CS 674
Assignment 4

Bryson Lingenfelter Nate Thom Christopher Lewis

Submitted: December 2, 2019
Due: December 2, 2019

0.1 Division of Work

0.1.1 Coding

Bryson: Motion Blur, Frequency Filtering
Chris: Noise Removal
Nate: Convolution in the Frequency Domain, Homomorphic Filtering

0.1.2 Writing

Bryson: Motion Blur, General Technical Discussion
Chris: Noise Removal
Nate: Convolution in the Frequency Domain, Homomorphic Filtering

1 Technical Discussion

In all sections we use frequency filtering to analyze and modify images. The steps of frequency filtering are as follows:

1. Given an NxN image and an MxM filter specified in the spatial domain, we zero-pad both to the smallest 2^k greater than $N + M - 1$. If the filter is specified in the frequency domain, we pad the image to the smallest 2^k greater than N.
2. Both the image and filter (if specified in the spatial domain) are centered in the spatial domain by multiplying each intensity value $f(x, y)$ by $(-1)^{x+y}$.
3. Both the image and filter (if specified in the spatial domain) are converted to the frequency domain using the 2D FFT.
4. Filtering is performed by performing complex multiplication on the image and the filter.
5. The image is converted back to the spatial domain using the 2D FFT.
6. Centering is removed in the spatial domain by multiplying each intensity value $f(x, y)$ by $(-1)^{x+y}$.
7. Image is unpadded.

1.1 Noise Removal

Spatial domain techniques struggle to remove periodic noise from an image because they only consider the pixel values in a limited region. The Fourier transform is ideal for removing these types of noise because it splits an image into its constituent frequencies, which can accurately capture a recurring noise pattern. Certain types of periodic noise, such as Cosine noise, will appear as frequencies far from the center of spectrum with abnormally high magnitude compared to the surrounding frequencies. Such frequencies can be isolated with a band-pass filter or removed with a band-reject filter, hopefully removing noise from the image.

Band-reject filters cause only frequencies within a certain range to become close to zero. There are many types of band-reject filters. There are only three that we considered for removing noise from pictures. These are: the Ideal band-reject filter, the Butterworth band-reject filter, and the Gaussian band-reject filter. The ideal band-reject filter strictly contains only 0's and 1's for it's values. 0 only exists when: $D_0 - \frac{W}{2} \leq D \leq D_0 + \frac{W}{2}$, otherwise the value is 1, which is considered white in this case. The Butterworth band-reject filter is described as: $H(u, v) = \frac{1}{1+[\frac{DW}{D^2-D_0^2}]^{2n}}$, where D is the distance from the center of the filter, D_0 is the cutoff frequency, and W is the width of the band. The Gaussian band-reject filter is described as: $H(u, v) = 1 - e^{-[\frac{D^2-D_0^2}{DW}]^2}$, where, again, D is the distance from the center of the filter, D_0 is the cutoff frequency, and W is the width of the band.

To specifically obtain the noise pattern from an image, we use a bandpass filter. Bandpass filters perform the opposite operation of a band-reject filter, which means that bandpass filters only keep frequencies inside of a certain range. This can be done very easily if you have the band-reject filters, as all these filters are described as is: $H_{BP}(u,v) = 1 - H_{BR}(u,v)$ where H_{BP} is the bandpass filter, and H_{BR} is the band-reject filter.

1.2 Frequency Domain Convolution

When attempting to develop and apply some transfer function for improving image quality we have the option of processing this image in the spatial domain or the frequency domain. In this experiment we explore the differences between both methods.

In order to achieve an effective comparison we must apply equivalent filters to equivalent images in both spatial and frequency domains. Any image of relatively good image quality is acceptable and we use the sobel filter because it is simple and well known. Sobel approximates the derivative of pixel values across an image, thus it computes edge detection. We simply convolve the sobel filter, with respect to x, across the selected image.

Sobel: $\frac{\partial f}{\partial y} =$

-1	-2	-1
0	0	0
1	2	1

$\frac{\partial f}{\partial x} =$

-1	0	1
-2	0	2
-1	0	1

The difficulty of this experiment lies in converting the spatial sobel filter into the frequency domain. To accomplish this we must first pad both the image and filter to account for wraparound error. We pad the image and the filter according to these two equations:

$$f_p(x,y) = \begin{cases} f(x,y) & 0 \leq x \leq A-1 \quad and \quad 0 \leq y \leq B-1 \\ 0 & A \leq x \leq P \quad or \quad B \leq y \leq Q \end{cases}$$

and

$$h_p(x,y) = \begin{cases} h(x,y) & 0 \leq x \leq C-1 \quad and \quad 0 \leq y \leq D-1 \\ 0 & C \leq x \leq P \quad or \quad D \leq y \leq Q \end{cases}$$

with

$$P \geq A + C - 1$$

and

$$Q \geq B + D - 1$$

It is very important that we maintain the symmetry of the filter when padding. To ensure this we append a row and column of zeros to the leading dimensions of the filter before padding. We also pad such that the center of the padded filter coincides with the center of the original filter. Now, we convert the image and filter to the frequency domain by centering (multiply each element by $(-1)^{x+y}$) and compute the DFT of the result. Finally, the filter is multiplied (complex hadamard product) across the image. The image is lastly converted back to the spatial domain for visualization.

Our results are as we expect them to be. The derivative in the x direction is approximated, which led to edge detection. In addition, both techniques yield the exact same results.

1.3 Motion Blur

Image restoration is performed in the frequency domain by dividing the degraded image by degradation kernel. This is possible because $g(x,y) = f(x,y) * h(x,y) \to G(u,v) = F(u,v)H(u,v)$, so an estimation of f (called $\hat{f}$) can be computed as $\hat{f}(x,y) = \mathcal{F}^{-1}(\frac{G(u,v)}{H(u,v)})$. This method is called **inverse filtering**. However, due to numerical stability issues when performing division, the estimated $\hat{f}$ can have prominent artifacts. Very small values in the degradation kernel result in very large values in the inverse kernel, and if there are too many such values the image will become completely obscured. To deal with this, the restoration is only carried out in a limited radius around the center of the spectrum to avoid large outliers causing artifacts.

Unfortunately, inverse filtering is highly sensitive to noise. Another method for for performing deconvolution to restore images is **Wiener filtering**. Wiener filtering attempts to minimize least-squares error accounting for noise by factoring in the noise power spectrum and the image power spectrum:

$$F(u,v) = \frac{1}{H(u,v)} \frac{|H(u,v)|^2}{|H(u,v)|^2 + \frac{S_n(u,v)}{S_f(u,v)}} G(u,v)$$

The signal-to-noise ratio $\frac{S_n(u,v)}{S_f(u,v)}$ is not known in practice, so it is approximated by a parameter k which we tune manually. By explicitly accounting for noise, Wiener filtering is far more robust than inverse filtering.

To test these methods, we look at degradation from motion blur. Motion blur is caused by camera motion during image capture, and can be modeled as $g(x,y) = \int_0^T f(x - x_0(t), y - y_0(t))dt$, where g is the degraded image, T is the exposure time of the camera, $x_0(t)$ is the horizontal position of the camera at time t, and $y_0(t)$ is the vertical position of the camera at time t. The Fourier transform of the degraded image is therefore:

$$\begin{aligned}
\mathcal{F}^{-1}(g(x,y)) &= \int\int_{-\infty}^{\infty}\int_0^T f(x - x_0(t), y - y_0(t))dt e^{-j2\pi(ux+vy)}dxdy & w = x - x_0(t)\\
&= \int_0^T e^{-j2\pi(ux_0(t)+vy_0(t))}dt \int\int_{-\infty}^{\infty} f(w,z)e^{-j2\pi(uw+vz)}dwdz & z = y - y_0(t)\\
&= \int_0^T e^{-j2\pi(ux_0(t)+vy_0(t))}dt F(u,v)
\end{aligned}$$

With $x_0(t) = \frac{at}{T}$ and $y_0(t) = \frac{bt}{T}$ (this allows only uniform linear motion), the degradation kernel integrates to $H(u,v) = \frac{T}{\pi(ua+vb)} sin(\pi(ua+vb))e^{-j\pi(ua+vb)}$. We sample the filter from $-\frac{n}{2}$ to $\frac{n}{2}$ so we do not need to center it.

