:
    a. To find the corresponding pixel from the original image to the current kernel element, we pass the current row + distance and the current column into the pixelEdgeMethods. We are now sliding the kernel vertically so the kernels will always align with the column of the target pixel. The corresponding pixels will be +/- d rows above or below the target pixel, where d is the distance of the current kernel element to the kernel center.

b. Instead of pulling corresponding pixels from the original image, we pull them from the result array of the row convolution. So, we do the row convolution on the original image → pass it on, and do the column convolution to this processed image → output of column convolution is the final result.

3. **Use your own words to explain how you combine Gx and Gy and obtain the final gradient for each pixel, how you apply the sensibility value with the gradient number, and how you make sure each channel value is still within [0,255].**

To combine the Gx and Gy matrices, I use two numpy methods first to add the two matrices together element by element `np.add()` and then take the square root of the summed values, again element by element `np.sqrt()`. This will give us the initial combination, resulting in the matrix G.

Then, I multiply each value in G by the determined sensitivity value and make sure that values less than 0 are updated to 0 and values greater than 255 are updated to 255 using `np.clip()`. This gives us the final G matrix.

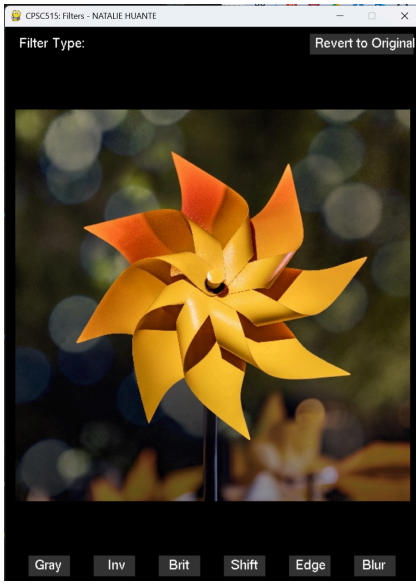
4. **Explain the purpose of the sensitivity parameter and how to interpret the impacts of changing its value on the gradients.**

The sensitivity parameter affects how many edges are detected in the resulting image. The values in G represent the gradient calculated at each pixel. So, if we decrease these values by a specific rate, we will have lower gradients assigned to each pixel. Since this is done before we compare to the threshold, a sensitivity of less than 1 will ultimately result in fewer pixels being greater than the threshold and identified as "edges" compared to leaving it at 1. In the resulting image, we will see only the more prominent edges represented by white pixels, the lower the sensitivity is. This makes sense conceptually because if we want to be less sensitive to edges, we should see fewer edges detected.

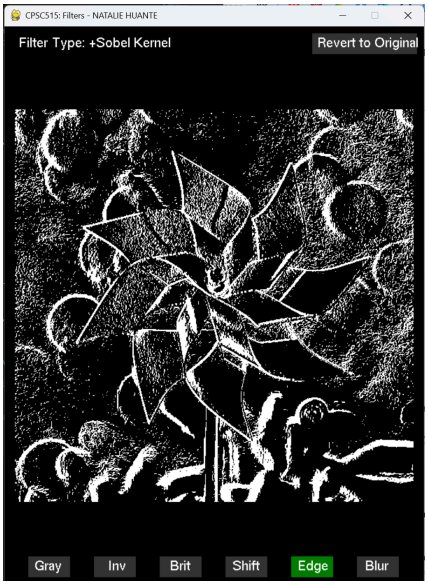
5. **How do you determine the threshold for detecting edges? What are your criteria for identifying a "good" threshold? What is your threshold?**

Based on my research online, I include a default calculation for the threshold based on the values of G. This calculation only occurs if no threshold is specified when the `edgeDetection()` function is called. However, having experimented with multiple thresholds, I found my preferred value to be 20. Higher thresholds such as 100, 50, and 25 don't include all the more prominent edges of the windmill, so I moved to lower values. I found values less than 20 to not add any more prominent edges. Instead, they allow for less prominent edges in the final image, which I associate with noise. Therefore, I use 20 as my threshold.

