Deev Patel & Wei Tong

CS 474: Image Processing & Interpretation

Monday, December 3, 2018

Programming Assignment # 4

Frequency Domain Filtering

Experiments

(1) Noise Removal

(3) Image Restoration

(4) Homomorphic Filtering

Division of Work

Experiment 1 and 3, of both the code and the report, was done by Deev. Experiment 4, of both the code and the report, was done by Wei. Additionally, both helped in implementing the common parts of the code and debugging. We believe this was close to an equal division of work.

Technical Discussion

(1) Noise Removal

The ability to remove noise is of vital importance to the field of image processing. Being able to clear images of noise is necessary for any further processing to be effective and efficient. However, the task of noise removal can be challenging because noise can come in various forms, and not all of them are well-defined. That being said, the two main mediums of noise removal are spatial domain filtering and frequency domain filtering. As the major models of spatial domain filtering were already discussed in Programming Assignment 2, in this assignment we will look at frequency domain filtering. In order for frequency domain filtering to occur, the Fourier Transform must be used to decompose an image into its fundamental frequencies. Since the basics of the Fourier Transform and its 2-D implementation were examined in Programming Assignment 3, in this experiment, we will build upon this and consider the Fourier Transform's practical implications for noise removal.

While the exact structure of noise is often unknown and random, we can use certain properties and characteristics of images and the image generation process to remove attributes likely to be noise. As a general trend, high frequencies are not as common as low frequencies in natural images. Moreover, noise is more likely to appear at higher frequencies. Thus, we can often assume certain structured high frequencies to be associated with noise and remove them. This can be accomplished by looking at the spectrum of the image and identifying areas of "unnaturally" high frequencies. Then, these frequencies can be attenuated by decreasing the weights (both real and imaginary). Essentially, we can define a band-reject filter to remove frequencies likely to be

noise. This can be powerful because noise caused by image generation processes (e.g. camera mechanics) often corresponds to specific frequency patterns in the spectrum.

(3) Image Restoration - Degradation model and noise

In addition to removing noise, we can devise filtering methodologies that seek to reverse some kind of modelable degradation process. A classic and useful example of this involves the ability to reverse motion blur caused by uniform, planar motion of the camera. We can accomplish this in the ideal case by mathematically modeling the degraded image created by a camera moving with a trajectory of $<x_0(t), y_0(t)>$ during the exposure time period of [0, T]. Since the captured pixels values, g(*x,y*), will just be the accumulation over time of the actual pixel values, f(*x,y*), where the camera is pointing, we write (Eq 1) below.

$$g(x,y) = \int_0^T f\left(x - x_0(t),\ y - y_0(t)\right) \mathrm{d}t \qquad (\text{Eq } 1)$$

Then, we can take the FFT and do a change of variables by letting $w_1 = x - x_0(t)$ and $w_2 = y - y_0(t)$.

$$\mathcal{F}\{g(x,y)\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_0^T f\left(x - x_0(t),\ y - y_0(t)\right) e^{-j2\pi(ux+vy)}\,\mathrm{d}t\,\mathrm{d}y\,\mathrm{d}x$$

$$\mathcal{F}\{g(x,y)\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_0^T f\left(w_1,\ w_2\right) e^{-j2\pi\left(u\left(w_1 + x_0(t)\right) + v\left(w_2 + y_0(t)\right)\right)}\,\mathrm{d}t\,\mathrm{d}w_2\,\mathrm{d}w_1$$

$$\mathcal{F}\{g(x,y)\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f\left(w_1,\ w_2\right) e^{-j2\pi\left(uw_1 + vw_2\right)}\,\mathrm{d}w_2\,\mathrm{d}w_1 \int_0^T e^{-j2\pi\left(ux_0(t) + vy_0(t)\right)}\,\mathrm{d}t$$

$$\mathcal{F}\{g(x,y)\} = \mathcal{F}\{f(x,y)\} \int_0^T e^{-j2\pi\left(ux_0(t) + vy_0(t)\right)}\,\mathrm{d}t$$

Finally, we can define the integral on the right side as H(*u, v*) and obtain (Eq. 2)

$$G(u,v) = F(u,v) \int_0^T e^{-j2\pi\left(\mathrm{ux}_0(t) + \mathrm{vy}_0(t)\right)} dt = F(u,v)\, H(u,v) \quad (\text{Eq } 2)$$

From (Eq. 2), we can solve for H(*u, v*) by applying the constraint of uniform planar motion. In other words, we can let $x_0(t) = \alpha t/T$ and $y_o(t) = \beta t/T$ for some constants α, β.

$$H(u,\ v) = \int_0^T e^{-j2\pi\left(\frac{u\alpha t}{T} + \frac{v\beta t}{T}\right)} dt = \int_0^T e^{\frac{-j2\pi t}{T}(u\alpha + v\beta)}\, dt = \left[\frac{e^{\frac{-j2\pi t(u\alpha + v\beta)}{T}}}{\frac{-j2\pi(u\alpha + v\beta)}{T}}\right]_0^T$$

$$\frac{T}{-j2\pi(\mathrm{u}\alpha + \mathrm{v}\beta)}\left[e^{\frac{-j2\pi t(u\alpha + v\beta)}{T}}\right]_0^T = \frac{T}{-j2\pi(\mathrm{u}\alpha + \mathrm{v}\beta)}\left[e^{-j2\pi(u\alpha + v\beta)} - 1\right]$$

$$\frac{Te^{-j\pi(\mathrm{u}\alpha + \mathrm{v}\beta)}}{-j2\pi(\mathrm{u}\alpha + \mathrm{v}\beta)}\left[e^{-j\pi(\mathrm{u}\alpha + \mathrm{v}\beta)} - e^{j\pi(u\alpha + v\beta)}\right] = \frac{Te^{-j\pi(u\alpha + v\beta)}}{\pi(\mathrm{u}\alpha + \mathrm{v}\beta)}\left[\frac{1}{2j}\left(e^{j\pi(\mathrm{u}\alpha + \mathrm{v}\beta)} - e^{-j\pi(u\alpha + v\beta)}\right)\right]$$

Finally, we can obtain (Eq. 3) by using Euler's formula for complex exponentials.

$$H(u,v) = \frac{T}{\pi(\mathrm{u}\alpha + \mathrm{v}\beta)} e^{-j\pi(u\alpha + v\beta)} \sin(\pi(\mathrm{u}\alpha + \mathrm{v}\beta)) \quad (\text{Eq } 3)$$

With (Eq 3), we now have a usable form of the ideal degradation model for linear, planar motion of the camera. Now, we can use this to try and reverse the degradation of images that are believed to be corrupted by camera motion. However, in order to do this, we must consider that real corrupted images are also affected by some kind of noise. This can be accomplished by modeling the noise as additive noise in the frequency domain, as shown in (Eq 4) below. Note that *G* is the FFT of the degraded image, *F* is the FFT of the ground truth image, *H* is the degradation model/function and *N* is the FFT of any additive noise.

$$G(u,v) = F(u,v)\, H(u,v) + N(u,v) \quad (\text{Eq } 4)$$

With this equation and the degradation model from (Eq 3), we can employ various restoration methodologies to try and obtain an estimate of the original image.

(3a) Inverse Filtering

Since we do not always know much about the noise function from (Eq 4), the simplest way to estimate the original image is to ignore the noise and divide the degraded image by the degradation function. This processes is shown in (Eq 5) and is the backbone of inverse filtering.

$$\widehat{F}(u,v) = \frac{G(u,v)}{H(u,v)} \quad (\text{Eq } 5)$$

However, when this methodology is considered alongside (Eq 4), we see that the estimation is not always accurate.

$$\widehat{F}(u,v) = \frac{F(u,v)\,H(u,v) + N(u,v)}{H(u,v)} = F(u,v) + \frac{N(u,v)}{H(u,v)}$$

If we substitute in for *G(u, v)* from (Eq 4), it becomes clear that for small values of *H*, the noise function has a very large impact on the estimation. Because of this, inverse filtering in its raw form is very unreliable at restoring degraded images.

We can make inverse filtering work much better by selectively ignoring places where the degradation function has small values. This way, the noise function is prevented from overpowering the term representing the original image. We can accomplish this by either setting a threshold for the degradation function so that small values are ignored all together, or by only considering the frequencies in a certain radius around the origin. The radius method works because most degradation models have small values at frequencies far away from the origin. So, only considering an area in the center of the spectrum reduces the likelihood of encountering small values of *H.* As the radius method is easier to implement and more common in practice, that is the method we will consider in this report.

(3b) Wiener Filtering

Wiener filtering is a restoration methodology that seeks to improve upon inverse filtering by taking the noise function into consideration instead of just ignoring it. At a high level, the idea behind Wiener filtering is to assume the image and noise to be random variables and minimize the mean square error between the original image and the estimated image in the spatial domain. Then, we can solve for the minimum in the frequency domain by making a few reasonable assumptions to reduce the complexity of the problem — (1) the noise and image are uncorrelated, (2) one or the other has a mean of zero, and (3) the intensity levels in the estimated image are a linear function of the levels in the original image. Ultimately, the minimum in the frequency domain turns out to be (Eq 6), where $|H|^2$ is the power spectrum of the degradation function, S_n is the power spectrum of the noise ($|N|^2$) and S_f is the power spectrum of the original, undegraded image.

$$\widehat{F}(u,v) = \left[\frac{1}{H(u,v)} \frac{|H(u,v)|^2}{|H(u,v)|^2 + \frac{S_n(u,v)}{S_f(u,v)}} \right] G(u,v) \quad \text{(Eq 6)}$$

While (Eq 6) is theoretically correct, it is practically hard to use in its basic form because we rarely know much about the noise, much less the undegraded image. Essentially, the term S_n/S_f is difficult to compute exactly. But, since all we are after is a good estimate of the original image, we can just treat S_n/S_f as a constant. This gives us (Eq 7), the usable form of Wiener filtering, in which K is chosen manually to obtain the best restored image.

$$\widehat{F}(u,v) \approx \left[\frac{1}{H(u,v)} \frac{|H(u,v)|^2}{|H(u,v)|^2 + K} \right] G(u,v) \quad \text{(Eq 7)}$$

(4) Homomorphic Filtering

Sometimes, images have shading problems that make it impossible to discern details that would otherwise be shown. Homomorphic filtering is the most commonly used technique to try and fix images with non-uniform illumination. It attempts to even out the illumination so that details that would otherwise be hidden by the darkness can be seen. This is done by modeling the the pixel values of an image as a simple product of the illumination and reflectance of the objects in the scene. Essentially, we model the image f(*x,y*) in the spatial domain as shown in (Eq 8), where i(*x,y*) is the illumination component and r(*x,y*) is the reflectance component.