1.4 Homomorphic Filtering

Homomorphic filter is used to help alleviate uneven illumination in an image. It accomplishes this by modeling an image as the multiplication of an illumination component and a reflection component. This translates to convolution in the frequency domain, making it difficult to deal with illumination and reflection separate. To attenuate illumination and amplify reflectance, the two components must be separated. We can accomplish this by applying the *log* function to the image, because multiplication becomes addition when using *log*. $ln(f(x,y)) = ln(i(x,y)r(x,y)) = ln(i(x,y)) + ln(r(x,y))$. Addition remains addition after the Fourier transform, so the components are no longer convolution in the frequency domain:

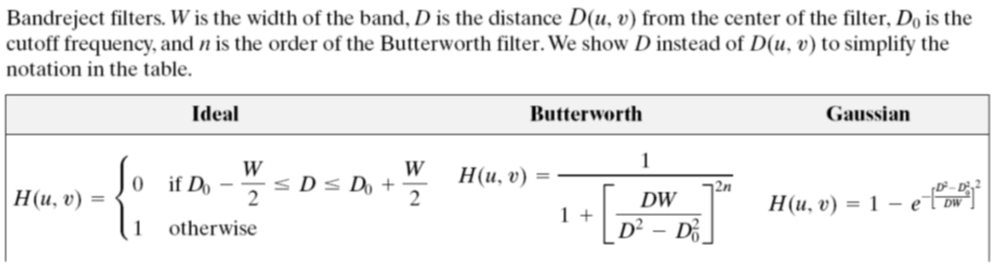
$$\begin{aligned}
\mathcal{F}(f(x,y)) &= \mathcal{F}(ln(i(x,y)) + ln(r(x,y)))\\
F(x,y)H &= \mathcal{F}(ln(i(x,y)))H + \mathcal{F}(ln(r(x,y)))H
\end{aligned}$$

The filter H is a high-emphasis filter, defined as $H(u,v) = (\gamma_H - \gamma_L)(1 - e^{-c(\frac{u^2+v^2}{D_0^2})}) + \gamma_L$ where D_0 is the cutoff frequency of the high-emphasis filter and γ_L and γ_H are the gains of the high and low frequencies, respectively. This filter emphasizes the details in the image without removing low frequencies. After filtering, each intensity value r is replaced by e^r to undo the *log* function.

2 Implementation Details

2.1 Noise Removal

Most of the time during implementation of real world noise removal, you don't have the exact function for the noise in the image, although we should have some understanding of the noise source. In this case, we do have some understanding of the noise source, it is a cosine noise function that was added to this image. Sinusoidal noise is generated from interferences during electrical image acquisition. This means we can use band filters in the frequency domain to change which frequencies we can see. To specifically remove the cosine noise from the image, we'd use a band-reject filter. Again, our picks are limited down to three options for the band-reject filter: Ideal, Butterworth, and Gaussian, which can be seen modeled both mathematically and in 3-D in Figure 2.1.



Bandreject filters. W is the width of the band, D is the distance $D(u, v)$ from the center of the filter, D_0 is the cutoff frequency, and n is the order of the Butterworth filter. We show D instead of $D(u, v)$ to simplify the notation in the table.

Ideal	Butterworth	Gaussian
$H(u,v) = \begin{cases} 0 & \text{if } D_0 - \frac{W}{2} \leq D \leq D_0 + \frac{W}{2} \\ 1 & \text{otherwise} \end{cases}$	$H(u,v) = \dfrac{1}{1 + \left[\dfrac{DW}{D^2 - D_0^2}\right]^{2n}}$	$H(u,v) = 1 - e^{-\left[\frac{D^2 - D_0^2}{DW}\right]^2}$

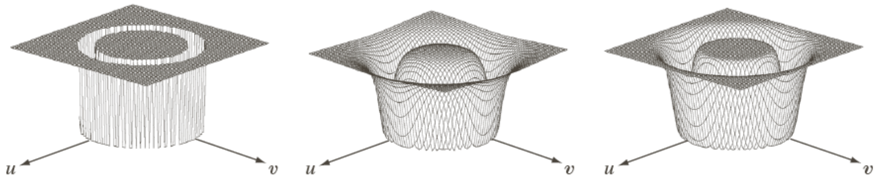


Figure 1: The band reject functions mathematical representations and their modeled counterparts.

2.2 Frequency Domain Convolution

The implementation for noise remove is quite simple. To begin, we store the sobel filter and input image in array-like data structures. Next, we compute the convolution of the filter across the input image. We are careful to ensure that any regions in which the filter is not entirely overlapping the image are padded with zeros.

Next, we pad both the mask and the image with zeros such that they are 512x512. We place the filter into the larger array of zeros such that the centers of both coincide. Essentially the center of the smaller filter appears at the center of the padded filter. Both padded arrays are transformed with the provided DFT function. We computer element wise multiplication and then find the inverse transform. Finally, both images are stored to memory for visualization.

2.3 Motion Blur

In implementation, a spatial filter is represented as a matrix containing real and imaginary components. As a result, to implement motion blur we first need to separate the kernel into real and imaginary components:

$$\begin{aligned} H(u,v) &= \frac{T}{\pi(ua+vb)} sin(\pi(ua+vb))e^{-j\pi(ua+vb)} \\ &= \frac{T}{\pi(ua+vb)} sin(\pi(ua+vb))(cos(\pi(ua+vb)) - jsin(\pi(ua+vb))) \\ &= \frac{T}{\pi(ua+vb)} sin(\pi(ua+vb))cos(\pi(ua+vb) - j\frac{T}{\pi(ua+vb)} sin(\pi(ua+vb))sin(\pi(ua+vb) \\ &= \frac{T}{\pi(ua+vb)} \frac{1}{2} sin(2\pi(ua+vb)) - j\frac{T}{\pi(ua+vb)} sin^2(\pi(ua+vb)) \end{aligned} \tag{1}$$

This allows us to sample a both the real and imaginary values at each point (u, v). Once the kernel has been computed, we implement an inversion function so we can invert the kernel then use multiplication as with frequency filtering. Again, we need to separate the real and imaginary components to implement the operation:

$$\begin{aligned}\frac{1}{a+bj} &= \frac{1}{a+bj}\frac{a-bj}{a-bj} \\ &= \frac{a-bj}{a^2+abj-abj-b^2j^2} \\ &= \frac{a-bj}{a^2+b^2} \\ &= \frac{a}{a^2+b^2}+j\frac{-b}{a^2+b^2}\end{aligned} \tag{2}$$

To implement the cutoff radius used in inverse filtering, the inverse kernel is set to 1 whenever $\sqrt{u^2+v^2}$ exceeds a given radius.

For Wiener filtering, we use complex division instead of inverting the filter. Again, we need to separate the operation into real and imaginary components for implementation:

$$\begin{aligned}\frac{a+bj}{c+dj} &= \frac{a+bj}{c+dj}\frac{c-dj}{c-dj} \\ &= \frac{ac-adj+bcj+bdj^2}{c^2+cdj-cdj-d^2j^2} \\ &= \frac{ac+bd+bcj-adj}{c^2+d^2} \\ &= \frac{ac+bd}{c^2+d^2}+j\frac{bc-ad}{c^2+d^2}\end{aligned} \tag{3}$$

For implementation convenience, we use the numpy function `numpy.random.normal` rather than the provided Box-Muller method to sample from a gaussian distribution for adding noise to images. The results are functionally identical.

To apply motion blur, we use the following process:

1. Load the image from disk into an array, adding 0s for imaginary components.
2. Add $Gaussian(\mu, \sigma)$ to each pixel to corrupt the image with noise. For tests without noise, $\mu = 0$, $\sigma = 0$ is used.
3. Center the spectrum of the image in the spatial domain.
4. Convert the Fourier transform of the image using the 2D FFT.
5. Generate the motion blur filter in the frequency domain using the provided T, a, and b parameters.
6. Multiply the image and filter in the frequency domain.
7. Convert the inverse transform of the filtered image.
8. Remove centering and padding from the image in the spatial domain.

To test invert filtering, we perform a similar process on the degraded image, but do not add additional noise. We also invert the filter using a given cutoff before multiplying with the image. If the filtering method is Wiener, we instead compute $\frac{|H(u,v)|^2}{|H(u,v)|^2+k}$ and $\frac{G(u,v)}{H(u,v)}$ using complex division then multiply the two to obtain the filtering result.

2.4 Homomorphic Filtering

Our implementation of homomorphic filtering follows much of the same work flow as our experiments comparing spatial filtering to frequency filtering. There are however some differences. In implementation we will gain control over the illumination and reflectance portions of the image once we have converted to the frequency domain and applied the homomorphic function. Thus, we begin by calculating the natural log of each pixel value in our image. In order to avoid undefined values we increase each pixel value by a value of 1. This is accounted for when we perform this process in reverse to restore our original image.

Following the computation of natural log over the input image, we compute the forward fourier transform and apply the homomorphic function:

$$H(u,v) = (\gamma_H - \gamma_L)(1 - e^{-c(\frac{u^2+v^2}{D_0^2})}) + \gamma_L$$

Since there is interest in the difference and interaction of differing values of γ_L and γ_H, we iterate over all possible configurations of the the parameters between [0,1] and [1,2] respectively. We iterate with a step size of tenths. This enables the formation

3 Results and Discussion

3.1 Noise Removal

First, we apply the Gaussian spatial 7x7 and 15x15 filters to the image. Their results can be seen in Figure 2. As you can see, the 7x7 filter does pretty much nothing to the initial image. The 15x15 causes so much smoothing that the image is now blurry, and it still leaves trace amounts of the sinusoidal noise.