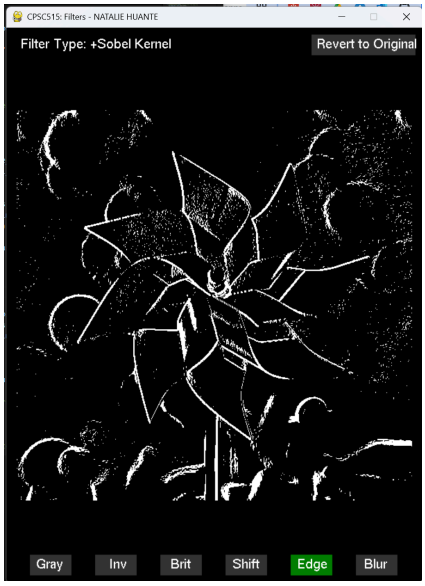
Results



Original Image

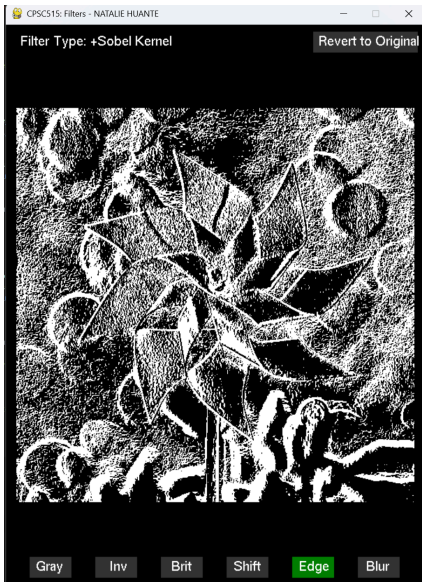


Threshold=15, Sensitivity=1



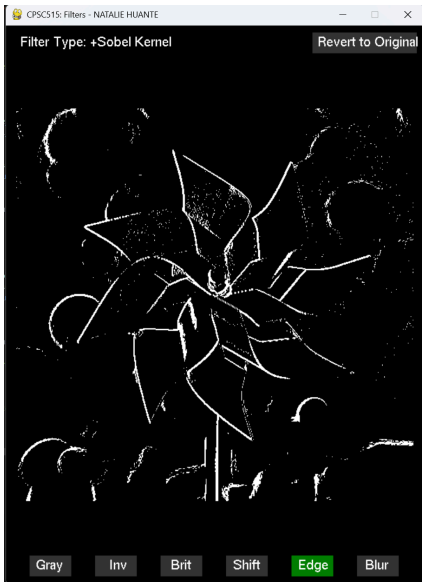
Threshold=15, Sensitivity=0.5

As shown above, a lower sensitivity will decrease the values in the G matrix and result in fewer detected edges in the final image. For this reason, a lower sensitivity will result in lower sensitivity to including less-prominent edges.