$$f(x,y) = i(x,y)\, r(x,y) \quad \text{(Eq 8)}$$

Now, we can separate the illumination and reflectance components of the image into a sum by applying a natural log function to both sides. Then, we can take the Fourier Transform of the resulting image. This transforms the multiplication of the illumination and reflectance components into a linear addition.

$$\mathfrak{F}\{\ln(f(x,y))\} = \mathfrak{F}\{\ln(i(x,y))\} + \mathfrak{F}\{\ln(r(x,y))\}$$

Essentially, the contribution of the illumination and reflectance components has been seperated into a sum in the frequency domain. This allows any multiplicative filter, H(*u,v*), to work on the two components separately. Now, we can take advantage of this by noticing that the reflectance component varies a lot and thus consists mostly of high frequencies. Meanwhile, the illumination component varies slowly and thus consists mostly of low frequencies. So, if we apply a high pass filter then the contribution of the illumination will be attenuated while the contribution of the reflectance will be amplified. The specific highpass filter used in this

assignment is given in (Eq 9). Note that γ_L and γ_H are the gains for the low and high frequencies respectively, D_0 is the cutoff frequency, and c is constant set to 1.

$$H(u,v) = (\gamma_H + \gamma_L)\left[1 - e^{-c\left(\frac{u^2+v^2}{{D_0}^2}\right)}\right] + \gamma_L \quad (\text{Eq } 9)$$

Once a high pass filter is applied, we can obtained the filtered image in the spatial domain by taking the FFT and undoing the application of the natural log function via exponentiation. This allows us to view more details in dark areas by alleviating problems caused by uneven illumination.

Implementation & Results

General Implementation Notes: The ImageComplex Class

In order to make things easier, a class called ImageComplex was created. This class was inspired by the given ImageType class, but implemented a more concrete set of functionalities. Specifically, the class was designed for complex images and stored pixel values as floating point values instead of integers. It implemented many of functionalities needed repeatedly throughout the various experiments. Notably, it included the ability to compute the FFT (by having an interface to the *fft2D* function given in Programming Assignment 3) and perform complex multiplication/division. The ImageComplex class was also designed to make switching between it and the given ImageType class easy. So it provided, a way to get to the real and imaginary parts as well as the spectrum as normalized ImageType variables. This made saving and exporting the images relatively simple, something that was essential for debugging and creating the images in this report. The entire class and its set of functionalities can be seen in the *Programs Listings* section.

(1) Noise Removal

To accomplish frequency domain filtering on the given "boy_noisy" image, a band-reject filter was defined to remove certain frequencies in the spectrum. In terms of implementation, this amounted to obtaining the spectrum as a ImageComplex type (through the forward FFT), making certain pixel values zero, and applying the inverse FFT to obtain the final image. The results and process of filtering the "boy_noisy" image can be seen in Fig. 1 and Fig. 2.

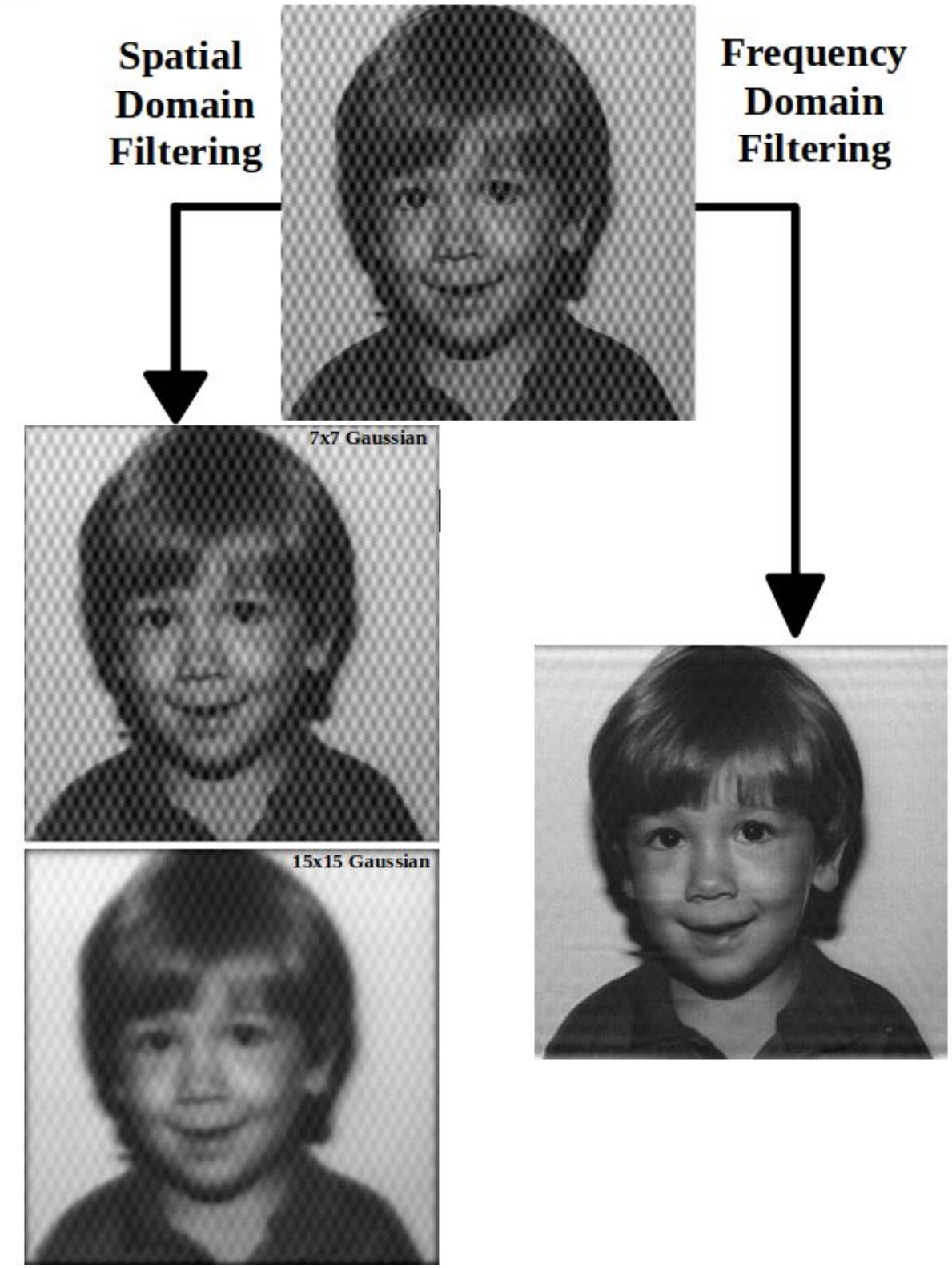

Figure 1: A comparison of spatial domain filtering (through Gaussian smoothing) and frequency domain filtering (through a modification of the spectrum) on the "boy_noisy" image.

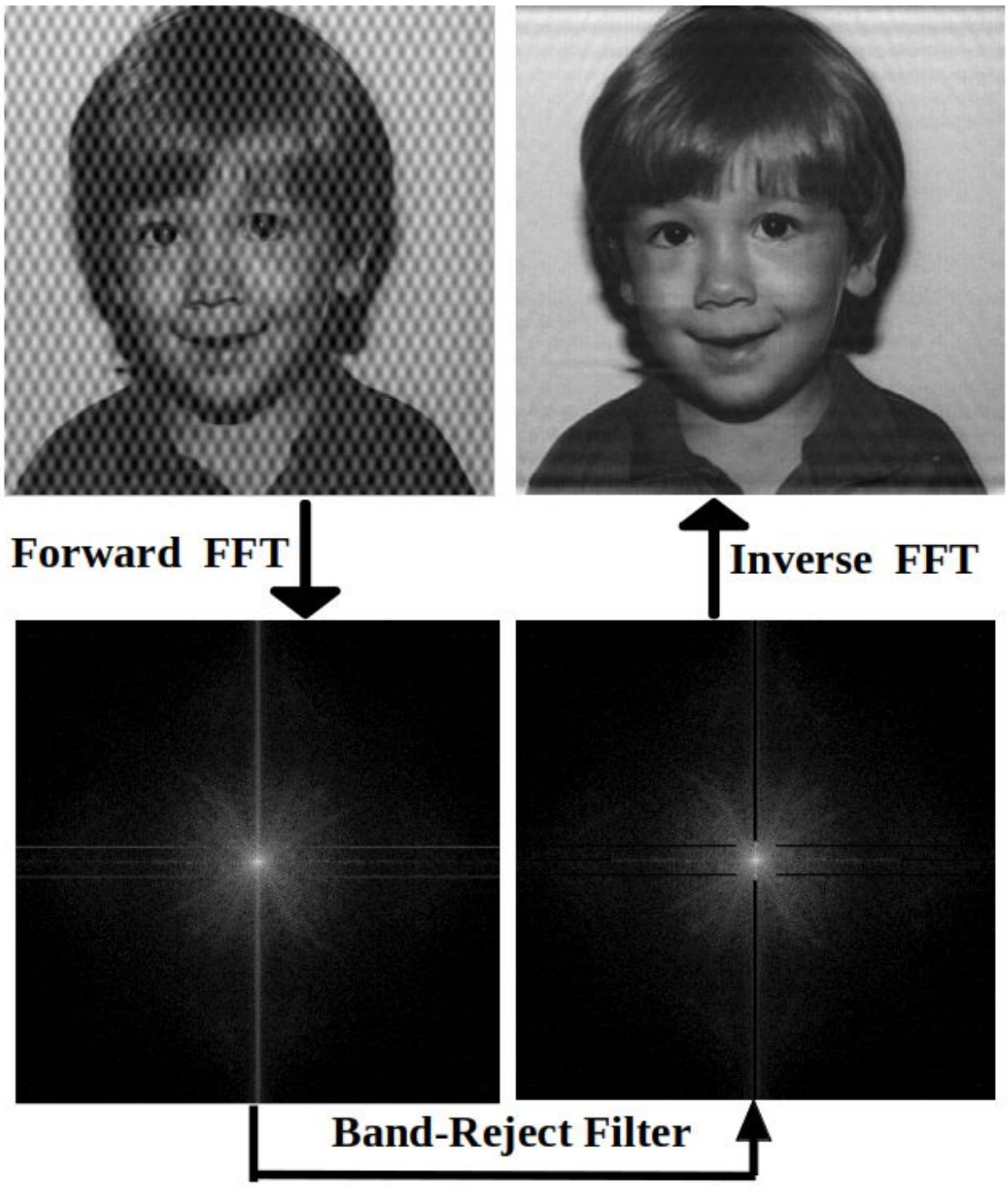

Figure 2: The process through which frequency domain filtering was accomplished on the "boy_noisy" image. Notably, one can see how parts of spectrum (bottom left image) were sharply attenuated through a band-reject filter.