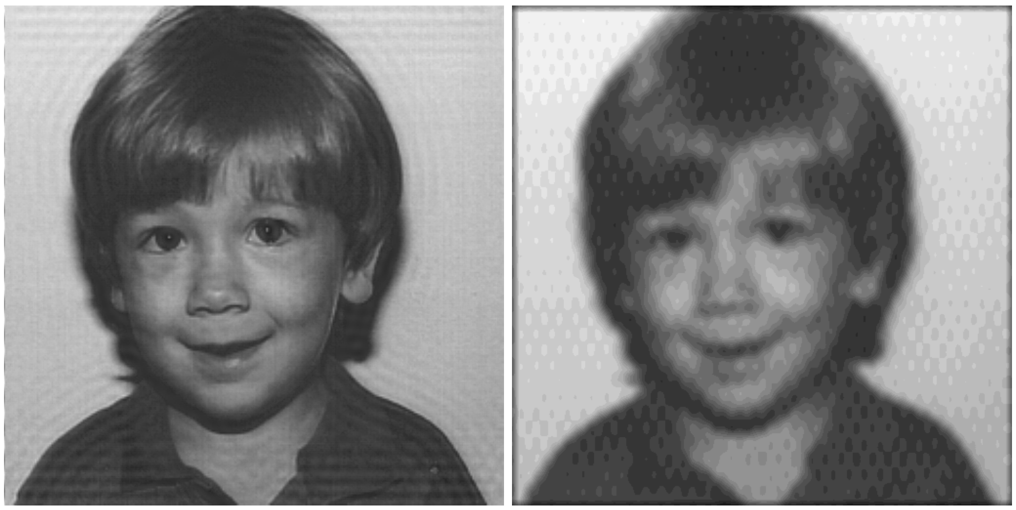


Figure 2: The Gaussian Spatial 7x7 filter on noisy boy(left) and the Gaussian 15x15 filter on noisy boy(right).

3.2 Frequency Domain Convolution

We are only partially satisfied with our results for experiment 2. We are certain that both filters should be providing equivalent results. We are also certain that the results for sobel, with respect to x, in the spatial domain are correct. Unfortunately, there appears to be an error with our sobel mask in the frequency domain. This error almost certainly manifests in our padding function. The center of the initial filter does not coincide with the center of the padded filter. Therefore we are not preserving the symmetry of the filter when converting to the frequency domain. All of this being said, the results are nearly correct. As expected, the values edges were enhanced because sobel is an approximation of the derivative. Additionally, areas of constant intensities are reduced to 0. The grey values seen in 3 are due to normalization for visualization purposes. Once again, we know that the results should be identical.

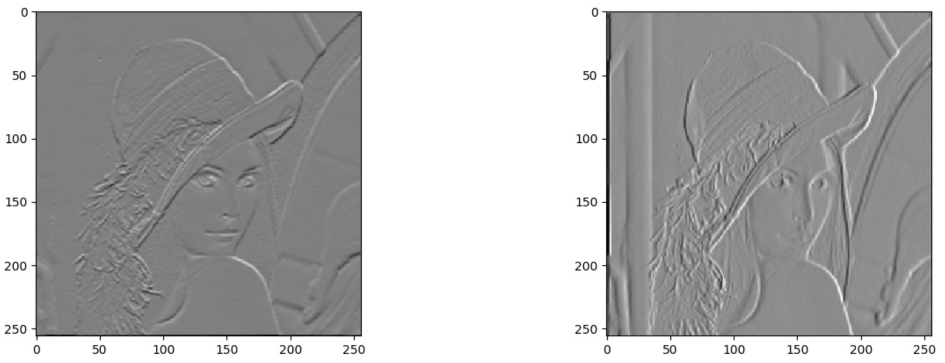


Figure 3: Results of the sobel masking, in the x direction, in the spatial domain (left) and in the frequency domain (right).

3.3 Motion Blur

First, we generate the motion blur filter. The motion blur somewhat like a sinc function, but only in one dimension and turned at an angle. This results in the spectrum being attenuated only in certain directions, causing the blurring. Because the filter is generated by a sine function, there are several areas where the magnitude is very small. This can be clearly seen by inverting the spectrum, as shown in Figure 4. Regions with very small magnitude now show up as bright lines in the inverted spectrum.

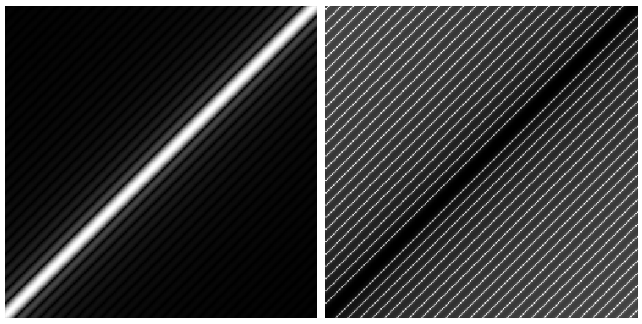


Figure 4: Spectrum of the motion blur (left) and its inverse (right).

We originally found that these lines of high values caused major issues for inverse filtering, regardless of cutoff radius. After experimenting with limiting small values to 0.05 (the modified filters are shown in Figure 5), which helped prevent them from overpowering the inverse, we discovered that the poor results were a result of the centering being 1 pixel off. We provide some tests on this limited filter later in the section.

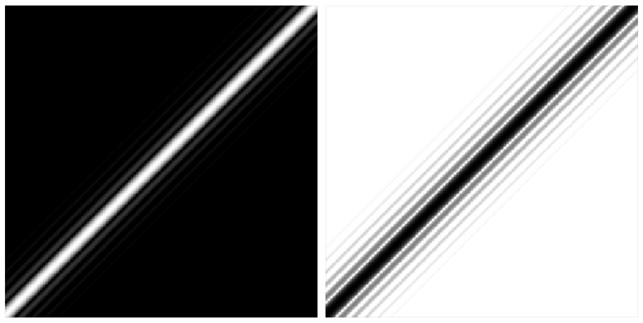


Figure 5: Spectrum of the motion blur (left) and its inverse (right) after limiting very small values to be no less than 0.05.

To test the filter, we use the Lenna image. Blurring results are shown in Figure 6. As can be seen from the figure, the degradation filter blurs values towards the upper left corner of the image. This makes sense, because the camera trajectory would follow $x_0(t) = .1t$ and $y_0(t) = .1t$ so the blur should follow a 45 degree angle.

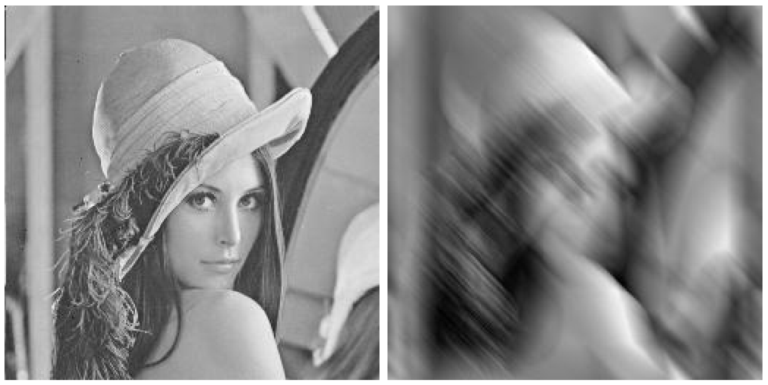


Figure 6: Blurring results on the Lenna image with $T = 1$, $a = 0.1$, $b = 0.1$

We first test inverse filtering, shown in Figure 7. The motion blur seems to be more or less successfully removed, but the removal process leaves behind major artifacts regardless of what cutoff ratio we use. A very small cutoff ratio fails to remove the blurring and still introduces major artifacts, and larger cutoffs more successfully remove blurring but do not remove the artifacts.

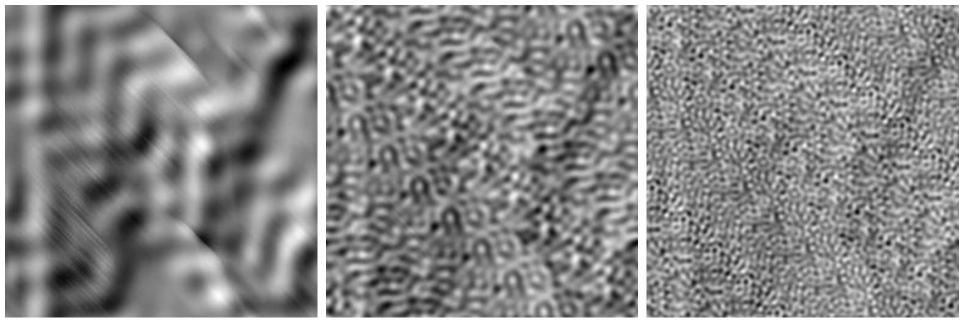


Figure 7: Inverse filtering with a cutoff radius of 10, 30, and 50.

We believe our lackluster results are caused by the small values discussed previously. As previously shown in Figure 5, there are still large outliers near the center of the spectrum so using a cutoff radius does not help very much. To test this hypothesis, we try inverse filtering with increasingly strict cutoffs for small values. The results are shown in Figure 8.