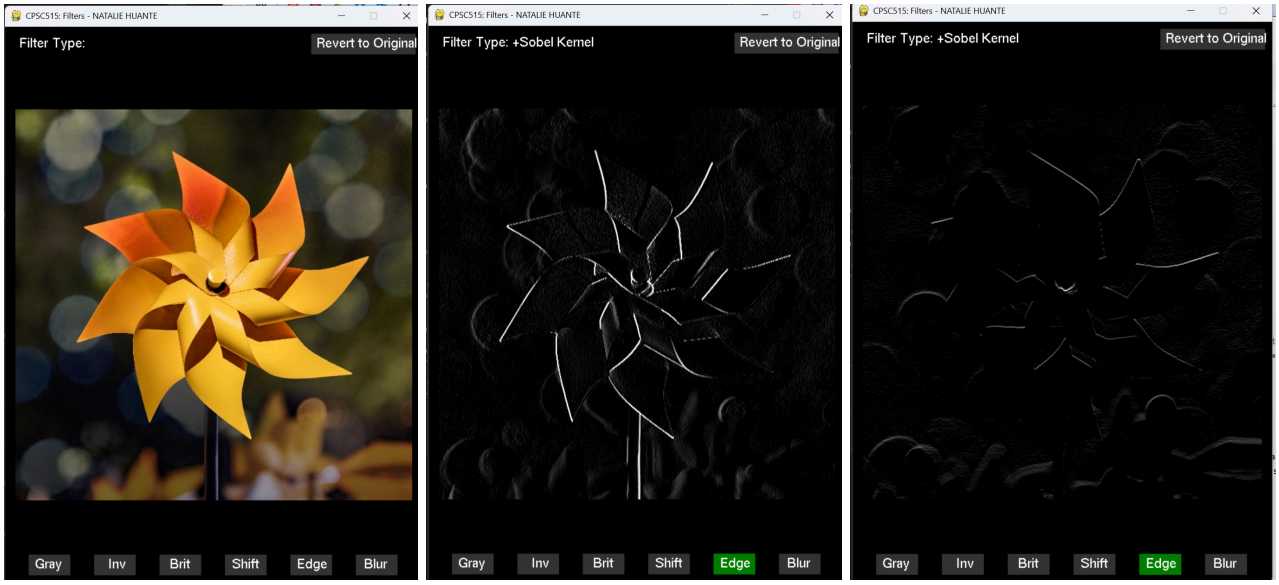
Threshold=5, Sensitivity=0.75

Threshold=20, Sensitivity=0.75



Threshold=30, Sensitivity=0.75

I chose to use threshold 20 since we can see the obvious edges of the windmill, and it isn't inclusive of the noise that lower thresholds do (as we can see in the left example above).

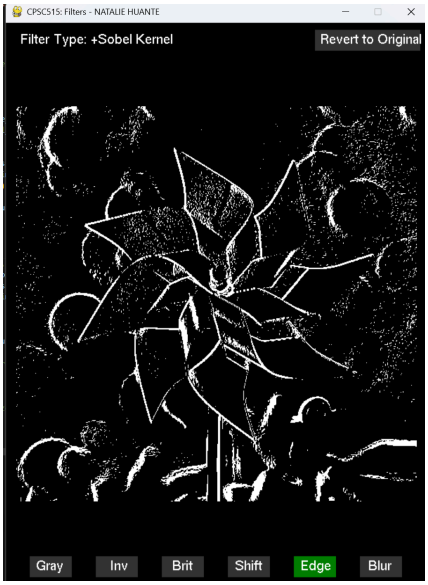


Original Image | Sobel-X Filter Only | Sobel-Y Filter Only

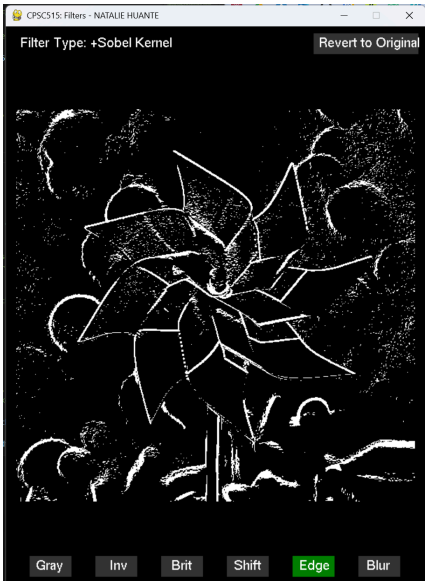
As we can see from the images above, the output of the Gx matrix alone identifies the vertical edges in our image. In contrast, the Gy matrix alone identifies the horizontal edges in our image. This makes sense because if we look at the gradients in the x-direction, any significant fluctuations from left to right will indicate a vertical edge. Likewise, the gradients in the y-direction will identify any considerable fluctuation from top to bottom, meaning there is a horizontal edge.

In response to our class discussion about applying kernels in different orders, I add the following section.

In the four images below, I test out the four combinations of applying the row kernels and column kernels for both Gx and Gy in the edge detection process. Although the results are similar, we can see a difference in the output in response to whether the row or column kernels were applied.

