CS 180: Project 2

Fun with Filters and Frequencies!

Meenakshi Mittal

Overview:

In this project, we explore some methods of algorithmic image filtering. We will try to detect edges using derivatives and gradients, simulate high-pass and low-pass filtering, and blend images together using Laplacian stacks.

Part 1: Fun with Filters:

Part 1.1: Finite Difference Operator:

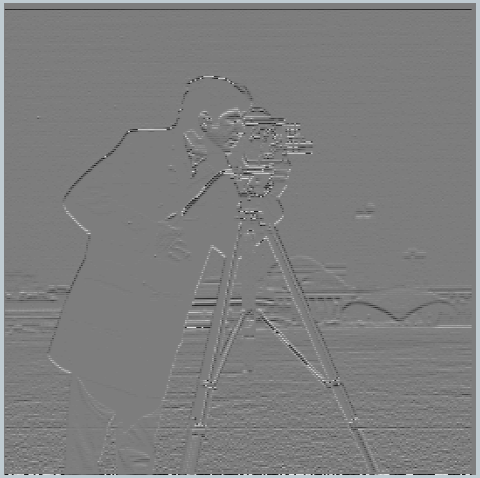
We can take the x and y partial derivatives of an image by convolving it with the finite difference operators. The operators look like this:

$$\mathbf{D_x} = \begin{bmatrix} 1 & -1 \end{bmatrix}, \mathbf{D_y} = \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

Note that 0.5 is added to all images for the purpose of displaying them, since many of the convolutions return negative values. Here, we take a look at the provided image of a cameraman and its convolutions with these operators:

Original

Partial Derivative in x



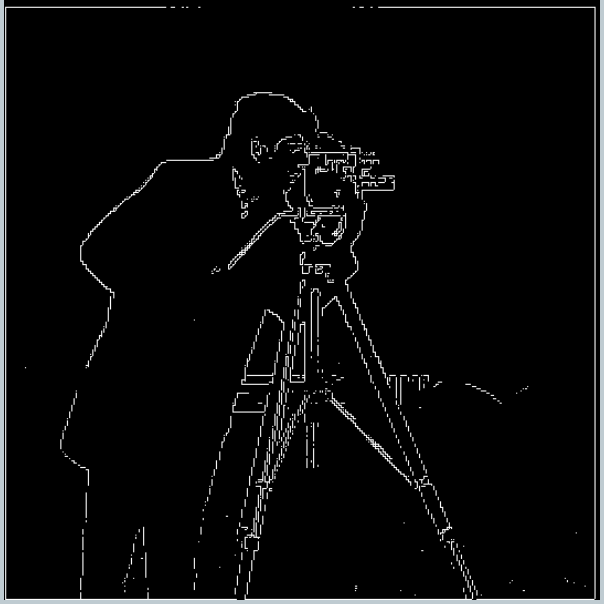
Partial Derivative in y

The partial derivative images appear noisy. We will address this in a bit.

We can also find the gradient magnitude of the image. To do this, we simply take the square root of the sum of the squares of the partial derivative images.

Below, we have the raw gradient magnitude image of the cameraman, computed as described. Alongside it is a binarized gradient magnitude image, using a threshold of 0.3. Any pixel with a higher brightness than 0.3 was converted to 1 (white), and anything lower was converted to 0 (black). This threshold was determined qualitatively, attempting to suppress as much noise as possible while still maintaining the image's edge details.

Raw Gradient Magnitude



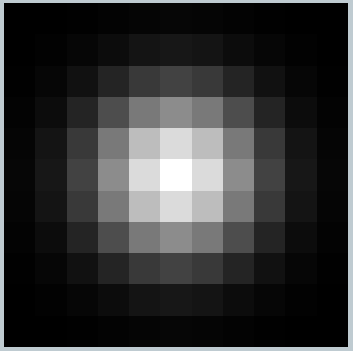
Binarized Gradient Magnitude

Part 1.2: Derivative of Gaussian (DoG) Filter:

The partial derivatives we displayed above look quite noisy. How can we fix this?

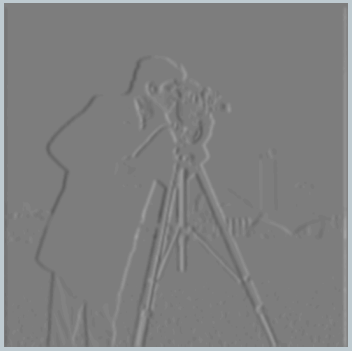
One option is to convolve the image with a Gaussian filter before proceeding with our derivative convolution. The intention is to remove some of the higher frequency features in the image that could potentially create noise after the convolution.

Below, we apply a Gaussian filter (with kernel size = 13 and sigma = 13/6) to the cameraman image, and then convolve it with the same partial derivative operators.