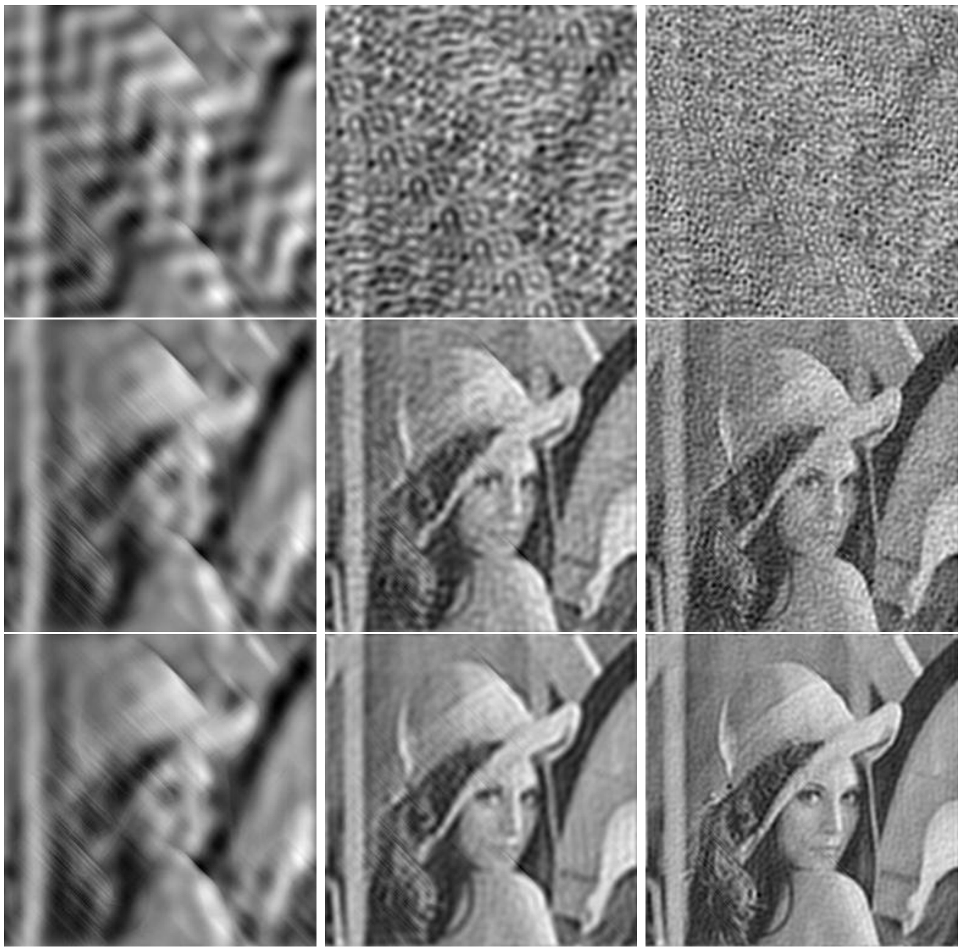


Figure 8: Inverse filtering with a cutoff of 10, 30, and 50 (left to right). The cutoff used for the filter increases from top to bottom from 0.0005 to 0.005 to 0.05.

As seen in the Figure, truncating small values substantially improves the results of inverse filtering. If small values are limited to be no less than 0.05, the deconvolution is far more successful than with the original filter. This shows that filter noise, in addition to image noise, can negatively impact results for inverse filtering.

To see how much better Wiener filtering performs, we try deconvolving using k values ranging from 0.0001 to 0.1. For these experiments we again use the original filter with no limiting. As shown in Figure 9, smaller values of k look similar to the inverse filtering results, while larger values of k more closely resemble the blurred image. This is expected, because a smaller k results in the noise term being very small, and with $k = 0$ Wiener filtering is equivalent to inverse filtering. With a high k, however, the magnitude of the filter in the denominator grows, weakening the amount of deconvolution. At $k = 1$, the filtering has no effect. The Wiener filter performs substantially better than the inverse filter, with far fewer artifacts appearing in the restored images.

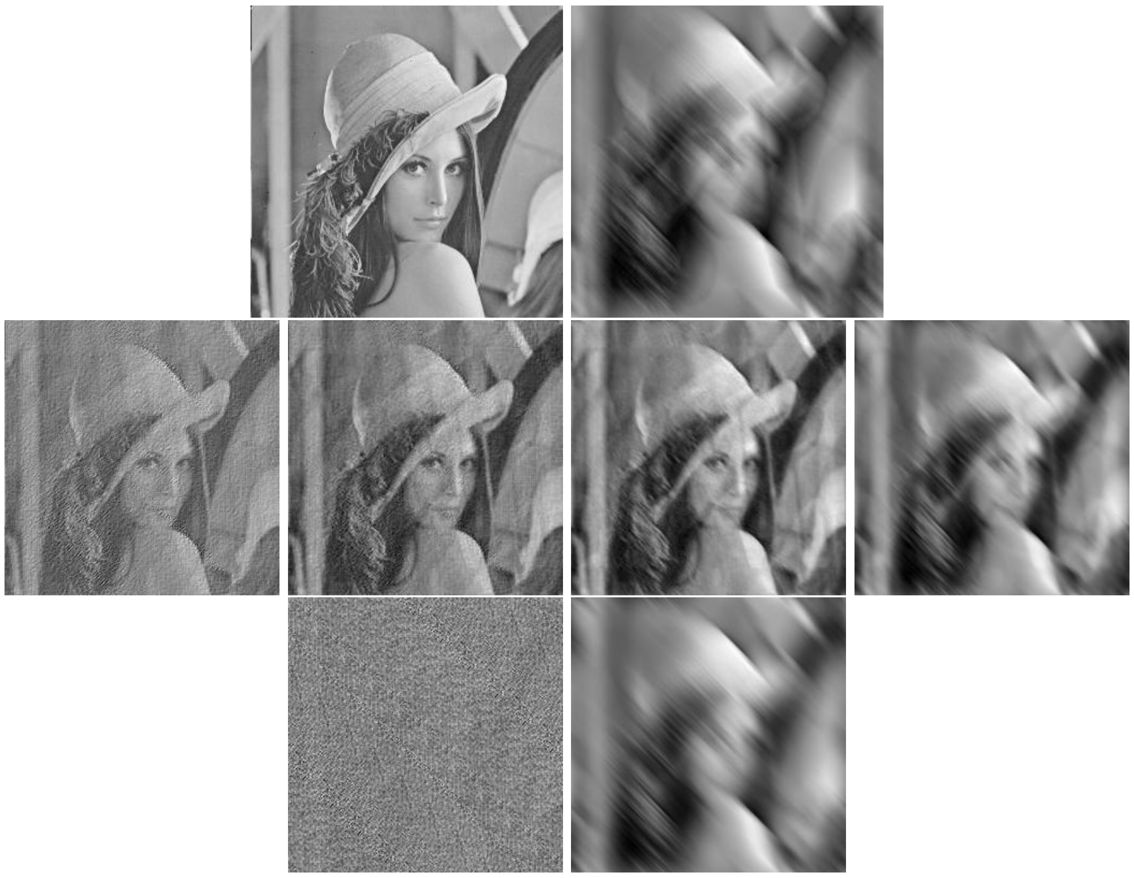


Figure 9: The first two images show the original and blurred image. The following 4 images show Wiener filtering with k in [0.0001, 0.001, 0.01, 0.1]. On the bottom we show results with $k = 0$ and $k = 1$, respectively

Next, we compare inverse filtering and Wiener filtering given different amounts of noise. Figure 10 shows the original image after adding noise, the blurred image, and the restored image with inverse or Wiener filtering. We use a cutoff radius of 10 for inverse filtering and a k value of 0.01 based on visual comparison of the results above. Wiener filtering performs substantially better in all cases, with inverse filtering being almost uninterpretable in all cases. The distinction is most pronounced when the noise grows to 100, at which point inverse filtering completely obscures the original image. Wiener filtering, however, produces a fairly reasonable image that is more clear than the noisy image it was restoring. These results make sense because Wiener filtering explicitly considers noise, and as a result is far better equipped to deal with noise than inverse filtering.