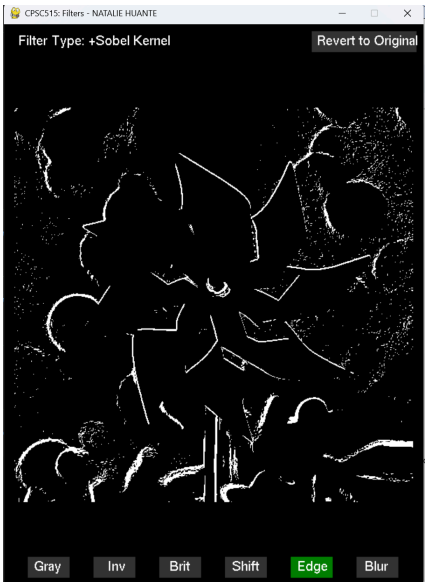


Gx, Gy = Row ⟶ Col (my approach)



Gx, Gy = Col ⟶ Row

Gx=Row ⟶ Col, Gy=Col⟶Row



Gx=Col ⟶ Row, Gy=Row⟶Col

Based on the above results, my approach of applying the row kernel before the column kernel for both Gx and Gy is one of the better combinations, so I leave it as is.

Triangle Blur

Implementation

1. **Use your own words to explain how you derive a triangle kernel of an arbitrary kernel size (odd number). What is your formula for calculating the weight at each index in a triangle kernel?**

To create a triangle kernel given its size (assuming it's odd), I do the following steps:

1. Initialize a 1d array of specified kernel size
2. Get the index of the center element with `kernel_size // 2`. For a kernel size of 3, this evaluates to 1.
3. For each element in the kernel, calculate its weight by…
    a. Calculating the denominator as the number of elements on either side of the center pixel plus the center pixel `1 + kernel_size // 2`. Then, square that value. This will give us the sum of the numerators for each element in the kernel. For example, in a kernel size of 3, the denominator will be `(3 // 2) + 1 → 2 → 2^2 = 4.`
    b. Calculate the numerator by first getting the distance from the current kernel element to the end of the kernel and then adding 1. So, for a kernel size of 3 and at the kernel element at index 0 (left of the center), its distance will be 0 (it's at the edge already). The numerator will then be its distance plus one. Here, the numerator is 1. (The goal of this process is to give the edge elements a numerator of 1 and to increase the numerators by 1 with each step you take toward the center. The center element should have the highest numerator of all the elements.)
    c. Set the value for that element as the numerator divided by the denominator.

2. **What does your triangle kernel look like for kernel sizes 5 and 9? Can you verify your result by deriving each triangle kernel by convolving two corresponding box kernels?**

```
Apply Image Blur: Start...
--Triangle Kernel = [0.1111111111111111, 0.2222222222222222, 0.3333333333333333, 0.2222222222222222, 0.1111111111111111]
--Corresponding Box Kernel
[ 0.01 0.02 0.04 0.02 0.01 ]
[ 0.02 0.05 0.07 0.05 0.02 ]
[ 0.04 0.07 0.11 0.07 0.04 ]
[ 0.02 0.05 0.07 0.05 0.02 ]
[ 0.01 0.02 0.04 0.02 0.01 ]
```

Kernel Generated of Size 5

```
Apply Image Blur: Start...
--Triangle Kernel = [0.04, 0.08, 0.12, 0.16, 0.2, 0.16, 0.12, 0.08, 0.04]
--Corresponding Box Kernel
[ 0.00 0.00 0.00 0.01 0.01 0.01 0.00 0.00 0.00 ]
[ 0.00 0.01 0.01 0.01 0.02 0.01 0.01 0.01 0.00 ]
[ 0.00 0.01 0.01 0.02 0.02 0.02 0.01 0.01 0.00 ]
[ 0.01 0.01 0.02 0.03 0.03 0.03 0.02 0.01 0.01 ]
[ 0.01 0.02 0.02 0.03 0.04 0.03 0.02 0.02 0.01 ]
[ 0.01 0.01 0.02 0.03 0.03 0.03 0.02 0.01 0.01 ]
[ 0.00 0.01 0.01 0.02 0.02 0.02 0.01 0.01 0.00 ]
[ 0.00 0.01 0.01 0.01 0.02 0.01 0.01 0.01 0.00 ]
[ 0.00 0.00 0.00 0.01 0.01 0.01 0.00 0.00 0.00 ]
```

Kernel Generated of Size 9

To make the comparisons easier, I highlight the center rows and columns in the corresponding box kernels for each kernel generated above.