(3) Image Restoration - Degradation model and noise

Before actually implementing inverse and wiener filtering, the degradation model from (Eq 3) was implemented and tested. This mainly involved creating a ImageComplex variable by sampling the function from (Eq 3) and using complex multiplication to multiply the FFT of the original image with the model to obtain the blurred image. An issue encountered during the process involved sampling at certain, special values. For instance, at frequency (0, 0), the sampling had to be manually made T+0j because even though (Eq 3) is undefined at this point, that is the limit of the function. Additionally, the sine and cosine values of certain angles had to be hard coded in. This was done to avoid the std::sin and std::cos functions from returning really small values instead of 0 at angles like 2π. This was necessary because it would have caused more issues later on, especially in regards to inverse filtering. Fig. 3 and 4 show the results of the implementation of the degradation model.

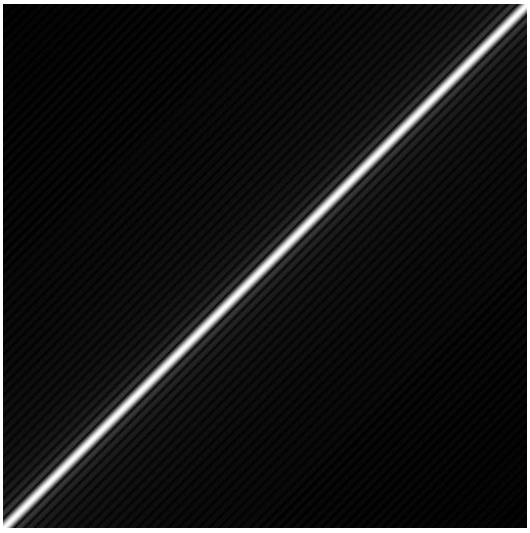

Figure 3: The spectrum of the centered degradation model for uniform, planar motion of the camera (Eq 3) with $\alpha = \beta = 0.1$ and $T = 1$. Note that positive u is rightward and positive v is downward.

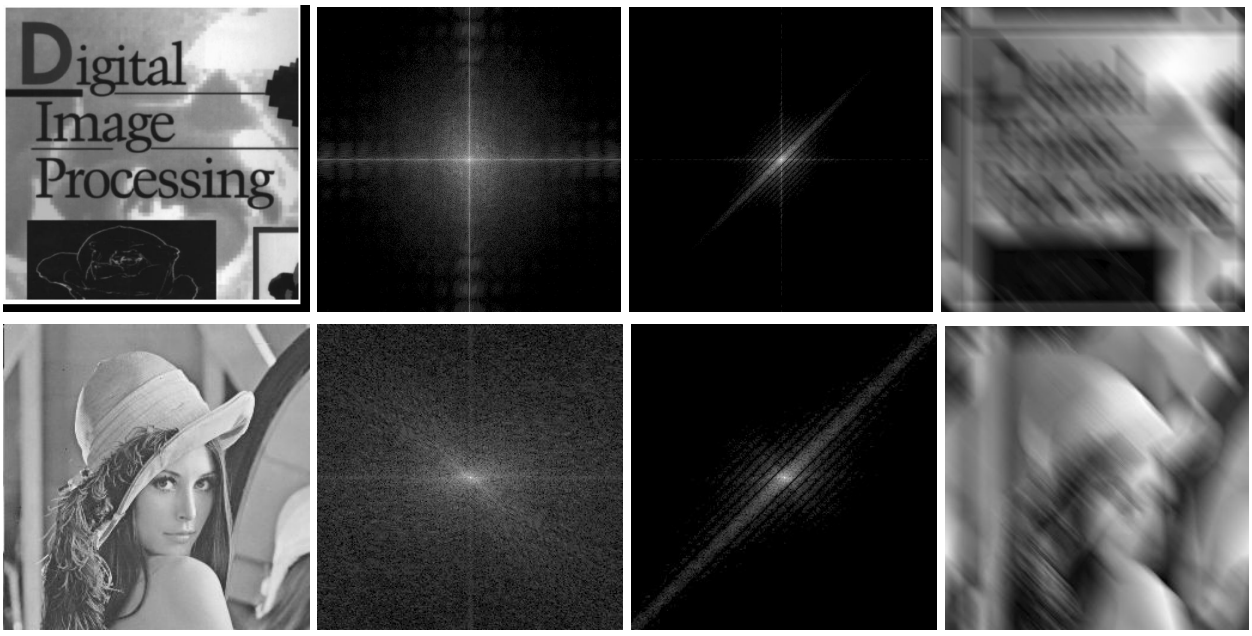

Figure 4: The application of the degradation model for uniform, planar motion of the camera (Eq 3) with $\alpha = \beta = 0.1$ and $T = 1$ on the "book" (top) and "lenna" (bottom) images. From left the right: original image in spatial domain, original image spectrum after FFT, blurred image spectrum after complex multiplication, blurred image in spatial domain after inverse FFT.

Once the degradation model was created the ability to add Gaussian noise was implemented. This involved using the given *box_muller()* function to create noise values in the spatial domain and then adding the noise, according the (Eq 4), in the frequency domain. Various combinations of noise with the degradation model are show in Fig. 5 on the next page. Note that some of the recommended values in the instructions (specifically $\mu=0$, $\sigma=1000$) produced results that were too degraded to perform restoration on. So we only attempted to restore the images with $\sigma=3$, 7, 10, & 100.

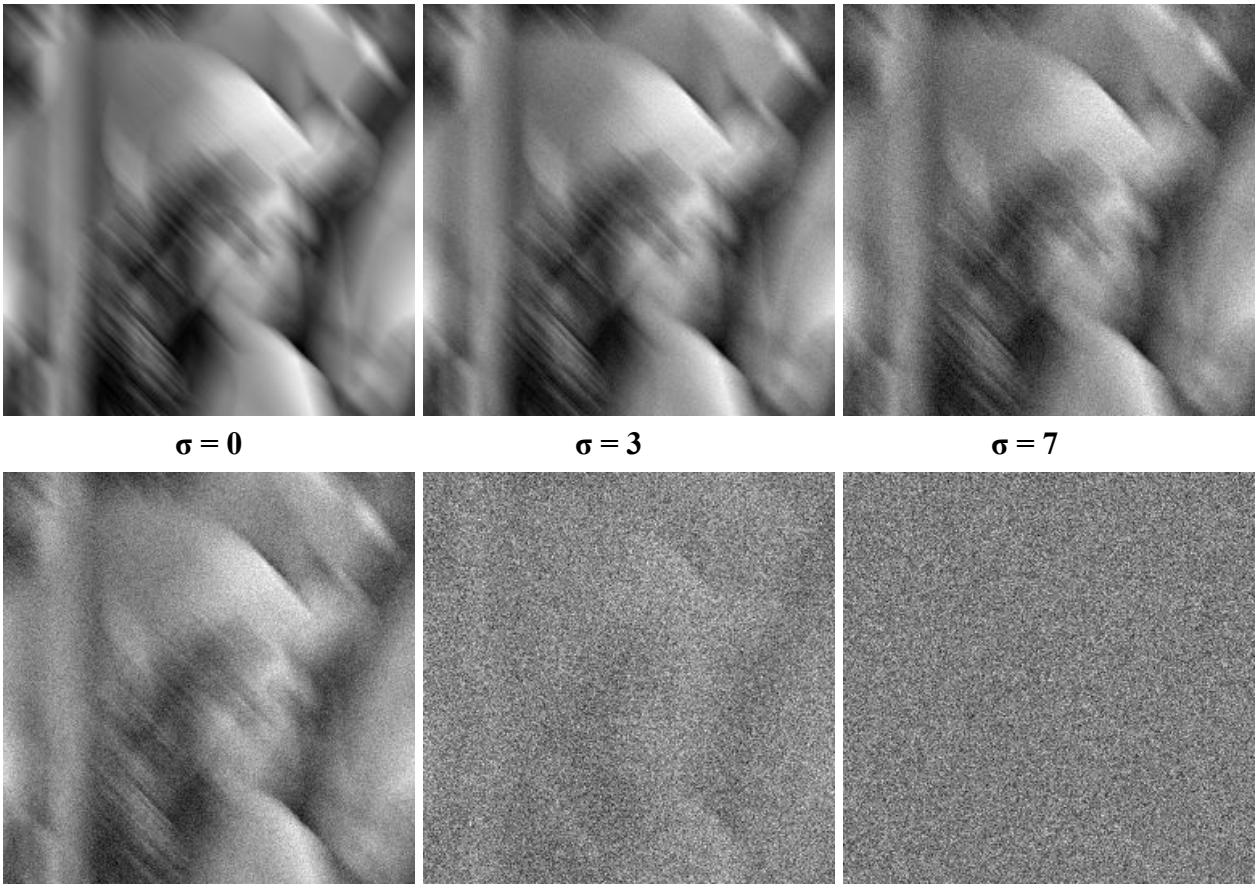

Figure 5: Combination of degradation model with Gaussian noise on "lenna" image. All added noise had μ=0. On top left is the degraded image with no noise added.

(3a) Inverse Filtering

The implementation for inverse filtering was relatively straightforward once the FFT of the blurred image and the degradation function were obtained. In order to apply (Eq 5), the FFT of the original image was multiplied (point-by-point) by the complex inverse of the degradation function. The only difficult part of this involved finding a way to ignore areas where the degradation function had low values. This was done through the radius method. So, various radii had to be manually tested to obtain good results. The results of inverse filtering can be seen in Fig. 6.