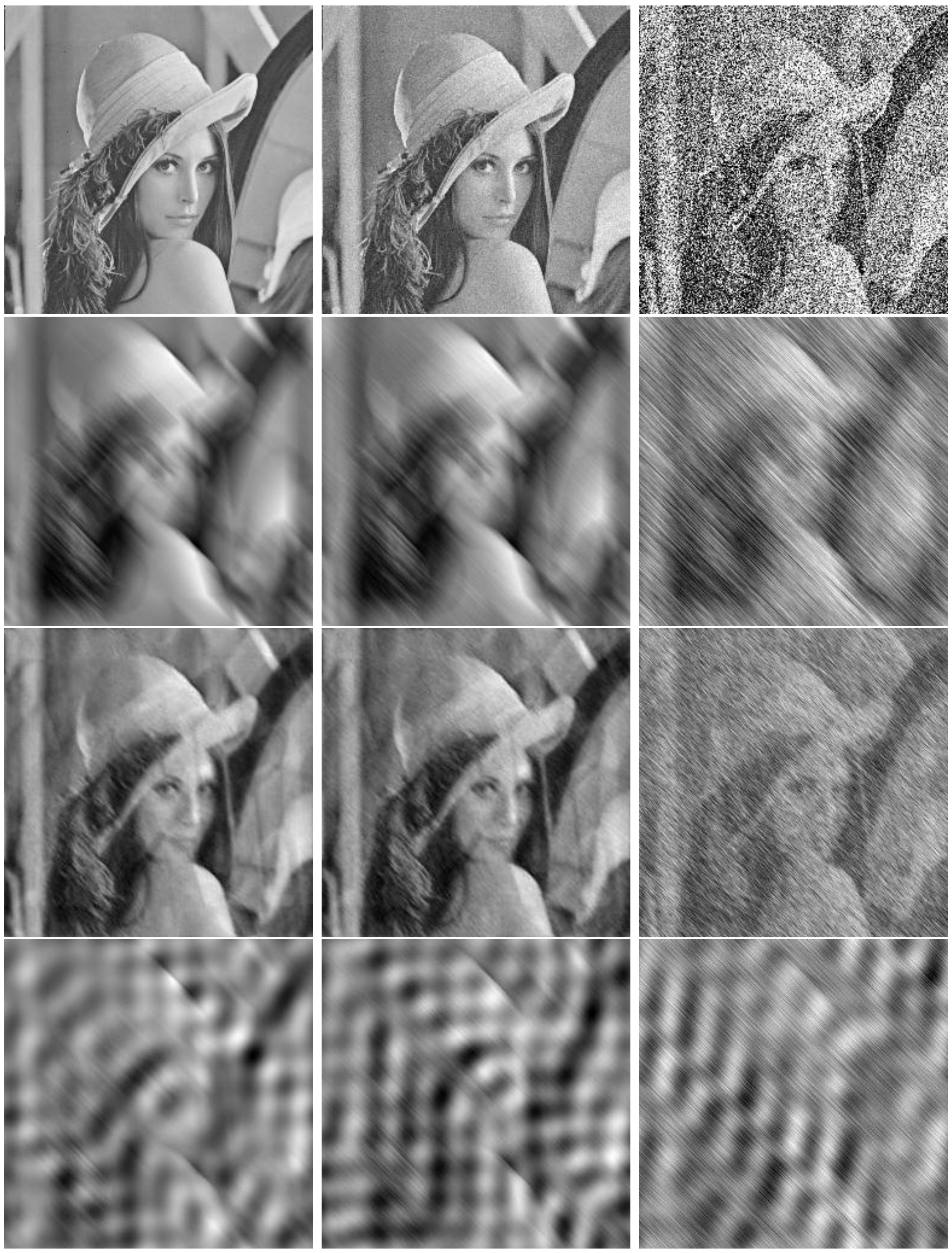


Figure 10: Top to bottom: original image, image degraded by motion blur, Weiner filtering results, inverse filtering results. The standard deviation of the noise increases from 1 to 10 to 100 from left to right.

For better comparison between inverse and Wiener filtering, we limit small values to 0.005 as before to help inverse filtering. The results of this comparison are shown in Figure 11. We use a cutoff radius of 30 and $k = 0.01$ for these results. Wiener filtering is relatively unaffected by the small value limit, but inverse filtering is massively affected. By limiting small values inverse filtering is able to produce reasonable results although the artifacts are still more prominant than in Wiener filtering.

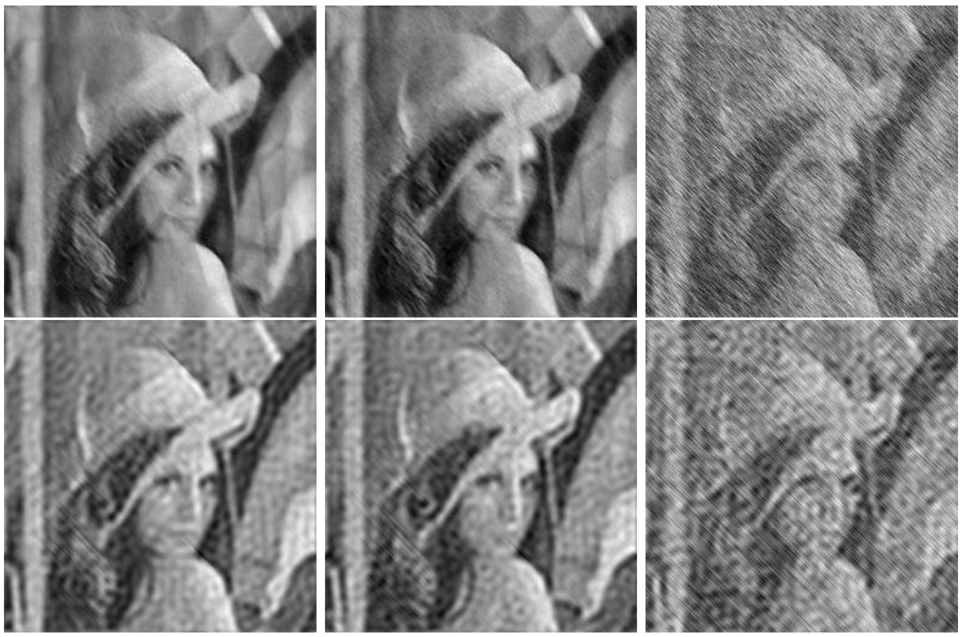


Figure 11: Results with inverse filtering (top) and Wiener filtering (bottom), with the standard deviation of the noise increasing from 1 to 10 to 100 from left to right.

3.4 Homomorphic Filtering

Our initial homomorphic filtering results appeared to be excellent, but did not vary with gamma low and gamma high. This was a red flag for us, and ended up being an issue of mask centering. You can see the initial result in 12. The error seemed to cause results similar to homomorphic filter with large values of gamma low and gamma high. This is likely because all values were being positively scaled due to incorrect shifting of filter values.

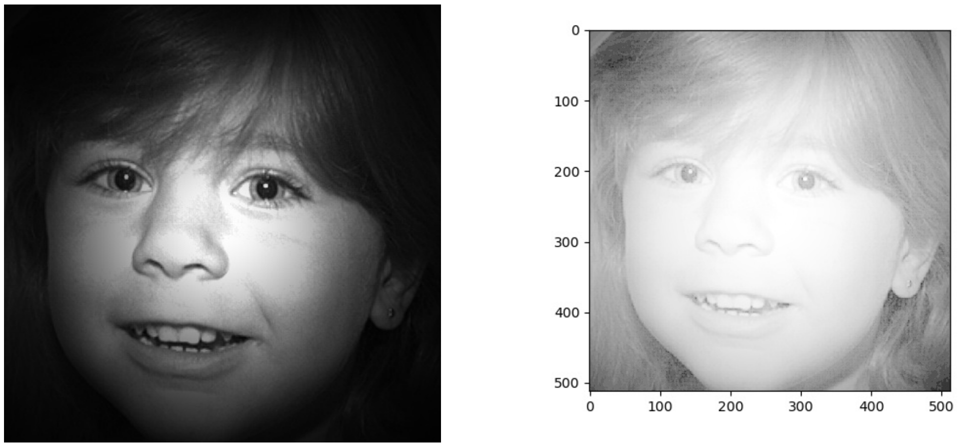


Figure 12: Original "girl" image (Left) and initial (erroneous) results (Right)

After repairing the centering issue, we attain results of excellent quality. We vary our gamma low and gamma high values between $[0, 1]$ and $[1, 2]$ respectively. We observe in figure 13 that an increase in gamma low is directly related to brightening dark regions and darkening light regions.

In regards to varying gamma high, we find that increasing values leads to artifacts in the image. This is observed in figure 14 Interestingly, for small values of gamma low, the severity of artifacts is much greater. This can be seen in Figure 15.

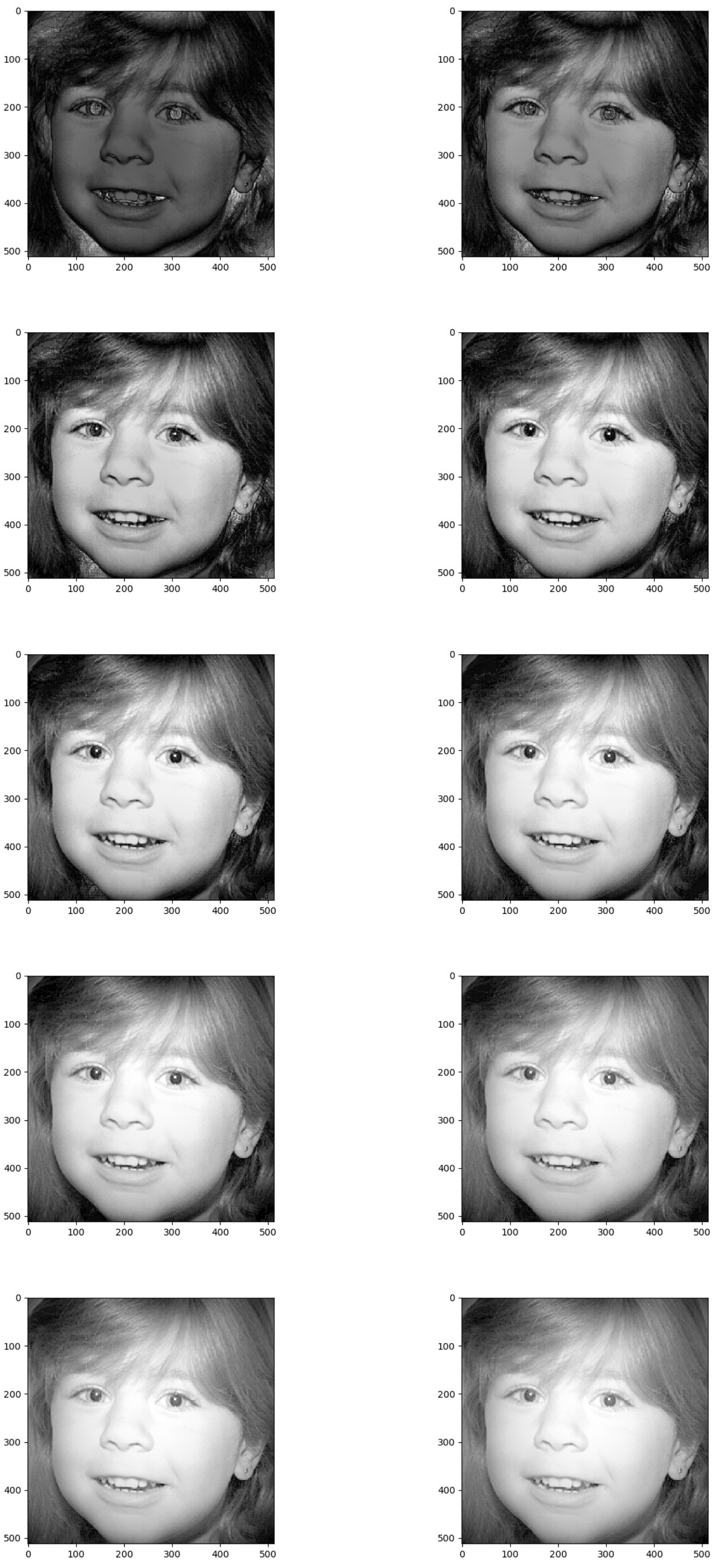


Figure 13: Samples showing results of increasingly large values of gamma low. The value of gamma high is held constant at 1.5 Smallest values appear at the top of the figure and grow from left to right and top to bottom