3. **Use your own words to explain how you apply a triangle filter using 1D convolution.**

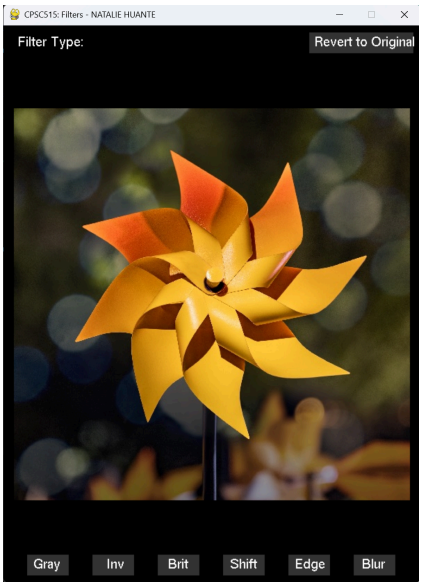
In this description, we are applying the row kernel first and the column kernel second. To apply the triangle filter, I will iterate over each pixel in the original image and store its new value in a temporary result array. To calculate its new value, I find the weighted sum (adding the product of each element in the triangle kernel by the corresponding pixel in the original image). For a triangle kernel of size 3, each pixel's new value will result from the weighted sum of 3 pixels from the original image. This iteration happens row by row, so row 0 (left to right), then row 1, and so on.

Once the row kernel has been applied to each pixel, we apply the column kernel. With the resulting image from the row convolution, we do the same process with the following differences: the corresponding pixels are grabbed from the temporary result rather than the original image, and we iterate through the image column by column (from bottom to top). The resulting array will be the final processed image.

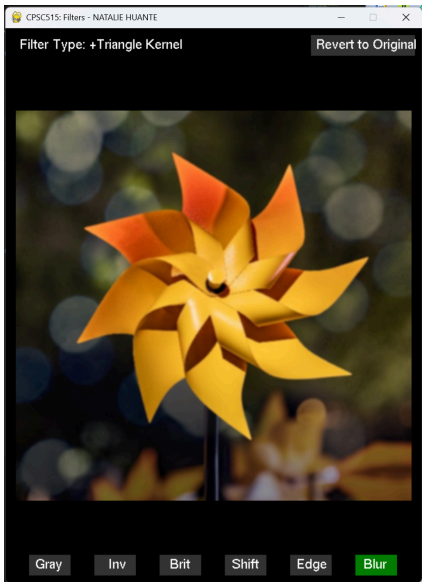
A few other notes on this are as follows:

- When grabbing corresponding pixels to kernel elements in the row convolution, we should access them from the same row as our target pixel but from columns to the left and right of the target pixel. However, the column convolution is different. We should access corresponding pixels from the same column but from rows above or below the target pixel. The kernels are 1D, so they only extend in one direction while the other stays the same.
- After each convolution and as we update the new pixel values of the result array, we clamp the new values to be within the [0,255] range. Values less than 0 will be assigned 0, and values greater than 255 will be assigned 255.