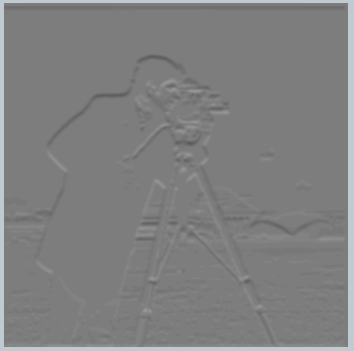


Gaussian Filter

Original with Gaussian Blur



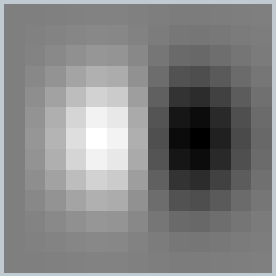
Partial Derivative in x

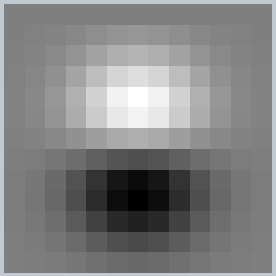


Partial Derivative in y

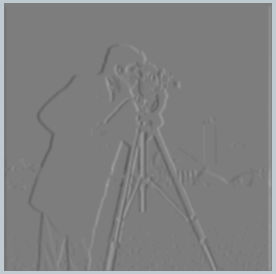
This looks much better! The noise we saw in the edges of the original partial derivative images appears to be entirely gone. The images almost look like they are embossed and appear to pop out of the page a bit. Now it is very clear to see the "positive" and "negative" partial derivatives within the image. Wherever it is dark, the image is transitioning from a brighter color to a darker color, and where it is light, the image is transitioning from dark to bright (when looked at from left to right or up to down).

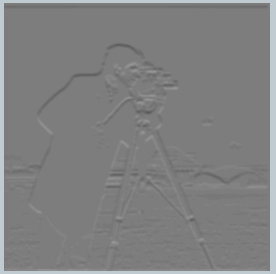
We will take this one step further by reducing the number of image convolutions from 2 to 1. We can do this by first convolving the Gaussian filter with the finite difference operators (a small and efficient calculation), and then our image with this new filter. The results of this are shown below:


Gaussian Filter + Dx


Gaussian Filter + Dy


Original


Partial Derivative in x


Partial Derivative in y

We get the same results as before, with a more efficient calculation. Yay!

Part 2: Fun with Frequencies!:

Part 2.1: Image "Sharpening"

We can also use the Gaussian filter to sharpen images. When we apply the Gaussian filter to an image, we retain the image's lower frequencies. If we take a Gaussian filtered and subtract it from the original, we are left with the image's higher frequencies. We can then simply take these higher frequencies and add some multiple of them back to our original image. This achieves the effect of image sharpening that we often see in photo editing applications.

Instead of computing a series of convolutions and image operations, we can condense this process into a single convolution called the "unsharp mask filter". If e is the identity filter, g is the Gaussian filter, and a is some multiplicative constant, the unsharp mask filter looks like this: ((1+a)e - ag). Here are some examples:

Original

Sharpened (Gaussian kernel size = 3, a = 5)

Original

Gaussian Blurred

Re-sharpened

Original

Gaussian Blurred

Re-sharpened

Original

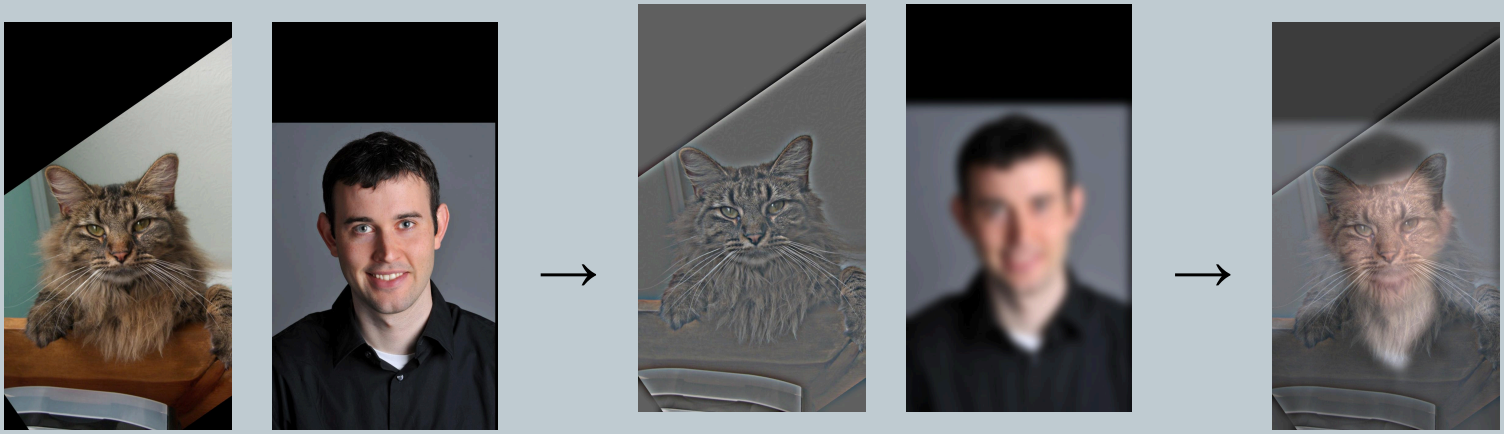
Gaussian Blurred

Re-sharpened

Part 2.2: Hybrid Images:

Now let's have some fun. We will create hybrid images using an approach described in a SIGGRAPH 2006 paper by Oliva, Torralba, and Schyns. The intention of these images is to appear differently when viewed up close versus from afar. This is achieved by overlaying the high-frequency features of one image and the low-frequency features of another image. To simulate the extraction of the low-frequency features, we are using our old friend the Gaussian filter. To simulate the extraction of the high-frequency features, we are simply subtracting the low-frequency features from the original image. The provided image alignment code was used to align and resize all of the following images.