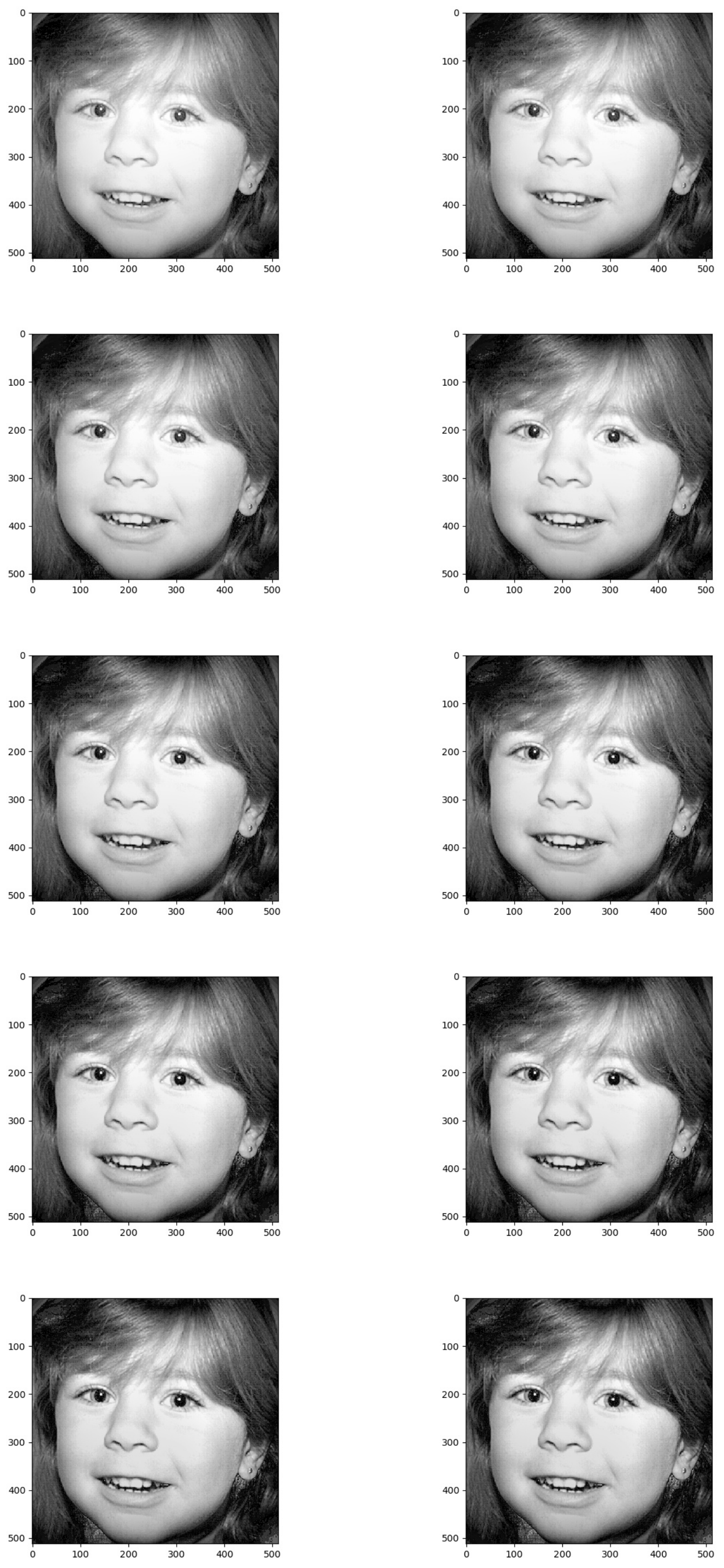


Figure 14: Samples showing results of increasingly large values of gamma high. Gamma low is fixed at .5. Smallest values appear at the top of the figure and grow from left to right and top to bottom

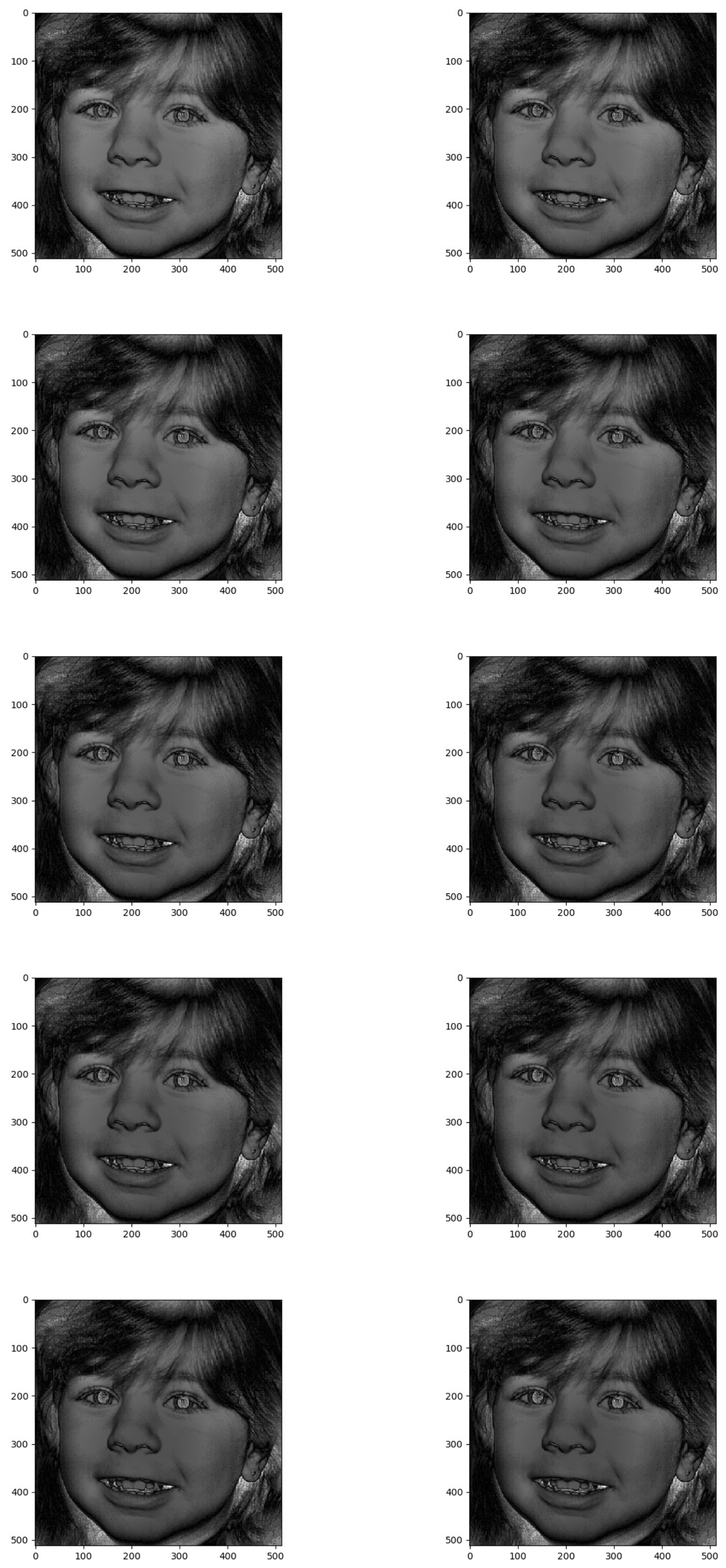


Figure 15: Samples showing the results of increasingly large values of gamma high. Here however, the value of gamma low is fixed at .1, rather than .5