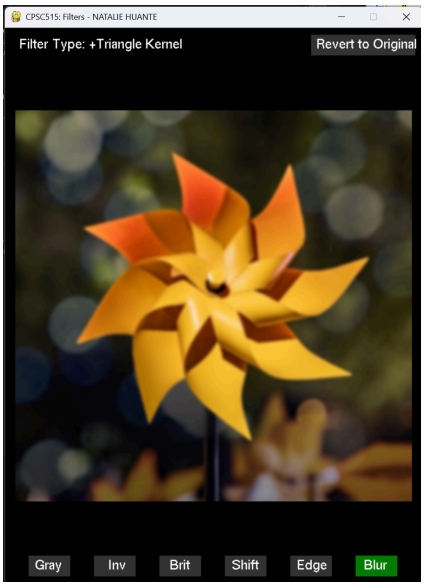
Results



Original



Kernel Size = 5 (14.4052 sec)



Kernel Size = 9 (25.1536 sec)

As shown above, the larger kernel size produces a blurrier image and takes longer to perform. This makes sense because we must handle more calculations for a larger kernel size. For a single pixel, a kernel size of 5 means we must compute the weighted sum of 25 kernel elements, while a kernel size of 9 increases that to 81 kernel elements. This same increase in kernel elements also accounts for the increased blurriness, as each pixel is affected by an even larger neighborhood (its own value has less of an effect on its new value).

Gaussian Blur

Implementation

1. **Use your own words to explain how you derive a Gaussian kernel based on an arbitrary blur radius. Specifically, how do you use blur radius to derive standard deviation (σ) and kernel size? How do you derive weights in the 1D Gaussian kernel based on those? Provide reliable references to support your decision.**

Given an arbitrary blur radius, I compute a Gaussian kernel through the following steps:

1. Compute the standard deviation and kernel size based on the blur radius.
    a. Per the instructions and my research online, the blur radius is typically set to three times the standard deviation from the average (or the center), so I calculate the sigma value as the blur radius divided by three.
    b. So, if the radius is three times the standard deviation, we can multiply the radius by two to find the sum of both sides of the kernel. We add one to this to account for the center pixel (assuming we only have odd-sized kernels). So, the kernel size will equal six times the standard deviation (sigma) plus an additional unit.
    c. It is important to note that if we had an infinitely large kernel, the weight of elements as we go further away from the center pixel would be closer to zero but never reach zero. This happens due to the nature of how we calculate the weights. Naturally, the question arises of where or at how many elements it is best to truncate the kernel. Given my research online, the recommended size is what I describe in parts (a) and (b). I include an example below from a *Digital Imaging Processing* textbook that shows why this is the case. Essentially, using a kernel size larger than `(6*sigma)+1` produces a similar blur and does not bring any benefits compared to using a kernel size of `(6*sigma)+1`. If anything, it is disadvantageous to do so, as a larger kernel size has a higher time and space complexity.

a b c

FIGURE 3.37 (a) Result of filtering Fig. 3.36(a) using a Gaussian kernels of size 43×43, with $\sigma = 7$. (b) Result of using a kernel of 85×85, with the same value of σ. (c) Difference image.

As we can see, given a sigma value of 7, a kernel size of 43 outputs the same result as a kernel size of 85. Opting for the smaller kernel gives our processing program better performance, so we should choose the smaller kernel.

2. Calculate the weights of the kernel elements.
    a. The center pixel should be 1, and the weight should decrease exponentially as we move further away from the center. I use the following formula to calculate the weight where x is the distance from the center of the kernel and sigma is the standard deviation:

$$Weight(x, sigma) = e^{-\frac{x^2}{2*sigma^2}}$$

    b. As I calculate the weight of each element, I keep a running sum of all the weights. Once all weights are calculated, I normalize the kernel by dividing each weight by the sum. This will result in the final Gaussian kernel.

3. Run the convolve1D function we created previously and pass in the Gaussian kernel generated as both the row kernel and column kernel. The resulting image is our output.

2. What do the kernels look like for Blur radii of 2 and 5? What are the corresponding standard deviation values (σ)? How do you interpret these values relating to image blur?

```
Apply Image Blur: Start...
-blur_radius = 2
-sigma = 0.6666666666666666
-kernel_side = 4.0
-kernel_size = 9.0
--Gaussian Kernel = [9.11100239e-09 2.39681882e-05 6.64571429e-03 1.94216240e-01
 5.98228136e-01 1.94216240e-01 6.64571429e-03 2.39681882e-05
 9.11100239e-09]
```