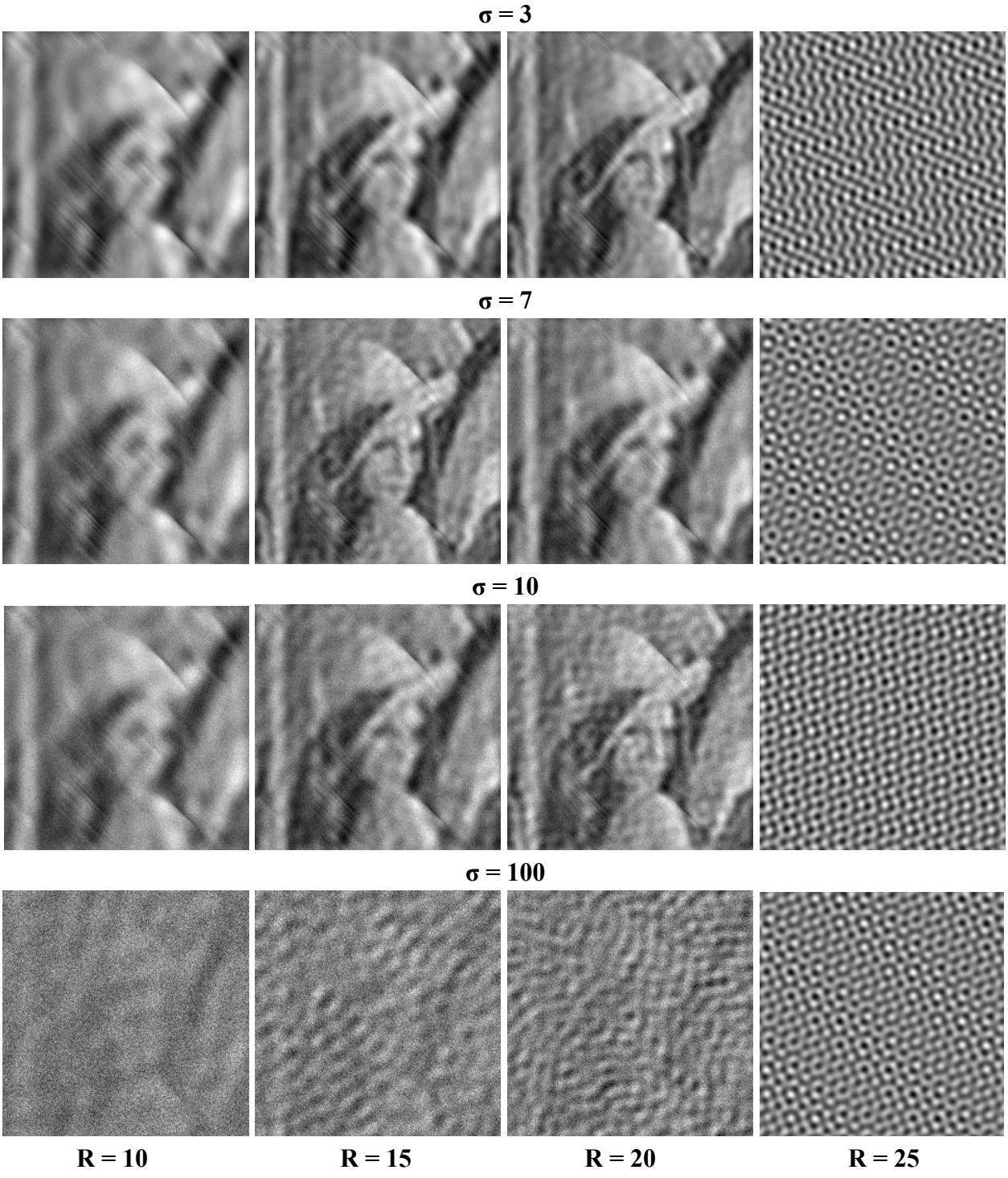

Figure 6: Result of inverse filtering on "lenna" image. Each row shows the restoration process on a specific degraded image (from Fig. 5) containing Gaussian noise (σ of noise is shown above each row). Each column shows images restored with a specific cutoff radius (as given by R below each column).

(3b) Wiener Filtering

The only extra thing that had to be added in order to perform Wiener filtering, was the ability to compute the power spectrum of images. This ability was added to the ImageComplex class and involved zeroing out the imaginary part and setting the real part at each pixel value to the square of the magnitude. Next, (Eq 7) was implemented in various steps, as can be seen in the code, located in section (3b) of the *Program Listings*. Finally, various values of K were experimented with to obtain the best results. The outcome of Wiener filtering is shown in Fig. 7 and Fig. 8. A comparison of Wiener filtering and inverse filtering is shown in Fig. 9.

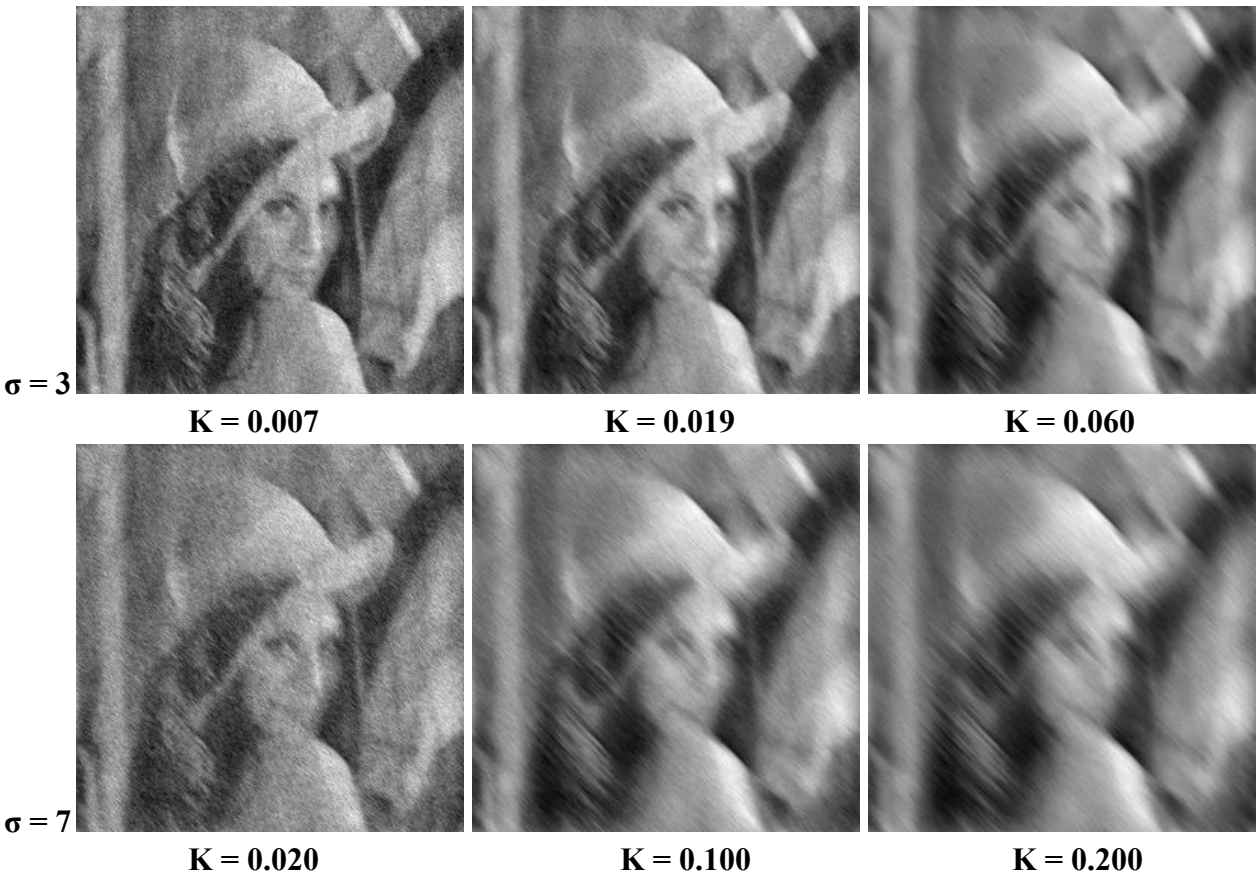

Figure 7: Result of Wiener filtering on "lenna" image. The first row shows the restoration process on the degraded image (from Fig. 5) with Gaussian noise of $\sigma = 3$. Similarly, the second row shows restoration with $\sigma = 7$. The K value (defined in Eq 7) used is shown below each image.

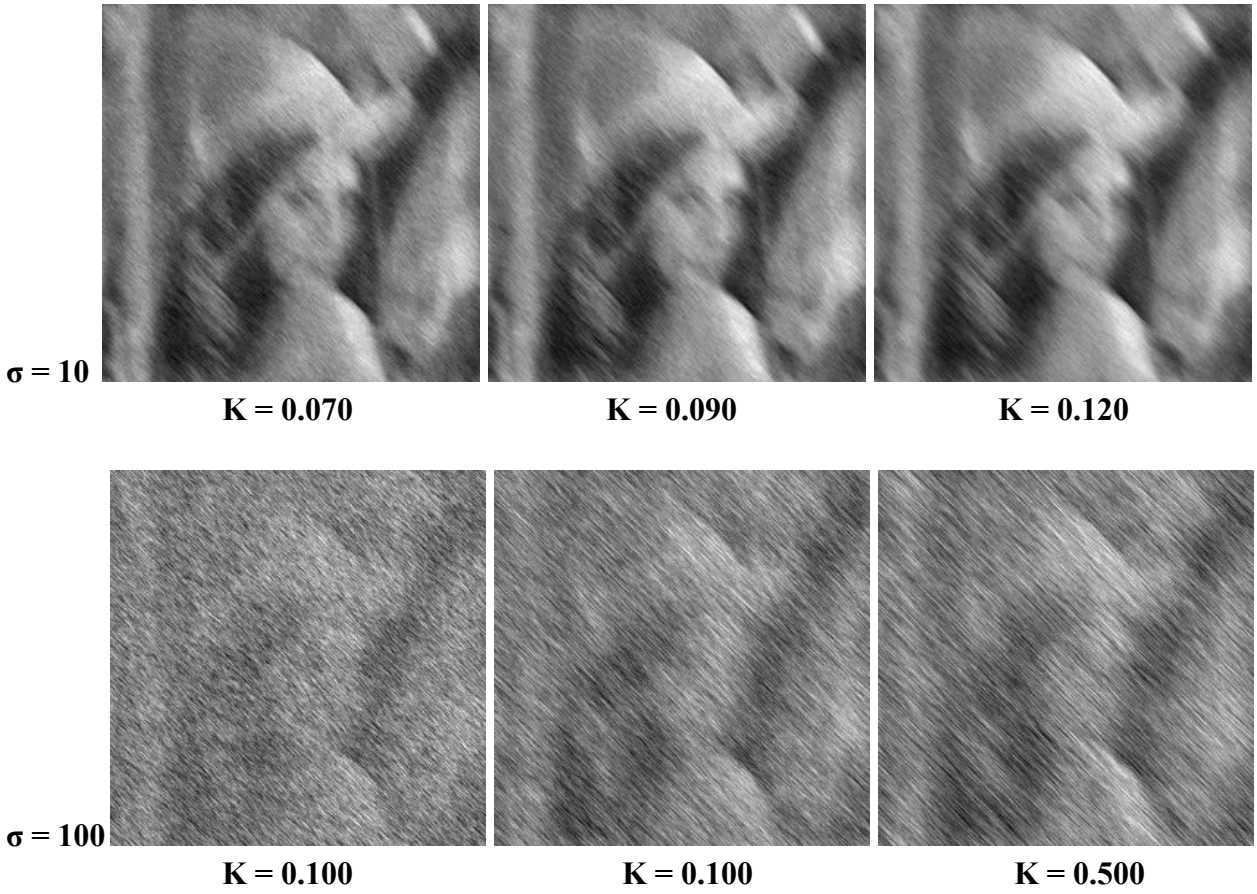

Figure 8: Result of Wiener filtering on "lenna" image. The first row shows the restoration process on the degraded image (from Fig. 5) with Gaussian noise of σ = 10. Similarly, the second row shows restoration with σ = 100. The K value (defined in Eq 7) used is shown below each image.