Take a look at Derek morphed with his cat Nutmeg. Look at it up close, and you'll primarily see Nutmeg. Zoom out or squint, and you'll only see Derek:

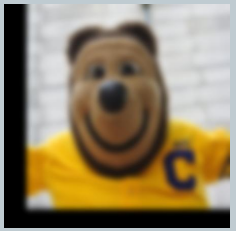
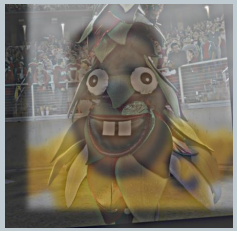


Aligned Nutmeg | Aligned Derek | High Frequency Nutmeg | Low Frequency Derek | Nutmeg-Derek

The optimal Gaussian kernel sizes was determined as 85 for both images.

Let's look at another one. From up close, we can only see the winner of the world's ugliest mascot award. But zoom out a bit, and you'll start to notice a friendly old face:

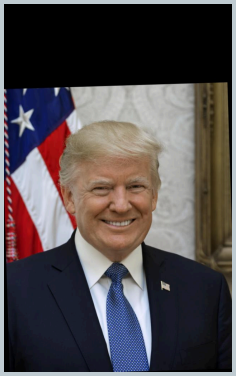
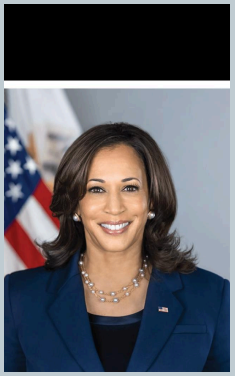


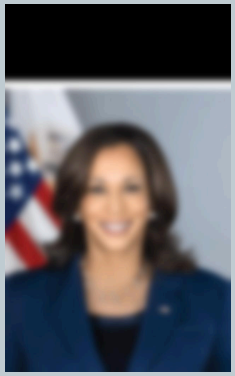




Aligned Tree | Aligned Oski | High Frequency Tree | Low Frequency Oski | Osktree

This one unfortunately doesn't look the best in my opinion. I think Oski's face and the tree are just too different from each other, in terms of shape and color (and quality of the schools they represent...). The best kernel sizes were determined to be 45 for the tree and 35 for Oski.

Moving on. In case anyone wasn't aware, tensions are pretty high in American politics at the moment. The presidential election is less than 2 months away! While the 2 candidates have very different views from one another, perhaps we can help them reach a middle ground...

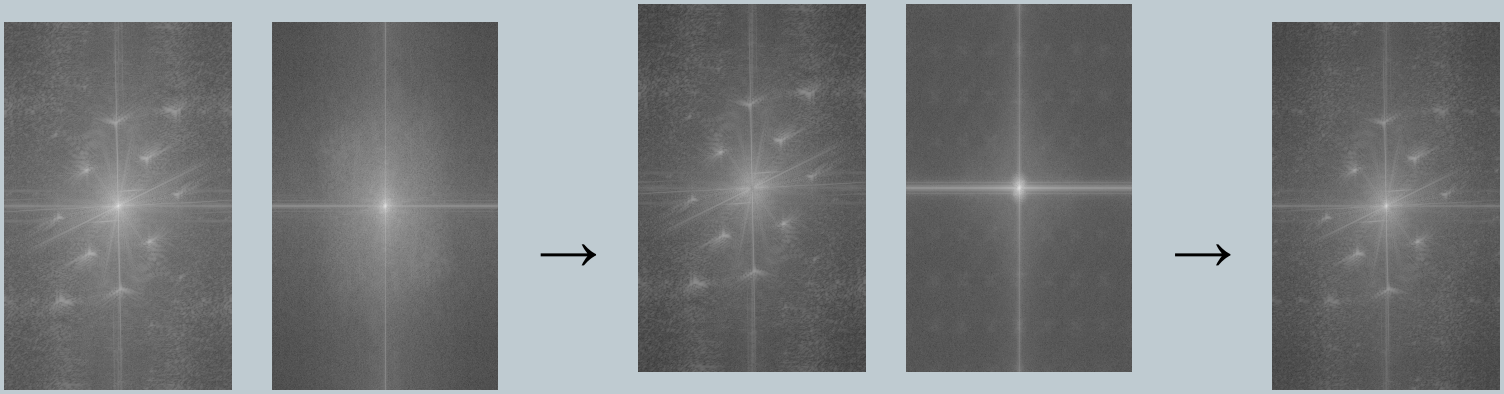






Aligned Trump | Aligned Harris | High Frequency Trump | Low Frequency Harris | Harrump

This one looks pretty good! The optimal Gaussian kernel sizes were determined to be 65 for Donald Trump and 55 for Kamala Harris.