At Blur Radius of 2, Sigma = ⅔, Kernel Size = 9

```
Apply Image Blur: Start...
-blur_radius = 5
-sigma = 1.6666666666666667
-kernel_side = 10.0
-kernel_size = 21.0
--Gaussian Kernel = [3.64552971e-09 1.11441711e-07 2.37677946e-06 3.53658407e-05
 3.67141158e-04 2.65910905e-03 1.34367182e-02 4.73700950e-02
 1.16511633e-01 1.99934762e-01 2.39365368e-01 1.99934762e-01
 1.16511633e-01 4.73700950e-02 1.34367182e-02 2.65910905e-03
 3.67141158e-04 3.53658407e-05 2.37677946e-06 1.11441711e-07
 3.64552971e-09]
```

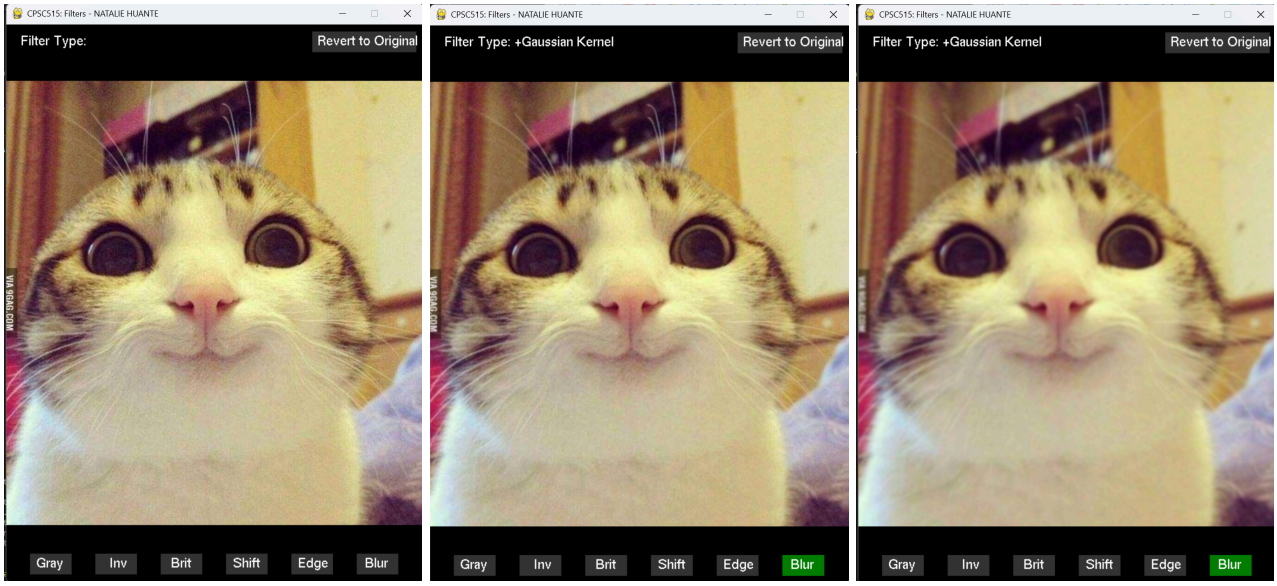
At Blur Radius of 5, Sigma = 1 ⅔, Kernel Size = 21

For greater blur radii, the generated Gaussian kernel will be greater in size and, therefore, weigh in more neighboring pixels for the new value of the current target pixel. This means the target pixel's old value contributes less to its new value, and the blur will be "distributed" more, so we get a greater blur effect on the resulting image.