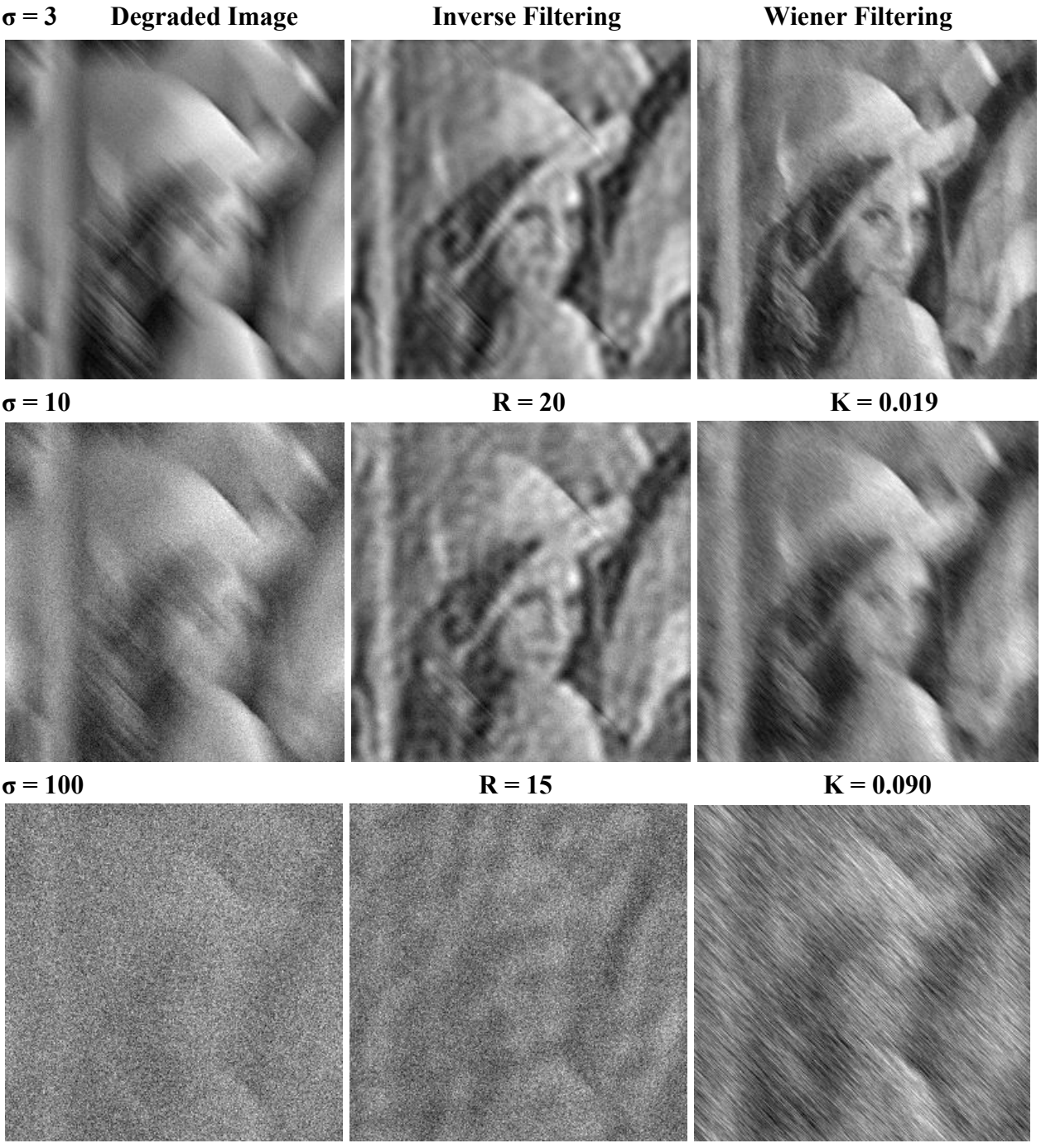

Figure 9: A comparison of Inverse and Wiener filtering on certain degraded "lenna" images (from Fig. 5). The σ value of the Gaussian noise in the degraded image is given above each row. The cutoff radius for inverse filtering (R) and constant for Wiener filtering (K) are given below each image where they apply. Note that this figure only shows the best restored image. Fig. 6, 7, & 8 show the results in more detail.

(4) Homomorphic Filtering

With the help of the ImageComplex class, the task of implementing Homomorphic filtering was relatively simple. After converting the image into an ImageComplex object, the natural log of every real value was taken in the spatial domain. Next, the FFT function within the ImageComplex class was used to transform the image into the frequency domain. Then, another ImageComplex class object was used to create for the filter from (Eq 9). Next, the filter was applied using the complexMultiplication function in the ImageComplex class. Finally, the image was run through the inverse FFT function and the exponential function was applied in the spatial domain. Initially, the γ_L and γ_H values were set at 0.5 and 1.5 respectively. However, for experimental purposes, various values in the range [0, 1] were selected for γ_L and various values in the range [1, 2] were selected for γ_H. Fig. 10 & 11 show the results of Homomorphic filtering.

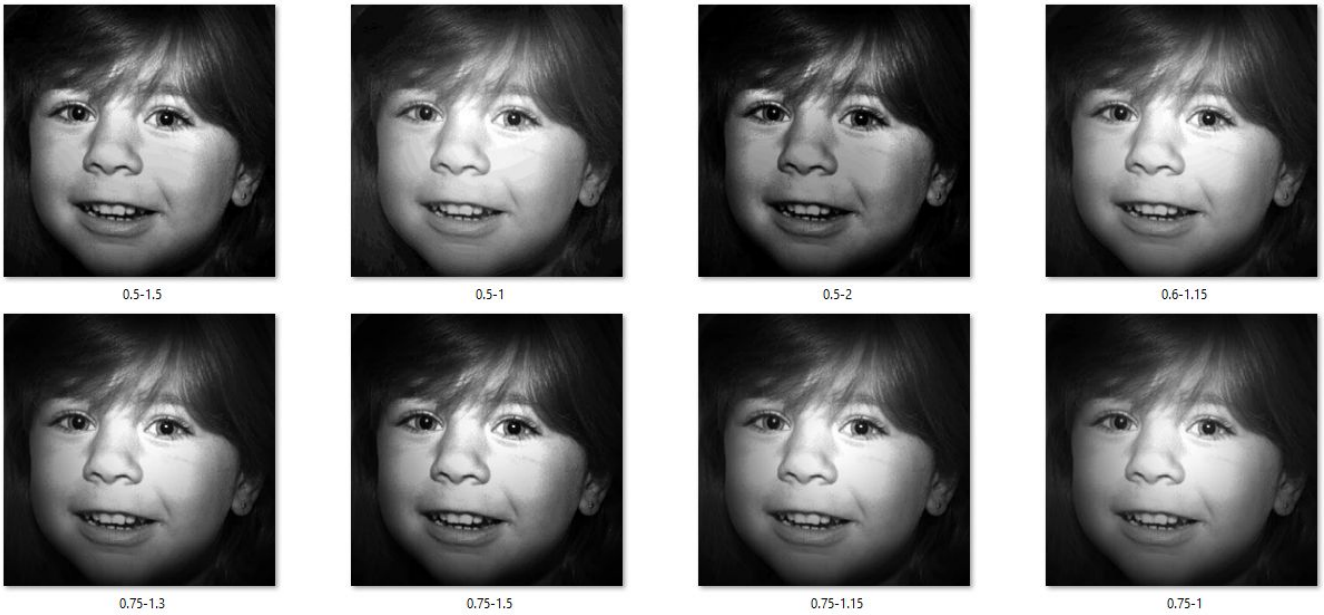

Figure 10: Results of Homomorphic filtering. The γ_L - γ_H value of the high-pass filter (Eq 9) is shown below each image. D_0 was set to 1.8.

Figure 11: Results of Homomorphic filtering. The γ_L - γ_H value of the high-pass filter (Eq 9) is shown below each image. D_0 was set to 1.8.

Discussion

(1) Noise Removal

From the results in Fig. 1, it is evident that frequency domain filtering can be very effective and powerful in removing certain types of noise. We can see that filtering the image in the frequency domain allows us to do things not easily possible in the spatial domain. From the left side of Fig. 1, it is evident that Gaussian smoothing fails to address the root of the noise and simply tries to cover it up by removing necessary details. By comparison, on the right side of the Fig. 1, we can see that frequency domain filtering removes the noise while keeping, and even restoring, important details.

In Fig. 2, we can see exactly why frequency domain filtering was so successful for the given image. From the bottom left image in the figure, we can see that the noise is highly structured in the frequency domain, something that is not as evident in the spatial domain. Namely, the noise seems to fall along a few vertical and horizontal lines in the spectrum. Because of this, a simple band-reject filter that zeros out certain weights, as shown in the bottom right image of Fig. 2, was able to improve the image's quality dramatically. However, it is important to note the cleaned image still has some artifacts. These could have been a result of the ringing effect from the sharp attenuation performed by the band-reject filter. So, it may be possible to get even better results by applying a filter that smoothly attenuates the targeted frequencies. For instance, applying butterworth filter(s) to slowly attenuate the frequencies targeted for removal would have made the final image better.

(3) Image Restoration - Degradation model and noise

The implementation of the degradation model from (Eq 3) was somewhat successful in that it created images that looked to be corrupted by motion blur. This can be seen on the right side of Fig. 4. However, looking at the degraded "book" image in the same figure, it is clear that the degradation model also caused certain artifacts in the filtered image. Namely, in the "book" image, we can see vertical and horizontal lines that should not be there. These artifacts are a result of the degradation function, whose spectrum can be seen in Fig. 3, attenuating the higher frequencies along the u and v axis. This goes to show how the continuous model is imperfect at simulating motion blur in real life images, which are necessarily discrete. However, the model did perform better, at least visually, on the "lenna" image. This is likely due to the fact that the"lenna" image is of smaller resolution (256x256 compared to 512x512) and relies more on lower frequencies.