Let's conduct a frequency analysis for this one. We first convert the images to grayscale, and then we find the log magnitude of the Fourier transform of all of the above images. We do this by applying the provided line of python code to each one: plt.imshow(np.log(np.abs(np.fft.fftshift(np.fft.fft2(gray_image))))). Here's how they look:



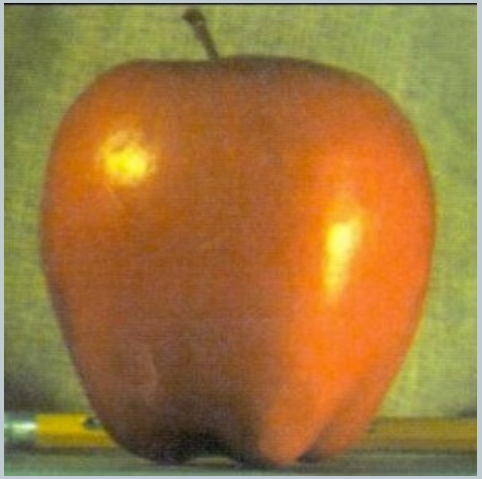
Aligned Trump | Aligned Harris | High Frequency Trump | Low Frequency Harris | Harrump

Let's take a closer look at these. In Trump's frequency plots, the most noticeable difference is a bright spot at the very center in the original image's plot, and a lack thereof in the high-frequency version. This makes sense, as low frequencies are typically present in the center of the plot and we tried to removed them. In Harris' frequency plots, we observe that the four quadrants are noticeably dimmer in the low-frequency plot. This again makes sense, due to the removal of higher frequencies that tend to appear farther from the center. There is also an intriguing artifact in the Harris low-frequency plot-- a faint 3x3 grid is visible in each of the four quadrants. I do not have a good explanation as to why these appeared.

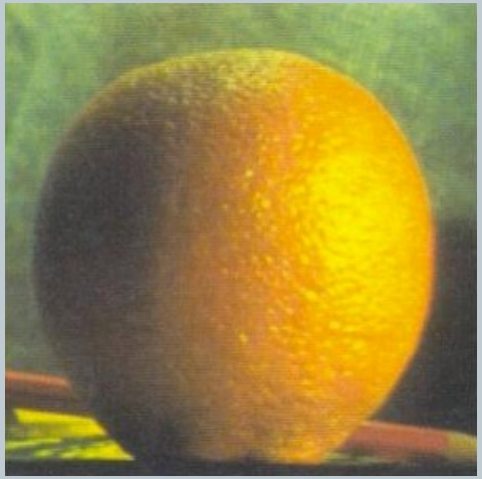
Multi-resolution Blending and the Oraple journey

Now we will attempt to seamlessly blend images together according to some mask. We will follow a multi resolution blending approach as described in the 1983 paper by Burt and Adelson. We will create a so-called "image spline", by gently distorting two images at different bands of frequencies around their seam.

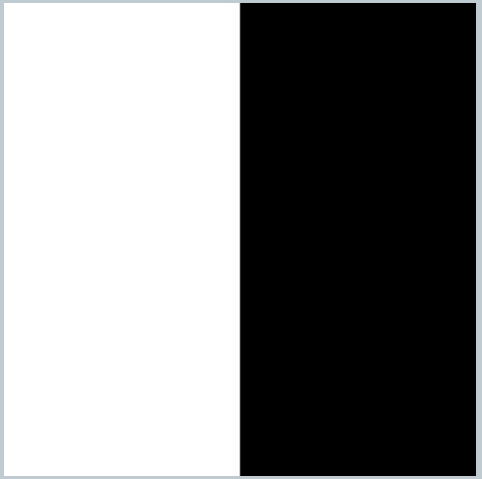
Out first step is to create and visualize Gaussian and Laplacian stacks for each image as shown below. The Gaussian filter is simply applied to our images repeatedly with the same kernel size (11 in our case) to create the Gaussian stack. The i-th layer of the Laplacian stack is created by computing gaussian_stack[i] - gaussian_stack[i-1]. Here we created stacks of 300 images each. A half black half white mask is used to mask both images. At the end, we can simply sum up the layers of our combined Laplacian stack to get our final blended image. Here is the Oraple:



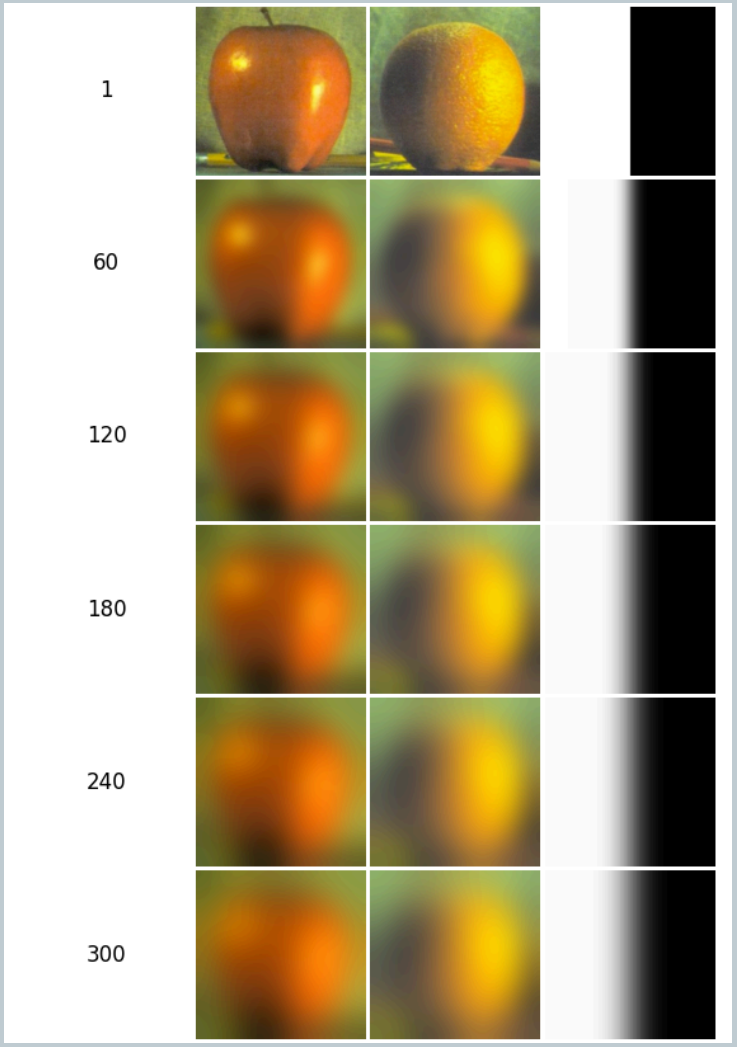
Apple

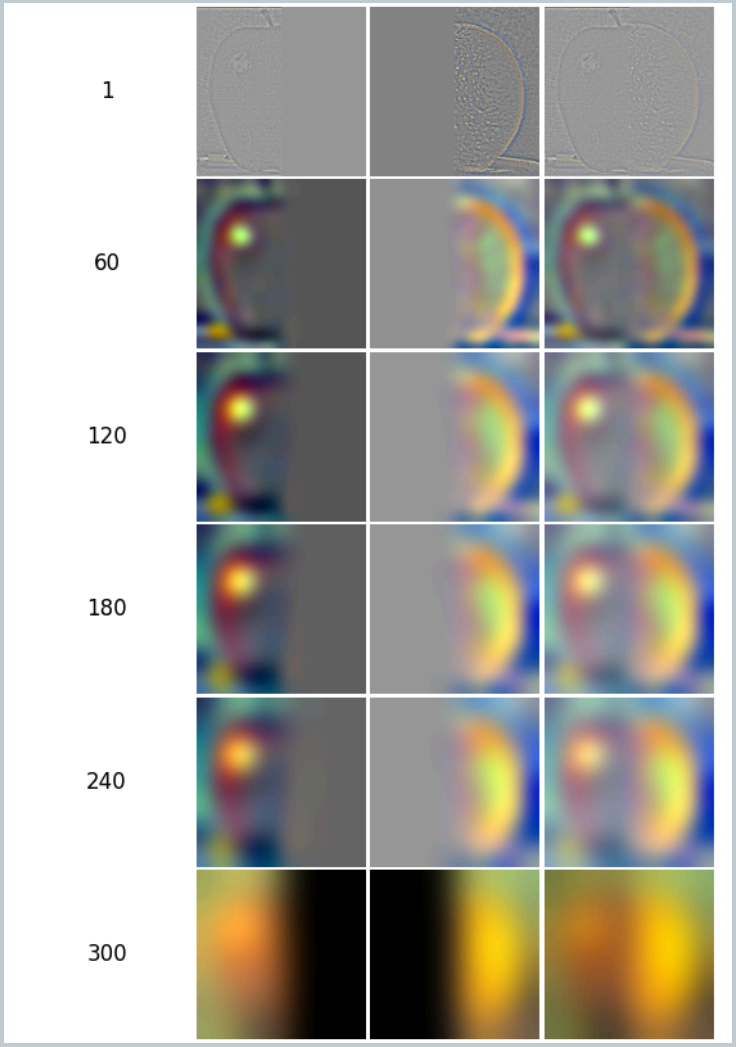


Orange



Mask


1
60
120
180
240
300


1
60
120
180
240
300

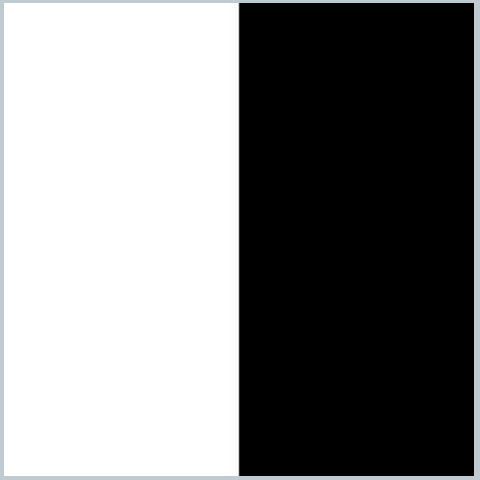
Gaussian Stacks Laplacian Stacks

Oraple

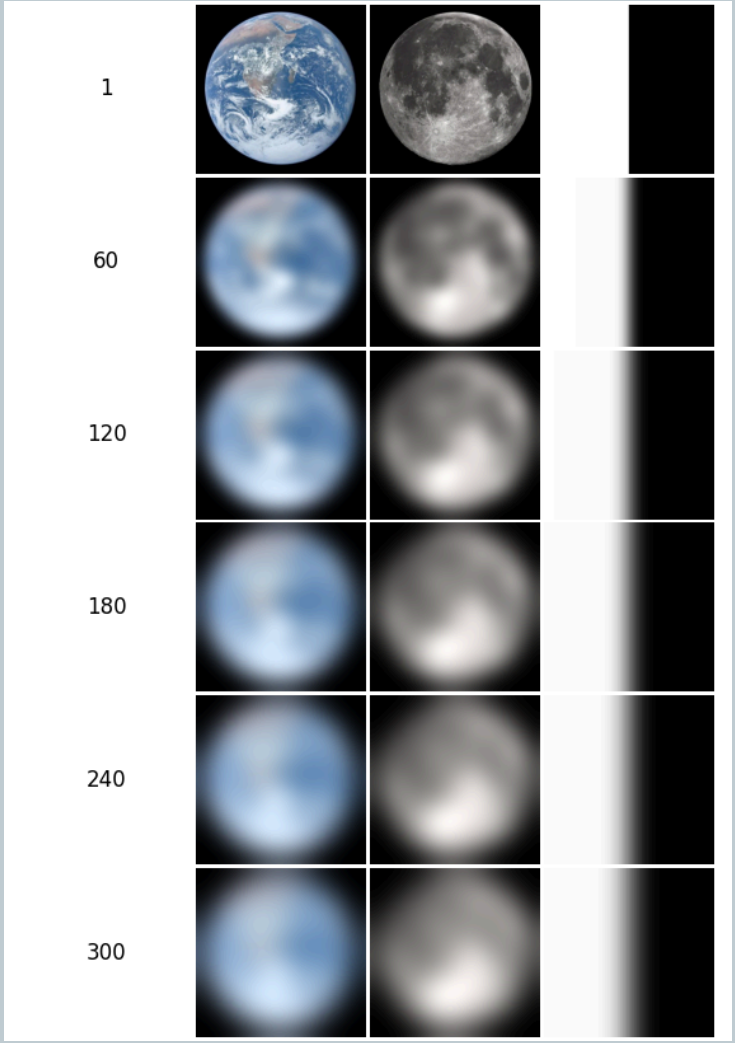
Lovely. Now we also attempt to seamlessly blend an image of the Earth and our Moon, courtesy of Google. I used the same image alignment code from earlier to align the 2 images. Again, we use the same mask, same Gaussian kernel size of 11, and create stacks of 300 images:

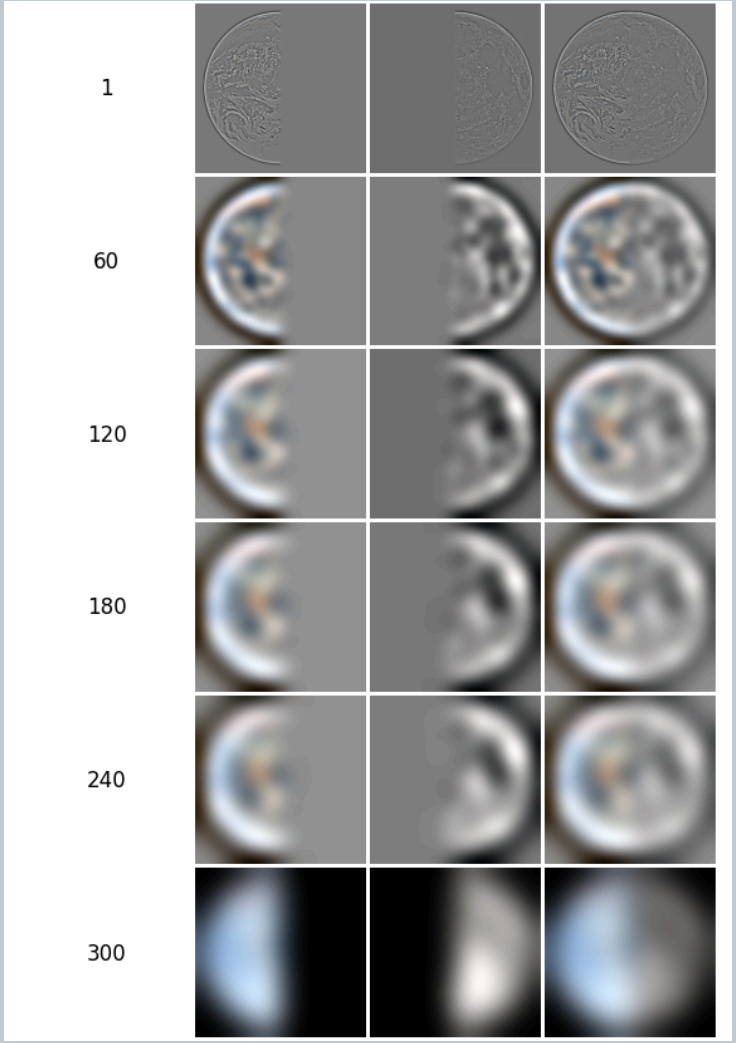
Earth

Moon



Mask


1
60
120
180
240
300


1
60
120
180
240
300

Gaussian Stacks

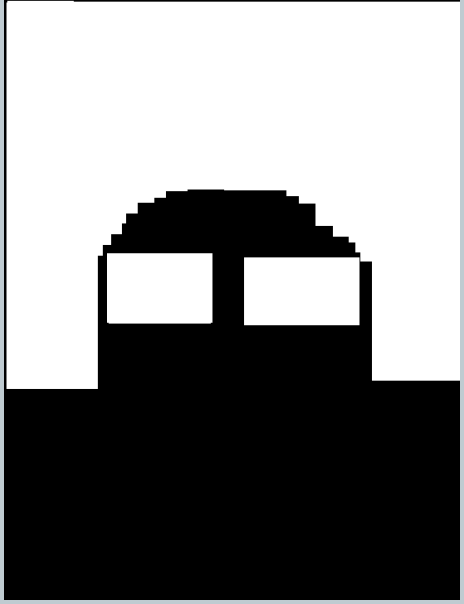
Laplacian Stacks

Moorth

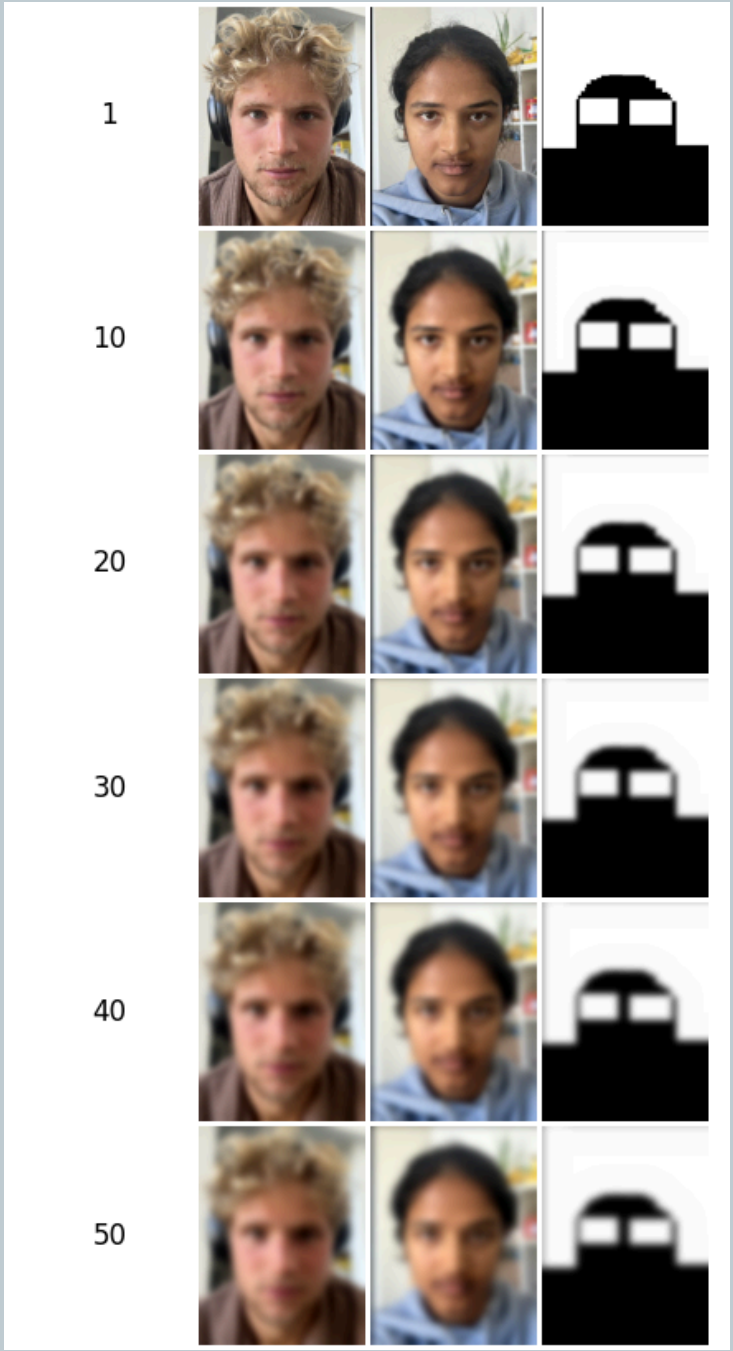
Finally, we attempt to use an irregular mask. Here, I am blending an image of myself with an image of my friend Wiktor. I aligned our headshots and then created a custom mask that combines his eyes and hair with my face. Again, using a Gaussian kernel of size 11, and this time the stacks have 50 images. Here are the results:

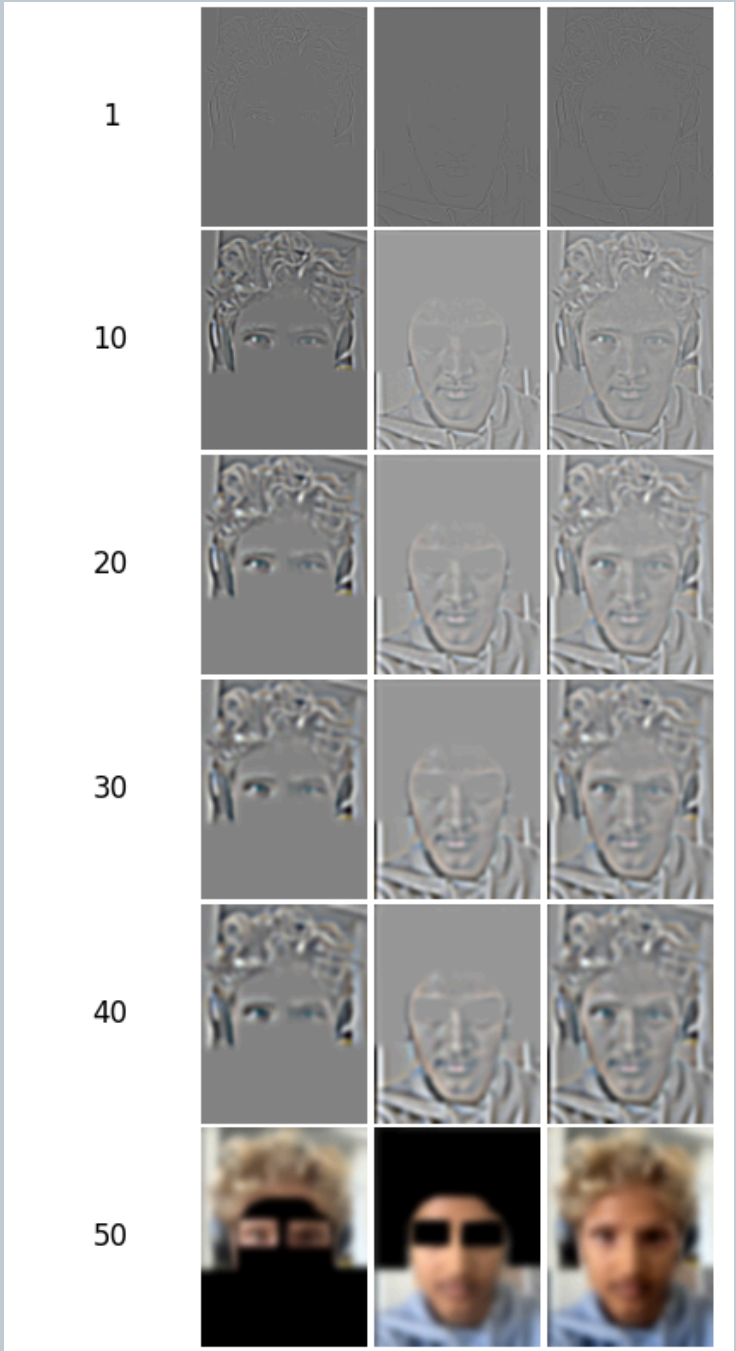
Wiktor

Meenakshi



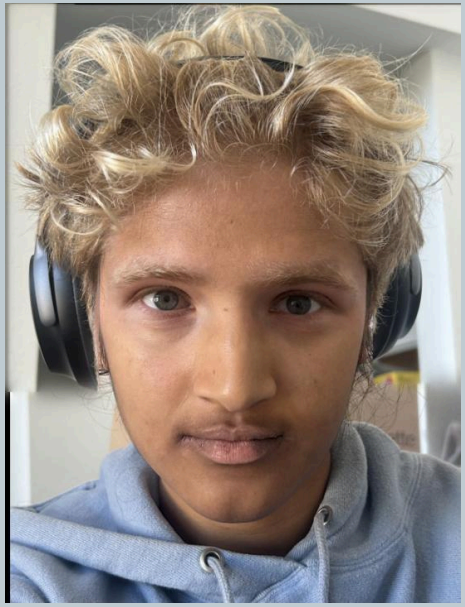
Mask


1
10
20
30
40
50


1
10
20
30
40
50

Gaussian Stacks

Laplacian Stacks



Wikshi

Woah. That worked better than I thought it would.

Anyways, that's the end of the project! I had a lot of fun with this one, and I appreciate the amount of creativity we were allowed to express. I think the most important thing I learned from this project was to always look for runtime optimizations. At some point, I found myself trying to use larger and larger Gaussian filters that were taking forever to run. I realized there was a much better solution to this, which was literally just applying a smaller sized Gaussian filter a larger number of times. This fixed most of the issues I was having. I also learned to appreciate the mighty power of the Gaussian filter. I had no idea Gaussian curves were used so extensively in computational photography, so that really surprised me.