4 Program Listing

4.1 Frequency Filtering Helper Functions

```
def add_zero(data):
    new_data = []
    for x in range(len(data)):
        new_data.append([])
        for y in range(len(data[x])):
            new_data[x].append(data[x][y])
            new_data[x].append(0)
    return new_data

def remove_zeros(data):
    new_data = []
    for i in range(len(data)):
        new_data.append([])
        for j in range(0, len(data[i]), 2):
            new_data[i].append(data[i][j])
    return new_data

def get_magnitude(data, use_log=True, square=False):
    mag = []
    for i in range(len(data)):
        mag.append([])
        for j in range(len(data[i]) // 2):
            if square:
                mag[i].append((data[i][2*j]**2) + (data[i][2*j + 1]**2))
            elif use_log:
                mag[i].append(log(((data[i][2*j]**2)+(data[i][2*j+1]**2))**(1/2)+1))
            else:
                mag[i].append(((data[i][2*j]**2) + (data[i][2*j + 1]**2))**(1/2))

    return mag

def plot_transform_2d(data, use_log=True):
    mag = get_magnitude(data, use_log)

    plt.imshow(mag, label="magnitude", cmap=plt.get_cmap("gray"))
    plt.show()

def center_2d_transform(data):
    new_data = []
    for i in range(len(data)):
        new_data.append([])
        for j in range(len(data[i])):
            new_data[i].append(data[i][j]*(-1)**(i+j))
    return new_data

def remove_centering_2d_transform(data):
    data = remove_zeros(data)
    return center_2d_transform(data)

# Hadamard is just broadcasting (complex mlutiplication)
```

```
def complex_hadamard(a, b):
    result = []
    for i in range(len(a)):
        result.append([])
        for j in range(len(a[0])//2):
            real_part = a[i][2*j]*b[i][2*j] - a[i][2*j+1]*b[i][2*j+1]
            complex_part = a[i][2*j]*b[i][2*j+1] + a[i][2*j+1]*b[i][2*j]
            result[-1].append(real_part)
            result[-1].append(complex_part)

    return result

def pad_zeros(image, filter, pad_value=0):
    n = len(image)
    m = len(filter)

    # Padding must be at least n + m - 1
    min_padding = n + m -1
    # Pad to a power of 2
    padding = 0
    exponent = 1
    while padding < min_padding:
        padding = 2**exponent
        exponent += 1

    for i in range(m,padding):
        filter.append([0 for _ in range(padding)])
    for i in range(m):
        filter[i].extend([0 for _ in range(padding-m)])

    for i in range(n,padding):
        image.append([pad_value for _ in range(padding)])
    for i in range(n):
        image[i].extend([pad_value for _ in range(padding-n)])

    return image, filter, padding - n

def unpad(image, pad_len):
    if pad_len == 0:
        return image
    for i in range(pad_len):
        image[i] = image[i][:pad_len]

    return image[:pad_len]

def normalize_0_255(input_list):
    min_v = min([min(i) for i in input_list])
    max_v = max([max(i) for i in input_list])

    for w in range(len(input_list)):
        for h in range(len(input_list[w])):
            input_list[w][h] = int((input_list[w][h] - min_v)
                                   * (255 / (max_v - min_v)))
    return input_list

def dft2d(data, mode):
    n = len(data)
    if mode is 1:
        data = add_zero(data)

    transform = []
    # dft of rows
    for i in range(len(data)):
        transform.append(fft_func.fourier_transform(data[i], mode, extend=False))

    # multiply by n
    if mode is 1:
        for i in range(len(transform)):
            for j in range(len(transform[i])):
                transform[i][j] *= n

    # get columns
    columns = []
    for x in range(len(transform)):
        columns.append([])
        for y in range(len(transform)):
            columns[x].append(transform[y][x*2])
            columns[x].append(transform[y][x*2+1])

    # transform columns
    for x in range(0, len(columns)):
        columns[x] = fft_func.fourier_transform(columns[x], mode, extend=False)
    # convert columns to rows
    transform = []
    for x in range(len(columns)):
        transform.append([])
        for y in range(len(columns)):
            transform[x].append(columns[y][x*2])
            transform[x].append(columns[y][x*2+1])

    return transform
```

4.2 Noise Removal

```
def list_to_image(image_list):
    height = len(image_list)
    width = len(image_list[0])
    output_image = Image.new("L", (width, height))
    output_image_pixels = output_image.load()
    for i in range(height):
        for j in range(width):
            output_image_pixels[j,i] = int(image_list[i][j])
    return output_image

#takes in the array we are editing (should be in frequency domain), the D0
#is the cutoff frequency that we have to estimate,
# W is the width of the band, and width/height is the original image's width/height
#it returns the edited array with the gaussianbandrejectfilter applied to it
def GaussianBandRejectFiltering(array, D0, W, Width, Height):
    for i in range(Width):
        for j in range(Height):
            #distance to the center of the filter, D(u,v)
            D = sqrt((i - (Width//2))**2 + (j - (Height//2))**2)
            #Gaussian Band Reject filter
            H = 1 - e**(-( (D - D0)**2 / (W)**2 )**2)
            #applyFilter
            array[i][j] = array[i][j] * H
    return array


#takes in the array we are editing (should be in frequency domain), the D0
#is the cutoff frequency that we have to estimate,
# W is the width of the band, and width/height is the original image's width/height
#it returns the edited array with the Gaussian band-pass filter applied to it,
#which is reject's filter, but inversed (1-H_BR) where H_BR is the reject's filter
def GaussianBandPassFiltering(array, D0, W, Width, Height):
    for i in range(Width):
        for j in range(Height):
            # distance to the center of the filter
            D = sqrt((i - (Width//2))**2 + (j - (Height//2))**2)
            # Gaussian Band Reject Filter
            H = 1 - e**(-( (D - D0)**2 / (W)**2 )**2)
            # Filter of band pass is the inverse of the reject filter, 1-H_BR
            H = 1 - H
            # apply filter
            array[i][j] = array[i][j] * H
    return array

def SquareImage(squareSize):
    size = squareSize/2
    size = int(size)
    im = Image.new("L", (512, 512), "black")
    im.paste("white", (128-size,128-size, 128+size,128+size))
    im.paste("white", (256-size, 256-size, 256+size, 256+size))
    return im

#needed to scale back after we filter
def ScaleValuesLinear(img, image_array, Height, Width):
    min_value = 99999999999
    max_value = -99999999999
    # find Min Max values
    for height in range(Height):
        for width in range(Width):
            # find Min Max Values
            if (image_array[height][width] > max_value):
                max_value = image_array[height][width]
            if (image_array[height][width] < min_value):
                min_value = image_array[height][width]

    scaled = 255 / max_value
    scaled = round(scaled, 10)
    for height in range(Height):
        for width in range(Width):
            value_from_array = image_array[height][width]
            value_from_array = int(value_from_array * scaled)
            #easy way to insert scaled pixels back into the image
            img.putpixel((width, height), value_from_array)
            image_array[height][width] = value_from_array

    return img
if noise_removal:
    gaussianMask7 = np.array([[1, 1, 2, 2,  2, 1, 1],
                              [1, 2, 2, 4,  2, 2, 1],
                              [2, 2, 4, 8,  4, 2, 2],
                              [2, 4, 8, 16, 8, 4, 2],
                              [2, 2, 4, 8,  4, 2, 2],
                              [1, 2, 2, 4,  2, 2, 1],
                              [1, 1, 2, 2,  2, 1, 1]], np.float32)
    gaussianMask15 = np.array([[2, 2, 3,  4,  5,  5,  6,  6,  6,  5,  5,  4,  3,
2, 2],
                               [2, 3, 4,  5,  7,  7,  8,  8,  8,  7,  7,  5,  4,  3, 2],
                               [3, 4, 6,  7,  9,  10, 10, 11, 10, 10, 9,  7,  6,  4, 3],
                               [4, 5, 7,  9,  10, 12, 13, 13, 13, 12, 10, 9,  7,  5, 4],
                               [5, 7, 9,  11, 13, 14, 15, 16, 15, 14, 13, 11, 9,  7, 5],
                               [5, 7, 10, 12, 14, 16, 17, 18, 17, 16, 14, 12, 10, 7, 5],
                               [6, 8, 10, 13, 15, 17, 19, 19, 19, 17, 15, 13, 10, 8, 6],
                               [6, 8, 11, 13, 16, 18, 19, 20, 19, 18, 16, 13, 11, 8, 6],
                               [6, 8, 10, 13, 15, 17, 19, 19, 19, 17, 15, 13, 10, 8, 6],
                               [5, 7, 10, 12, 14, 16, 17, 18, 17, 16, 14, 12, 10, 7, 5],
                               [5, 7, 9,  11, 13, 14, 15, 16, 15, 14, 13, 11, 9,  7, 5],
                               [4, 5, 7,  9,  10, 12, 13, 13, 13, 12, 10, 9,  7,  5, 4],
                               [3, 4, 6,  7,  9,  10, 10, 11, 10, 10, 9,  7,  6,  4, 3],
                               [2, 3, 4,  5,  7,  7,  8,  8,  8,  7,  7,  5,  4,  3, 2],
                               [2, 2, 3,  4,  5,  5,  6,  6,  6,  5,  5,  4,  3,  2, 2]], np.fl

    image = Image.open("boy_noisy.pgm")
    pixels = list(image.getdata())
    width, height = image.size
    newImage = Image.new("L", (width, height))

    # # N = height x width
    N = (newImage.size[0] + newImage.size[1]) // 4
    pixels = normalizeImage(pixels, height, width, N)

    pixels = np.fft.fft2(pixels)

    pixels = GBRF(pixels.real, 72, 6, width, height, False)

    pixels = np.fft.ifft2(pixels)

    pixels = np.array(pixels.real)
    pixelImage = Image.fromarray(pixels.real, 'L')
    newImage = ScaleValuesLinear(pixelImage, pixels.real, height, width)
```