The addition of Gaussian noise worked as expected. However, as can be seen from the bottom row of Fig. 5, some of the noise values given in the instructions were too high for restoration purposes. For instance, adding noise with mean zero and standard deviation of 1,000 caused the whole image to be lost. This makes sense because the noise was added in the spatial domain, where all the pixel values are in the range [0, 255]. Since adding noise with a standard deviation of 1,000 caused too much corruption for any restoration to makes sense, inverse and Wiener filtering were tested only on the images (from Fig. 5) corrupted by noise of $\sigma = 3, 7, 10, \& 1000$.

(3a) Inverse Filtering

In Fig. 6, it can be seen that inverse filtering was often able to restore the "lenna" image into a form where the viewer can see the major details of the image. For the most part, a cutoff radius in the range [15, 20] produced the best results. The one exception to this was in the image corrupted with Gaussian noise of $\sigma = 100$, where it seemed that a cutoff radius of 10 was best. However, even then, the face in the image is barely discernible. So, it seems that Gaussian noise of $\sigma = 100$ is too much for inverse filtering to handle. Throughout the images, it is evident that inverse filtering does not take the additive noise into account. While the restored images do not show much motion blur, they are plagued by noise. This is something that Wiener filtering should be better at addressing.

(3b) Wiener Filtering

In Fig. 7 and Fig. 8, we can see that Wiener filtering was able to restore the degraded images to a pretty good extent. In terms of choosing the K value from (Eq 7), it seems that increasing it causes more of the motion blur to remain in the image while reducing it causes more noise to remain. So, finding the optimal K value is important for Wiener filtering to work. Moreover, it is evident that the K value is dependent upon the noise being added. This is something that is mathematically provable from (Eq 6).

In Fig. 9, we can see that Wiener filtering tends to perform better than inverse filtering. This is especially true when there is more noise, as show in the bottom row of the Fig. 9. In the images restored with Wiener filtering, we can see more of the details, such as those in the face of the women. In the images restored with inverse filtering, these details are often covered by blobs

of noise. This should be expected because Wiener filtering actually accounts for noise while inverse filtering is adversely affected by it. As a general trend, it seems the Wiener filtering produces results that strike a balance between fixing the noise and motion blur, while inverse filtering only tends to fix motion blur.

(4) Homomorphic Filtering

The results of Homomorphic filtering, location in Fig. 10 & 11, show that it can have some success in fixing shading problems in images. The best result seems to appear when γ_L was 0.6 and γ_H was 1.15. Most of the tips of the hair could be seen on the left side, and some details of the hair on the bottom right side were also visible. At the same time though, a lot of the fine detail for the hair that was originally in the part of the image that had shading problems was lost. The result with γ values 0.75 - 1.3 kept the most detail of all the tested results and showed more details than the original image, but some parts of the image, particularly the corners were still too dark to see. In the image result with γ values 0.75 - 1.15, a lot of extra detail was shown. However, fine detail such as differentiating between strands of hair was lost, and spots of uniform color appeared more frequently, due to the over-aggressive filter. The default γ values of 0.5 - 1.5 had the same problem but was even worse. The top left corner had an extremely large spot of pure black, and the hair in the bottom left showed a large section of the image as just one shade. These results can likely be attributed to data that is simply not there in the first place due to shading errors when the image was created.

Program Listings

The ImageComplex Class

Throughout the three experiments the ImageComplex class is used to make the code easier and simpler. The class's declaration and implementation is given below.

```
//Class to hold complex image pixel values. Stores value as floating points.
//Useful for manipulating data in frequency domain.
class ImageComplex {

public:
  //Constructs image to hold specified number of values.
  ImageComplex(int rows, int cols);

  //Constructs image by copying over values from existing image.
  ImageComplex(const ImageComplex & other);

  //Constructs image from pixel values from ImageType variables.
  //imgR and imgI must have same size.
  ImageComplex(const ImageType & imgR, const ImageType & imgI);

  //Deallocate memory holding pixel data.
  ~ImageComplex(void);

  //Copies over pixel values - rhs must have same size of image.
  ImageComplex& operator=(const ImageComplex & rhs);

  //Adds pixel values of other image - other must have same size.
  void operator+=(const ImageComplex & other);

  //Adds constant to real parts - needed for Wiener filtering.
  ImageComplex operator+(float valReal) const;

  //Get size of image.
  void getImageInfo(int & rows, int & cols) const;

  //Copy pixel data to ImageType variables.
  //All non integer values are rounded down.
  //imgR and imgI must have same size as image.
  void getImageType(ImageType & imgR, ImageType & imgI,
                    bool normalize = true) const;
```

```
  //Get spectrum of image as ImageType variable.
  //Applies log transformation: log(1+val).
  //spectrum must have same size as image.
  void getSpectrum(ImageType & spectrum, bool normalize = true) const;

  //Get real and imaginary value at specific pixel.
  void getPixelVal(int row, int col, float & valR, float & valI) const;

  //Set real and imaginary value at specific pixel.
  void setPixelVal(int row, int col, float valR, float valI);

  //Apply 2D FFT to image. Note the magnitude is shifted internally.
  void applyFFT(bool forward = true);

  //Compute power spectrum of image. Useful for Wiener filtering.
  void powerSpectrum(void);

  //Apply point-by-point complex multiplication.
  //rhs must have same size as image.
  //Negative cutOffRadius means to multiply entire spectrum.
  void complexMultiplication(const ImageComplex & rhs,
                             const float cutoffRadius = -1);

  //Compute complex multiplicative inverse. Needed to do division.
  //Returns calling image for chaining with multiplication.
  ImageComplex& complexInverse(void);

  //Print pixel values to screen. Useful for debugging.
  //Will print to screen if fileName is empty.
  void printPixelValues(const char * fileName = nullptr) const;


private:
      int m_rows;
      int m_cols;

      float ** m_dataR; //real values
      float ** m_dataI; //imaginary value
};
```

```
//Constructs image to hold specified number of values.
ImageComplex::ImageComplex(int rows, int cols)
 : m_rows(rows), m_cols(cols), m_dataR(nullptr), m_dataI(nullptr) {
  //allocate space for real and imaginary parts of data
  m_dataR = new float* [m_rows];
  m_dataI = new float* [m_rows];
  for (int r = 0; r < m_rows; ++r) {
    m_dataR[r] = new float[m_cols]();
    m_dataI[r] = new float[m_cols]();
  }
}

//Constructs image by copying over values from existing image.
ImageComplex::ImageComplex(const ImageComplex & other)
 : ImageComplex(other.m_rows, other.m_cols) {
  //copy over data values
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
      m_dataR[r][c] = other.m_dataR[r][c];
      m_dataI[r][c] = other.m_dataI[r][c];
    }
}

//Constructs image from pixel values from ImageType variables.
//imgR and imgI must have same size.
ImageComplex::ImageComplex(const ImageType & imgR, const ImageType & imgI)
 : m_rows(0), m_cols(0), m_dataR(nullptr), m_dataI(nullptr) {
  //get image info and ensure it is good
  int rowsR, rowsI, colsR, colsI, q;
  imgR.getImageInfo(rowsR, colsR, q);
  imgI.getImageInfo(rowsI, colsI, q);
  assert(rowsR == rowsI && colsR == colsI);
  m_rows = rowsR, m_cols = colsR;

  //allocate space for real and imaginary parts of data
  m_dataR = new float* [m_rows];
  m_dataI = new float* [m_rows];
  for (int r = 0; r < m_rows; ++r) {
    m_dataR[r] = new float[m_cols];
    m_dataI[r] = new float[m_cols];
  }
```

```
  //copy over values
  int tempR, tempI;
  for (int r = 0; r < m_rows; ++r)
  for (int c = 0; c < m_cols; ++c) {
    imgR.getPixelVal(r, c, tempR);
    imgI.getPixelVal(r, c, tempI);
    m_dataR[r][c] = float(tempR);
    m_dataI[r][c] = float(tempI);
  }
}

//Deallocate memory holding pixel data.
ImageComplex::~ImageComplex(void) {
  //free up used space
  for (int r = 0; r < m_rows; ++r) {
    delete[] m_dataR[r];
    delete[] m_dataI[r];
  }
  delete[] m_dataR;
  delete[] m_dataI;
}

//Copies over pixel values - rhs must have same size of image.
ImageComplex& ImageComplex::operator=(const ImageComplex & rhs) {
  //only defined if two images have same size
  assert(m_rows == rhs.m_rows && m_cols == rhs.m_cols);

  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
      m_dataR[r][c] = rhs.m_dataR[r][c];
      m_dataI[r][c] = rhs.m_dataI[r][c];
    }

  return *this;
}

//Adds pixel values of other image - other must have same size.
void ImageComplex::operator+=(const ImageComplex & other) {

  //only defined if two images have same size
  assert(m_rows == other.m_rows && m_cols == other.m_cols);
```

```
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
      m_dataR[r][c] += other.m_dataR[r][c];
      m_dataI[r][c] += other.m_dataI[r][c];
    }
}

//Adds constant to real parts - needed for Wiener filtering.
ImageComplex ImageComplex::operator+(float valReal) const {
  ImageComplex sum(*this);

  //add value to real part of each pixel
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c)
      sum.m_dataR[r][c] += valReal;

  return sum;
}

//Get size of image.
void ImageComplex::getImageInfo(int & rows, int & cols) const {
  rows = m_rows;
  cols = m_cols;
}

//Copy pixel data to ImageType variables.
//All non integer values are rounded down.
//imgR and imgI must have same size as image.

void ImageComplex::getImageType(ImageType & imgR, ImageType & imgI,
                                bool normalize) const {
  //get image info and ensure it is valid
  int rowsR, rowsI, colsR, colsI, q;
  imgR.getImageInfo(rowsR, colsR, q);
  imgI.getImageInfo(rowsI, colsI, q);
  assert(rowsR == rowsI && m_rows == rowsR &&
         colsR == colsI && m_cols == colsR);

  //move over values. Round down to nearest integer
  int tempR, tempI;
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
```

```
      tempR = (int) m_dataR[r][c];
      tempI = (int) m_dataI[r][c];
      imgR.setPixelVal(r, c, tempR);
      imgI.setPixelVal(r, c, tempI);
    }

  //renormalized values if requested
  if (normalize) {
    Helper::remapValues(imgR);
    Helper::remapValues(imgI);
  }
}

//Get spectrum of image as ImageType variable.
//Applies log transformation: log(1+val).
//spectrum must have same size as image.

void ImageComplex::getSpectrum(ImageType & spectrum, bool normalize) const {
  //get image info and ensure it is valid
  int spectrumR, spectrumC, q;
  spectrum.getImageInfo(spectrumR, spectrumC, q);
  assert(spectrumR == m_rows && spectrumC == m_cols);

  //function to get spectrum value - applies log transformation
  auto getNewVal = [](float real, float imaginary) ->int {
    float val = std::sqrt((double) real * real +
                          (double) imaginary * imaginary);
    return ((int) std::log(1 + val));
  };

  //find each value of spectrum
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c)
      spectrum.setPixelVal(r, c, getNewVal(m_dataR[r][c], m_dataI[r][c]));

  //renormalized values if requested
  if (normalize)
    Helper::remapValues(spectrum);
}
```

```
//Get real and imaginary value at specific pixel.
void ImageComplex::getPixelVal(int row, int col,
                               float & valR, float & valI) const {
  valR = m_dataR[row][col];
  valI = m_dataI[row][col];
}


//Set real and imaginary value at specific pixel.
void ImageComplex::setPixelVal(int row, int col, float valR, float valI) {
      if (valR == -0.f) valR = 0.0;
      if (valI == -0.f) valI = 0.0;
      m_dataR[row][col] = valR;
      m_dataI[row][col] = valI
}

//Apply 2D FFT to image. Note the function internally shifts the magnitude.
void ImageComplex::applyFFT(bool forward) {
  //move data into usable form - make sure to extend data to power of 2
  int extendedR = std::pow(2, std::ceil(log(m_rows) / std::log(2)));
  int extendedC = std::pow(2, std::ceil(log(m_cols) / std::log(2)));
  std::vector<float> dataR(extendedR * extendedC, 0.f),
                     dataI(extendedR * extendedC, 0.f);
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
      //copy over data. Invert signs as necessary for shifting
      dataR[r * extendedR + c] = m_dataR[r][c] * (((r + c) % 2) ? 1 : -1);
      dataI[r * extendedR + c] = m_dataI[r][c] * (((r + c) % 2) ? 1 : -1);
    }

  //apply FFT
  fft2D(extendedR, extendedC, &dataR[0], &dataI[0], ((forward) ? -1 : 1));

  //save new values. Make sure to invert shifting transformation
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
      m_dataR[r][c] = dataR[r * extendedR + c] * (((r + c) % 2) ? 1 : -1);
      m_dataI[r][c] = dataI[r * extendedR + c] * (((r + c) % 2) ? 1 : -1);
    }
}
```

```
//Compute power spectrum of image. Useful for Wiener filtering.
void ImageComplex::powerSpectrum(void) {
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
      m_dataR[r][c] = m_dataR[r][c] * m_dataR[r][c] +
                      m_dataI[r][c] * m_dataI[r][c];
      m_dataI[r][c] = 0;
    }
}

//Apply point-by-point complex multiplication. rhs must have same size.
//Negative cutOffRadius means to multiply entire spectrum.
void ImageComplex::complexMultiplication(const ImageComplex & rhs,
                                         const float cutoffRadius) {
  //only defined if two images have same size
  assert(m_rows == rhs.m_rows && m_cols == rhs.m_cols);

  float tempR, tempI;
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
      int u = r - m_rows / 2, v = c - m_cols / 2; //u and v coordinates
      if (cutoffRadius < 0 || std::sqrt(u * u + v * v) <= cutoffRadius)
        tempR = m_dataR[r][c] * rhs.m_dataR[r][c] -
                m_dataI[r][c] * rhs.m_dataI[r][c];
        tempI = m_dataR[r][c] * rhs.m_dataI[r][c] +
                m_dataI[r][c] * rhs.m_dataR[r][c];
        m_dataR[r][c] = tempR, m_dataI[r][c] = tempI;
      }
    }
}

//Compute complex multiplicative inverse. Needed to do division.
//Returns calling image for chaining with multiplication.
ImageComplex& ImageComplex::complexInverse(void) {
  for (int r = 0; r < m_rows; ++r)
    for (int c = 0; c < m_cols; ++c) {
      float x = m_dataR[r][c] * m_dataR[r][c] +
                m_dataI[r][c] * m_dataI[r][c];
      if (x != 0.f) m_dataR[r][c] /= x, m_dataI[r][c] /= -x;
    }
  return *this;
}
```

```
//Print pixel values to screen. Useful for debugging.
//Will print to screen if fileName is empty.
void ImageComplex::printPixelValues(const char * fileName) const {
  if (!fileName) { //print to terminal
    for (int r = 0; r < m_rows; ++r) {
      std::cout << std::endl << " | ";
      for (int c = 0; c < m_cols; ++c)
        std::cout << m_dataR[r][c] << "+j" << m_dataI[r][c] << " | ";
    }
  } else { //print to text file
    std::ofstream file(fileName);

    for (int r = 0; r < m_rows; ++r) {
      file << std::endl << " | ";
      for (int c = 0; c < m_cols; ++c)
        file << m_dataR[r][c] << "+j" << m_dataI[r][c] << " | ";
    }
    file.close();
  }
}
```