3. **Use your own words to explain how you implement Gaussian blur by applying 1D Gaussian kernels through 1D convolution.**

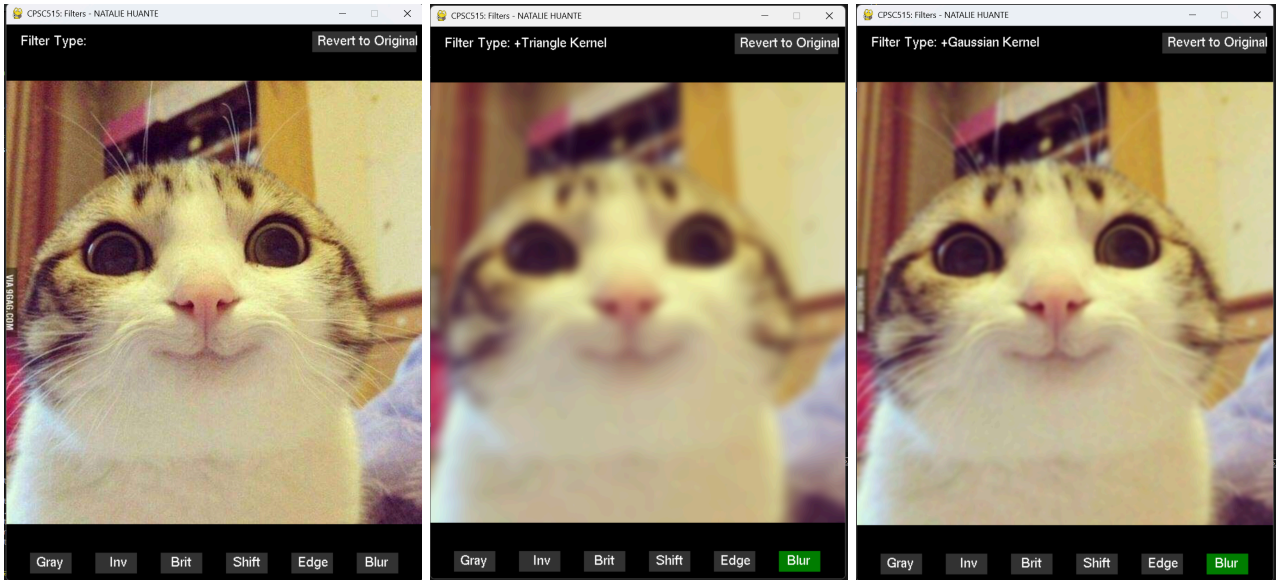
Like the triangle blur, once the kernel has been generated, we apply it using 1D convolution using the `convolve1D()` function we created in previous steps. We simply pass in the same 1D Gaussian kernel twice.

Results



Original Image | Blur Radius = 2 (11.0215 sec) | Blur Radius = 5 (23.9939 sec)

A greater blur radius performs a slower Gaussian blurring process than otherwise since having a larger kernel means more kernel elements to weigh and add when calculating the new pixel values. As described previously, the larger kernel will also distribute the blurring effect more and consider more “neighbors” in the weighted sum of each target pixel, resulting in a blurrier image.



Original Image Triangle Kernel Size=29 (118.1204 sec) Gaussian Kernel Size=29 (44.6056 sec)

Above, I compare the results of a triangle and a Gaussian kernel of the same size of 29 elements. The triangle kernel creates a blurrier result than the Gaussian kernel and takes much longer to perform the computation. The blurrier effect for the triangle kernel is due to the more even distribution of weights across the kernel compared to the exponential difference in the latter. The triangle blur in this example took more than two and half the processing time of the Gaussian blur.

References:

- np.uint8 conversion
- numpy array function
- YouTube Video: Separatable Kernels & Separable Convolutions
- Online Image Edge Detection Processor (for comparing to my output)
- OpenCV Documentation
- OpenCV Forum: Gaussian Kernels
- Wikipedia Gaussian Blur Process
- StackOverflow - How to Detect Kernel Size from Blur Radius
- Stack Overflow - Gaussian Blur Relationship Between Standard Deviation Radius and Kernel Size
- NVIDIA Documentation - Chapter 40 Incremental Computation Gaussian
- Science Direct - Gaussian Blur
- OpenCV Documentation - Smoothing and Blurring
- HIPR - Gaussian Smoothing Process
- Wikipedia - Gaussian Filter Digital Implementation
- Digital Image Processing by Rafael C. Gonzalez and Richard E. Woods
- WordPress - Gaussian Filter Relation Between Standard Deviation and Filter Size Radius in Pixels