4.3 Frequency Domain Convolution

```
def convolve(image, convolution, padding=0):
    width, height = len(image[0]), len(image)
    conv_size = len(convolution)

    new_img = []

    for i in range(width):
        new_img.append([])
        for j in range(height):
            weighted_sum = 0
            for n in range(conv_size):
                for k in range(conv_size):
                    v_offset = n - conv_size // 2
                    h_offset = k - conv_size // 2
                    if i + h_offset < 0 or i + h_offset >= width\
                                        or j + v_offset < 0\
                                        or j + v_offset >= height:
                        weighted_sum += padding
                    else:
                        weighted_sum += image[i+h_offset][j+v_offset]*
                        convolution[n][k]
            new_img[i].append(int(weighted_sum))

    return new_img

sobel = [[-1,0,1],[-2,0,2],[-1,0,1]]

lenna = get_image_list("/home/nthom/Documents/CS674/images-pgm/lenna.pgm")

spatial_filtered_image = convolve(lenna, sobel)

sobel = [[0,0,0,0],[0,-1,0,1],[0,-2,0,2],[0,-1,0,1]]
lenna, filter, pad_len = pad_zeros(lenna, sobel)

centered_nate = center_2d_transform(lenna)
centered_filter = center_2d_transform(sobel)

image_transform = dft2d(centered_nate, 1)
filter_transform = dft2d(centered_filter, 1)

filtered_image_freq = complex_hadamard(image_transform, filter_transform)

image_inverse_transform = dft2d(filtered_image_freq, -1)

uncentered = remove_centering_2d_transform(image_inverse_transform)

filtered_image = unpad(uncentered, pad_len)
```

4.4 Motion Blur

```python
def complex_division(a, b, cutoff=30):
    result = []
    for i in range(len(a)):
        result.append([])
        for j in range(len(a[0])//2):
            if ((j-len(a)//2)**2 + (i-len(a)//2)**2)**(1/2) > cutoff:
                result[-1].append(a[i][2*j])
                result[-1].append(a[i][2*j+1])
                continue
            real_part = (a[i][2*j]*b[i][2*j] + a[i][2*j+1]*b[i][2*j+1]) \
                        / (b[i][2*j]**2 + b[i][2*j+1]**2)
            complex_part = (a[i][2*j+1]*b[i][2*j] - a[i][2*j]*b[i][2*j+1]) \
                           / (b[i][2*j]**2 + b[i][2*j+1]**2)
            result[-1].append(real_part)
            result[-1].append(complex_part)

    return result

def complex_invert(matrix, cutoff=30):
    result = []
    for i in range(len(matrix)):
        result.append([])
        for j in range(len(matrix[0])//2):
            if ((j-len(matrix)//2)**2 + (i-len(matrix)//2)**2)**(1/2) > cutoff:
                result[-1].append(1)
                result[-1].append(1)
                continue
            real_part = matrix[i][2*j]/(matrix[i][2*j]**2+matrix[i][2*j+1]**2)
            complex_part = -matrix[i][2*j+1]/(matrix[i][2*j]**2+matrix[i][2*j+1]**2)
            result[-1].append(real_part)
            result[-1].append(complex_part)

    return result

def complex_square(matrix):
    result = []
    for i in range(len(matrix)):
        result.append([])
        for j in range(len(matrix[0])//2):
            real_part = matrix[i][2*j]**2 - matrix[i][2*j+1]**2
            complex_part = 2*matrix[i][2*j]*matrix[i][2*j+1]
            result[-1].append(real_part)
            result[-1].append(complex_part)

    return result

def add_scalar(matrix, scalar):
    result = []
    for i in range(len(matrix)):
        result.append([])
        for j in range(len(matrix[0])//2):
            real_part = matrix[i][2*j] + scalar
            complex_part = matrix[i][2*j+1]
            result[-1].append(real_part)
            result[-1].append(complex_part)

    return result

def motion_blur_filter(T, a, b, size):
    p1 = lambda u,v: T/(pi*(u*a + v*b + 0.00001))
    real_p = lambda u,v: p1(u,v)*1/2*sin(2*pi*(u*a + v*b + 0.00001))
    imaginary_p = lambda u,v: -p1(u,v)*sin(pi*(u*a + v*b + 0.00001))**2

    filter = []
    for i in range(-size//2, size//2):
        filter.append([])
        for j in range(-size//2, size//2):
            real_part = real_p(i,j)
            im_part = imaginary_p(i,j)
            if real_part > 0:
                filter[-1].append(max(real_part, .0005))
            else:
                filter[-1].append(min(real_part, -.0005))
            if im_part > 0:
                filter[-1].append(max(im_part, .0005))
            else:
                filter[-1].append(min(im_part, -.0005))

    return filter

def normalize_2d(input_list):
    list_sum = 0
    for i in input_list:
        list_sum += sum(i)

    output_list = [[float(input_list[i][j])/list_sum
                    for j in range(len(input_list[i]))]
                   for i in range(len(input_list))]

    return output_list

def add_noise(matrix, mu, sigma):
    result = []
    for i in range(len(matrix)):
        result.append([])
        for j in range(len(matrix[0])):
            result[-1].append(max(min(int(matrix[i][j]
                                      + np.random.normal(mu,sigma)), 255),0))

    return result

if apply_motion_blur:
    nate = get_image_list("lenna.pgm")
    noisy_nate = add_noise(nate, 0, 100)
    centered_nate = center_2d_transform(noisy_nate)
    image_transform = dft2d(centered_nate, 1)

    filter = motion_blur_filter(1, .1, .1, 256)
    filtered_image = complex_hadamard(image_transform, filter)

    filtered_nate = dft2d(filtered_image, -1)

    new_nate = remove_centering_2d_transform(filtered_nate)

    nate_out = list_to_image(normalize_0_255(new_nate))
    nate_out.save("lenna-motion.jpg")

    noisy_nate_out = list_to_image(noisy_nate)
    noisy_nate_out.save("lenna-noisy.jpg")

if remove_motion_blur:
    nate = get_image_list("lenna-motion.jpg")
    centered_nate = center_2d_transform(nate)
    image_transform = dft2d(centered_nate, 1)

    filter = motion_blur_filter(1, .1, .1, 256)

    for cutoff in [10, 30, 50]:
        invert_filter = complex_invert(filter, cutoff=cutoff)

        filtered_image = complex_hadamard(image_transform, invert_filter)

        filtered_nate = dft2d(filtered_image, -1)

        new_nate = remove_centering_2d_transform(filtered_nate)

        nate_out = list_to_image(normalize_0_255(new_nate))
        nate_out.save("lenna-restore-{}.jpg".format(cutoff))

if remove_motion_blur_wiener:
    nate = get_image_list("lenna-motion.jpg")
    centered_nate = center_2d_transform(nate)
    image_transform = dft2d(centered_nate, 1)

    filter = motion_blur_filter(1, .1, .1, 256)
    filter_square = add_zero(get_magnitude(filter, square=True))

    for k in [1, .1, .01, .001, .0001, 0]:
        noise_term = complex_division(filter_square,
                                      add_scalar(filter_square, k), cutoff=500)
        inverse_filtered = complex_division(image_transform, filter, cutoff=500)
        filtered_image = complex_hadamard(noise_term, inverse_filtered)

        filtered_nate = dft2d(filtered_image, -1)

        new_nate = remove_centering_2d_transform(filtered_nate)

        nate_out = list_to_image(normalize_0_255(new_nate))
        nate_out.save("lenna-restore-wiener-{}.jpg".format(k))
```

4.5 Homomorphic Filtering

```
girl = get_image_list("girl.gif")
ln_girl = []
for i in range(len(girl)):
    ln_girl.append([])
    for j in range(len(girl[i])):
        ln_girl[i].append(log(girl[i][j] + 1))
centered_girl = center_2d_transform(ln_girl)
transformed_girl = dft2d(centered_girl, 1)
cuttoff_frequency = 1.8
c = 1
for lambda_low in range(11):
    for lambda_high in range(11, 21):
        filter = []
        for i in range(-len(transformed_girl)//2, len(transformed_girl)//2+1):
            filter.append([])
            for j in range(-len(transformed_girl)//2, len(transformed_girl)/
            /2+1):
                lambda_difference = lambda_high - lambda_low
                spectrum_position_sum = i**2 + j**2
                filter_term = 1 - (exp(-c * (spectrum_position_sum /
                cuttoff_frequency ** 2)))
                filter[-1].append((lambda_difference * filter_term) +
                lambda_low)
                filter[-1].append(0)
        print(len(transformed_girl), len(transformed_girl[0]))
        print(len(filter), len(filter[0]))
        filtered_girl = complex_hadamard(transformed_girl, filter)
        inverse_transformed_girl = dft2d(filtered_girl, -1)
        uncentered_girl = get_magnitude(inverse_transformed_girl)
        output_girl = []
        for i in range(len(uncentered_girl)):
            output_girl.append([])
            for j in range(len(uncentered_girl[i])):
                output_girl[i].append(exp(uncentered_girl[i][j] - 1))
```