(1) Noise Removal

Below is the code that was used to generate the results from experiment 1. The two helper functions at the top are used to implement the band-reject filter. The code for spatial Gaussian filtering is not shown. However, it was discussed and provided in Programming Assignment 2.

```
//Helper function to change the values in a specific row of an image over
the given range of columns - useful for band-reject filter.
void changeRow(ImageComplex & img, int row, int colMin, int colMax,
               int newValR = 0, int newValI = 0) {
  for (int c = colMin; c <= colMax; ++c)
    img.setPixelVal(row, c, newValR, newValI);
}
```

```
//Helper function to change the values in a specific column of an image over
the given range of rows - useful for band-reject filter.
void changeCol(ImageComplex & img, int col, int rowMin, int rowMax,
               int newValR = 0, int newValI = 0) {
  for (int r = rowMin; r <= rowMax; ++r)
    img.setPixelVal(r, col, newValR, newValI);
}

//Main driver code for Experiment # 1.
int main(int argc, char * argv[]) {
  //Read input image
  const std::string imgFile = "images/PA04/Experiment1/boy_noisy.pgm";
  int imgRows, imgCols, Q;
  bool type;
  readImageHeader(imgFile.c_str(), imgRows, imgCols, Q, type);
  ImageType imgSpatial(imgRows, imgCols, Q);
  readImage(imgFile.c_str(), imgSpatial);

  //apply 7x7 Gaussian mask in spatial domain to original and save result
  ImageType img7x7Gau(imgSpatial);
  Helper::gaussianSpatial(img7x7Gau, Helper::GaussianFilterSize::small);
  writeImage("images/PA04/Experiment1/7x7Gaussian.pgm", img7x7Gau);

  //apply 15x15 Gaussian mask in spatial domain to original and save result
  ImageType img15x15Gau(imgSpatial);
  Helper::gaussianSpatial(img15x15Gau, Helper::GaussianFilterSize::large);
  writeImage("images/PA04/Experiment1/15x15Gaussian.pgm", img15x15Gau);

  //make imaginary part of image (all zeros) and create complex image
  ImageType imgI(imgRows, imgCols, Q);
  ImageComplex img(imgSpatial, imgI); //use ImageComplex class

  img.applyFFT(true); //apply forward FFT

  //save spectrum for visualization
  ImageType spectrum(imgRows, imgCols, Q);
  img.getSpectrum(spectrum);
  writeImage("images/PA04/Experiment1/noisy_spectrum.pgm", spectrum);

  //black out areas of rows where noise is prevalent
  int cutOffBottom = 235, cutOffTop = 278;
  std::vector<int> remove = {241, 240, 271, 272};
```

```
  for (int r : remove) {
    changeRow(img, r, 0, cutOffBottom);
    changeRow(img, r, cutOffTop, imgCols - 1);
  }
  cutOffBottom = 100, cutOffTop = 411;
  remove = {255, 256};
  for (int r : remove) {
    changeRow(img, r, 0, cutOffBottom);
    changeRow(img, r, cutOffTop, imgCols - 1);
  }

  //black out areas of columns were noise is prevalent
  cutOffBottom = 235, cutOffTop = 278;
  remove = {254, 255, 256, 257};
  for (int c : remove) {
    changeCol(img, c, 0, cutOffBottom);
    changeCol(img, c, cutOffTop, imgRows - 1);
  }

  //save spectrum for visualization
  ImageType specFiltered(imgRows, imgCols, Q);
  img.getSpectrum(specFiltered);
  writeImage("images/PA04/Experiment1/filtered_spectrum.pgm", specFiltered);

  img.applyFFT(false); //apply inverse FFT

  //save real part of filtered image
  ImageType imgFilteredR(imgRows, imgCols, Q),
            imgFilteredI(imgRows, imgCols, Q);
  img.getImageType(imgFilteredR, imgFilteredI);
  writeImage("images/PA04/Experiment1/filtered.pgm", imgFilteredR);

  return 0; //done
}
```

(3) Image Restoration - Degradation model and noise

On the next page are the three functions used to apply the camera degradation model from (Eq 3) and the addition of Gaussian noise with the help of the given *box_muller()* function.

```
//Function that creates motion blur mask in frequency domain.
//Follows given formula and makes sure mask is centered.
void createMotionBlurFilter(ImageComplex & blurrMaskFrequency,
                            float a = 0.1, float b = 0.1, float T = 1.0) {
  int imgRows, imgCols;
  blurrMaskFrequency.getImageInfo(imgRows, imgCols);

  float valR, valI, si, co;
  for (int r = 0; r < imgRows; ++r)
    for (int c = 0; c < imgCols; ++c) {
      int u = r - imgRows / 2, v = c - imgCols / 2; //u and v coordinates
      float x = u * a + v * b; //repeating part of formula
      //normalize angle to 0-2 (pi)
      float an = std::fmod(x, 2);
      if (an < 0.f) an = 2 - std::abs(an);

      //manually set certain sine and cosine values to avoid rounding errors
      if (an == 0.f) si = 0, co = 1;
      else if (an == 0.5) si = 1, co = 0;
      else if (an == 1.f) si = 0, co = -1;
      else if (an == 1.5) si = -1, co = 0;
      else si = std::sin(M_PI * an), co = std::cos(M_PI * an);

      if (x != 0.f) { //ensure we don't divide by zero
        //real and imaginary value according to formula
        valR = (T / (M_PI * x)) * si * co;
        valI = -(T / (M_PI * x)) * si * si;
      } else { //value due to limits of formula
        valR = T, valI = 0;
      }

      blurrMaskFrequency.setPixelVal(r, c, valR, valI); //set new value
    }
}

//Function to take ComplexImage (in spatial domain) and compute the blurred
image (in frequency domain).
//Uses degradation model discussed in class and in the book.
void getMotionBlurred(const ImageComplex & imgSpatial,
                      ImageComplex & imgBlurredFrequency) {
  int imgRows, imgCols;
  imgSpatial.getImageInfo(imgRows, imgCols);
```

```
  ImageComplex img(imgSpatial);

  img.applyFFT(true); //apply forward FFT

  //save spectrum of original image for visualization
  ImageType imgSpectrum(imgRows, imgCols, 255);
  img.getSpectrum(imgSpectrum);
  writeImage("images/PA04/Experiment3/original_spectrum.pgm", imgSpectrum);

  //create motion blur mask in frequency domain
  ImageComplex blurMaskFrequency(imgRows, imgCols);
  createMotionBlurFilter(blurMaskFrequency);

  //apply motion blur to image in frequency domain
  img.complexMultiplication(blurMaskFrequency); //do complex multiplication
  imgBlurredFrequency = img;
}
```

(3a) Inverse Filtering

Below is the function used to apply inverse filtering.

```
//Function to take degraded ComplexImage (in frequency domain) and restore
it (to spatial domain) via inverse filtering.
//Uses degradation model discussed in class and in the book.
void applyInverse(const ImageComplex & imgDegradedFrequency,
                  ImageComplex & imgRestoredSpatial,
                  float cutoffR) {
  int imgRows, imgCols;
  imgDegradedFrequency.getImageInfo(imgRows, imgCols);
  ImageComplex img(imgDegradedFrequency);

  //create motion blur mask in frequency domain
  ImageComplex blurMaskFrequency(imgRows, imgCols);
  createMotionBlurFilter(blurMaskFrequency);

  //divide by motion degradation mask with cutoff radius specified
  img.complexMultiplication(blurMaskFrequency.complexInverse(), cutoffR);

  //save spectrum of restored image for visualization
  ImageType imgRestoredSpectrum(imgRows, imgCols, 255);
```

```
  img.getSpectrum(imgRestoredSpectrum);
  writeImage("images/PA04/Experiment3/restored_spectrum_inverse.pgm",
             imgRestoredSpectrum);

  img.applyFFT(false); //apply inverse FFT to get restored image
  imgRestoredSpatial = img;
}
```

(3b) Wiener Filtering

Below is the code used to apply Wiener filtering.

```
//Function to take degraded ComplexImage (in frequency domain) and restore
it (to spatial domain) via Wiener filtering.
//Uses model discussed in class and in the book with provided K.
void applyWiener(const ImageComplex & imgDegradedFrequency,
                 ImageComplex & imgRestoredSpatial, float K) {
  int imgRows, imgCols;
  imgDegradedFrequency.getImageInfo(imgRows, imgCols);
  ImageComplex img(imgDegradedFrequency);

  //create motion blur mask in frequency domain
  ImageComplex blurMaskFrequency(imgRows, imgCols);
  createMotionBlurFilter(blurMaskFrequency);

  //compute necessary power images
  ImageComplex blurPower(blurMaskFrequency);
  blurPower.powerSpectrum();
  ImageComplex blurPowerPlusK = blurPower + K;

  //use wiener formula as given in notes/book
  blurPower.complexMultiplication(blurPowerPlusK.complexInverse());
  img.complexMultiplication(blurPower);
  img.complexMultiplication(blurMaskFrequency.complexInverse());

  //save spectrum of restored image for visualization
  ImageType imgRestoredSpectrum(imgRows, imgCols, 255);
  img.getSpectrum(imgRestoredSpectrum);
  writeImage("images/PA04/Experiment3/restored_spectrum_wiener.pgm",
             imgRestoredSpectrum);
```

```
  img.applyFFT(false); //apply inverse FFT to get to spatial domain
  imgRestoredSpatial = img;
}
```

(3) Main Driver Coder

Below is the main driver code for experiment 3. It that takes an image, corrupts is by applying the motion blur (Eq 3) and Gaussian noise, and then restores it via inverse and Wiener filtering.

```
//main driver code for Experiment # 3
int main(int argc, char * argv[]) {
  //main parameters for run
  const std::string imgFile = "images/PA04/Experiment3/lenna.pgm";
  const float noiseMean = 0.f, noiseStandardDeviation = 10.f;
  const int inverseCutoffRadius = 20;
  const float wienerK = 0.007;

  std::srand(time(nullptr));
  std::cout << std::endl << "Experiment 3" << std::endl;

  // Read input image
  int imgRows, imgCols, Q;
  bool type;
  readImageHeader(imgFile.c_str(), imgRows, imgCols, Q, type);
  ImageType imgSpatialR(imgRows, imgCols, Q);
  readImage(imgFile.c_str(), imgSpatialR);

  //make imaginary part (all zeros) and use ComplexImage for simplicity
  ImageType imgSpatialI(imgRows, imgCols, Q);
  ImageComplex imgSpatial(imgSpatialR, imgSpatialI);

  //get blurred image in frequency domain
  ImageComplex imgBlurredFrequency(imgRows, imgCols);
  getMotionBlurred(imgSpatial, imgBlurredFrequency);

  //save spectrum of blurred image for visualization
  ImageType imgBlurredSpectrum(imgRows, imgCols, Q);
  imgBlurredFrequency.getSpectrum(imgBlurredSpectrum);
```

```
writeImage("images/PA04/Experiment3/blurred_spectrum.pgm",
           imgBlurredSpectrum);

//compute & save real part of blurred image for visualization/verification
ImageComplex imgBlurredSpatial(imgBlurredFrequency);
imgBlurredSpatial.applyFFT(false);
ImageType imgBlurredSpatialR(imgRows, imgCols, Q),
imgBlurredSpatialI(imgRows, imgCols, Q);
imgBlurredSpatial.getImageType(imgBlurredSpatialR, imgBlurredSpatialI);
writeImage("Experiment3/blurred.pgm", imgBlurredSpatialR);

//add noise to blurred image in frequency domain
ImageComplex imgNoiseFrequency(imgBlurredFrequency);
addGaussianNoise(imgNoiseFrequency, noiseMean, noiseStandardDeviation);

//save spectrum of noisy image for visualization
ImageType imgNoiseSpectrum(imgRows, imgCols, 255);
imgNoiseFrequency.getSpectrum(imgNoiseSpectrum);
writeImage("Experiment3/blurred_noisy_spectrum.pgm", imgNoiseSpectrum);

//compute and save real part of noisy image for visualization/verification
ImageComplex imgNoiseSpatial(imgNoiseFrequency);
imgNoiseSpatial.applyFFT(false);
ImageType imgNoiseSpatialR(imgRows, imgCols, Q),
          imgNoiseSpatialI(imgRows, imgCols, Q);
imgNoiseSpatial.getImageType(imgNoiseSpatialR, imgNoiseSpatialI);
writeImage("Experiment3/blurred_noisy.pgm", imgNoiseSpatialR);

//get spatial domain restored image from inverse filtering
ImageComplex imgRestoredInverse(imgRows, imgCols);
applyInverse(imgNoiseFrequency, imgRestoredInverse, inverseCutoffRadius);

//compute & save real part of inverse image for visualization/verification
ImageType imgRestoredInverseR(imgRows, imgCols, Q),
imgRestoredInverseI(imgRows, imgCols, Q);
imgRestoredInverse.getImageType(imgRestoredInverseR, imgRestoredInverseI);
writeImage("Experiment3/restored_inverse.pgm", imgRestoredInverseR);

//get spatial domain restored image from wiener filtering
ImageComplex imgRestoredWiener(imgRows, imgCols);
applyWiener(imgNoiseFrequency, imgRestoredWiener, wienerK);
```

```
  //compute & save real part of Wiener image for visualization/verification
  ImageType imgRestoredWienerR(imgRows, imgCols, Q),
  imgRestoredWienerI(imgRows, imgCols, Q);
  imgRestoredWiener.getImageType(imgRestoredWienerR, imgRestoredWienerI);
  writeImage("Experiment3/restored_wiener.pgm", imgRestoredWienerR);

  return 0; //done
}
```

(4) Homomorphic Filtering

Below is the code for homomorphic filtering. The comments show what is happening at a high level, while additional details can be read in section (4) of *Implementation and Results*.

```
//main driver code for Experiment # 4
int main(int argc, char * argv[]) {

  const std::string imgFile = "images/PA04/Experiment4/girl.pgm",
                    outFile = "images/PA04/Experiment4/filterd_girl.pgm";
  //read image
  int N, M, Q;
  bool type;
  readImageHeader(imgFile.c_str(), N, M, Q, type);
  ImageType og_image(N, M, Q);
  readImage(imgFile.c_str(), og_image);

  //create imaginary part of image - zero by default
  ImageType og_imaginary(N, M, Q);

  //use ImageComplex for simplicity since
  //logarithmic functions will work on it
  ImageComplex ln_image(og_image, og_imaginary);

  //natural log the original image
  float tempR, tempI;
  for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; j++) {
      ln_image.getPixelVal(i, j, tempR, tempI);
      ln_image.setPixelVal(i, j, std::log(tempR), tempI);
    }
```

```
  ln_image.applyFFT(true); //Forward Fourier Transform

  //display spectrum
  ImageType imgSpec(N, M, Q);
  ln_image.getSpectrum(imgSpec);
  writeImage("images/PA04/Experiment4/ln_spectrum.pgm", imgSpec);

  //Create high pass filter
  const float y_low = 0.5, y_high = 1.5, d_cutoff = 1.8, c = 1;
  ImageComplex filter(N, M);
  for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; j++) {
      int u = j - N / 2, v = i - M / 2; //u and v coordinates of pixel
      tempR = 1 - std::exp(-c * ((u * u + v * v) / (d_cutoff * d_cutoff)));
      tempR *= (y_high - y_low);
      tempR += y_low;
      filter.setPixelVal(i, j, tempR, 0);
    }

  ln_image.complexMultiplication(filter); //multiply by filter

  //display spectrum
  ln_image.getSpectrum(imgSpec);
  writeImage("images/PA04/Experiment4/filtered_spectrum.pgm", imgSpec);

  ln_image.applyFFT(false);//Inverse Fourier Transform

  //apply exponential function in spatial domain
  ImageType modified_real(N, M, Q), modified_imaginary(N, M, Q);
  ImageComplex exp_image(N, M);
  for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; j++) {
      ln_image.getPixelVal(i, j, tempR, tempI);
      exp_image.setPixelVal(i, j, std::exp(tempR), tempI);
    }

  //change type back to ImageType and save result
  exp_image.getImageType(modified_real, modified_imaginary);
  writeImage(outFile .c_str(), modified_real);

  return 0; //done
}
```