97+10

Image Enhancement in Space and in Frequency (October 3, 2011)

Ali Ghezawi

ECE 572 – Digital Image Processing

Abstract— **The work dealt with spatial and frequency-based filtering targeting high frequencies, low frequencies, and both frequencies. High-pass and low-pass Butterworth and Gaussian frequency filters were used as well as Laplacian and Sobel edge detectors. Unsharp masking demonstrated its sharpening ability, and the nature of the DFT on simple functions was explored. Homomorphic filtering covered the filtering of both high and low frequencies. The results primarily showed the differences between spatial and frequency methods with frequency-based methods seeming to be the champion for linear methods.**

I. The Fourier Transform

A. A Square

Great!

LET the discrete unit step be

$$\mathrm{u}[n] \triangleq \begin{cases} 1, n \geq 0 \\ 0, n < 0 \end{cases} \tag{1}$$

A discrete square of side length s shifted m units from the x and y origin then can be written as

$$f[x,y] = \begin{pmatrix} \mathrm{u}[x-m]- \\ \mathrm{u}[x-s-m] \end{pmatrix} \begin{pmatrix} \mathrm{u}[y-m]- \\ \mathrm{u}[x-s-m] \end{pmatrix} \tag{2}$$

where this function is enclosed inside an r by c image surrounded by zeros. The discrete Fourier transform (DFT) of (2) repeated every r and c pixels for x and y respectively can then be found as

$$f[x,y] \underset{r,c}{\overset{\mathcal{DFT}}{\Longleftrightarrow}} F[u,v] = \underbrace{\sum_{x=m}^{s+m-1} e^{-\frac{j2\pi ux}{r}}}_{F_x[u]} \underbrace{\sum_{y=m}^{s+m-1} e^{-\frac{j2\pi vy}{c}}}_{F_y[v]} \tag{3}$$

Thus, from the property of separability, F is the multiplication of F_x and F_y, which have the same form, so the following derivation will only be for F_x. Afterward, we will find F_y by inspection and produce the product to find the overall DFT.

$$\begin{aligned} F_x[u] &= \frac{e^{-\frac{j2\pi um}{r}} - e^{-\frac{j2\pi u(s+m)}{r}}}{1-e^{-\frac{j2\pi u}{r}}} \\ F_x[u] &= e^{-\frac{j2\pi um}{r}} \frac{1-e^{-\frac{j2\pi us}{r}}}{1-e^{-\frac{j2\pi u}{r}}} \\ F_x[u] &= e^{-\frac{j2\pi u}{r}\left(m+\frac{s}{2}-\frac{1}{2}\right)} \frac{e^{\frac{j\pi us}{r}} - e^{-\frac{j\pi us}{r}}}{e^{\frac{j\pi u}{r}} - e^{-\frac{j\pi u}{r}}} \\ F_x[u] &= e^{-\frac{j2\pi u}{r}\left(m+\frac{s}{2}-\frac{1}{2}\right)} \frac{\sin\left(\frac{\pi us}{r}\right)}{\sin\left(\frac{\pi u}{r}\right)} \end{aligned} \tag{4}$$

The transform then is

$$F[u,v] = e^{-\frac{j2\pi u}{r}\left(m+\frac{s}{2}-\frac{1}{2}\right)} \frac{\sin\left(\frac{\pi us}{r}\right)}{\sin\left(\frac{\pi u}{r}\right)} e^{-\frac{j2\pi v}{c}\left(m+\frac{s}{2}-\frac{1}{2}\right)} \frac{\sin\left(\frac{\pi vs}{c}\right)}{\sin\left(\frac{\pi v}{c}\right)} \tag{5}$$

In the special case where $s = 1$, which makes a unit impulse, the DFT simplifies to

$$\begin{aligned} F[u,v] &= e^{-\frac{j2\pi um}{r}} \frac{\sin\left(\frac{\pi u}{r}\right)}{\sin\left(\frac{\pi u}{r}\right)} e^{-\frac{j2\pi vm}{c}} \frac{\sin\left(\frac{\pi v}{c}\right)}{\sin\left(\frac{\pi v}{c}\right)} \\ F[u,v] &= e^{-\frac{j2\pi um}{r}} e^{-\frac{j2\pi vm}{c}} \end{aligned} \tag{6}$$

which is, as expected, a constant in magnitude for all u and v with a phase shift introduced by the circular shift m.

In the experiment, we considered only the magnitude, and since the magnitude of a complex exponential is one, (5) simplified to

$$\left|F[u,v]\right| = W\left|\frac{\sin\left(\frac{\pi us}{r}\right)\sin\left(\frac{\pi vs}{c}\right)}{\sin\left(\frac{\pi u}{r}\right)\sin\left(\frac{\pi v}{c}\right)}\right| \tag{7}$$

and (6) simplified to

$$\left|F[u,v]\right| = W \tag{8}$$

where W was 255 for white.

Fig. 1 shows the experimental results of finding the DFT of various types of squares. As expected by (8), the DFT for $s = 1$ resulted in a white image. This whiteness was not only visually true: the pixel values were all 255 after the transformation. The form of (7) is that of multiplied Dirichlet functions, which are infinite sums of equally spaced sinc functions. As such, along the axes, where the other Dirichlet multiplicatively contributed one, the DFT looked much like a sinc function (and looks fully like a Dirichlet function). The DFTs had their largest hump at the origin, equal to W, with subsequent humps within a period never being as large as W. And if either u or v evaluated its Dirichlet to zero, the DFT at that entire row or column equaled zero. Last, as s/r or s/c grew, more local maximums showed in the DFT with the extreme case of $s = 1$ resulting in only one 'local maximum'. Also, when $s = 2$, it had only one local maximum. This result is expected since (7) shows a larger s makes the numerator more frequent than the denominator.

B. *A Line*

F for a line image, f, is the multiplication of F_x and F_y, which both have the same form as a square's F_x and F_y. Hence, by inspection, the DFT of a unit line is

$$F[u,v] = e^{-\frac{j2\pi u}{r}\left(m+\frac{s}{2}-\frac{1}{2}\right)}\frac{\sin\left(\frac{\pi ua}{r}\right)}{\sin\left(\frac{\pi u}{r}\right)}e^{-\frac{j2\pi v}{c}\left(m+\frac{s}{2}-\frac{1}{2}\right)}\frac{\sin\left(\frac{\pi vb}{c}\right)}{\sin\left(\frac{\pi v}{c}\right)} \tag{9}$$

And its scaled magnitude is

$$\left|F[u,v]\right| = W\left|\frac{\sin\left(\frac{\pi ua}{r}\right)\sin\left(\frac{\pi vb}{c}\right)}{\sin\left(\frac{\pi u}{r}\right)\sin\left(\frac{\pi v}{c}\right)}\right| \tag{10}$$

where a and b are the lines's height and width.

The experimental results for the line are in fig. 2. If exclusively either a or b equaled one, the DFT simplified to a single Dirichlet, varying only with v or u respectively. Hence, in that special case, it produced lines of varying intensity

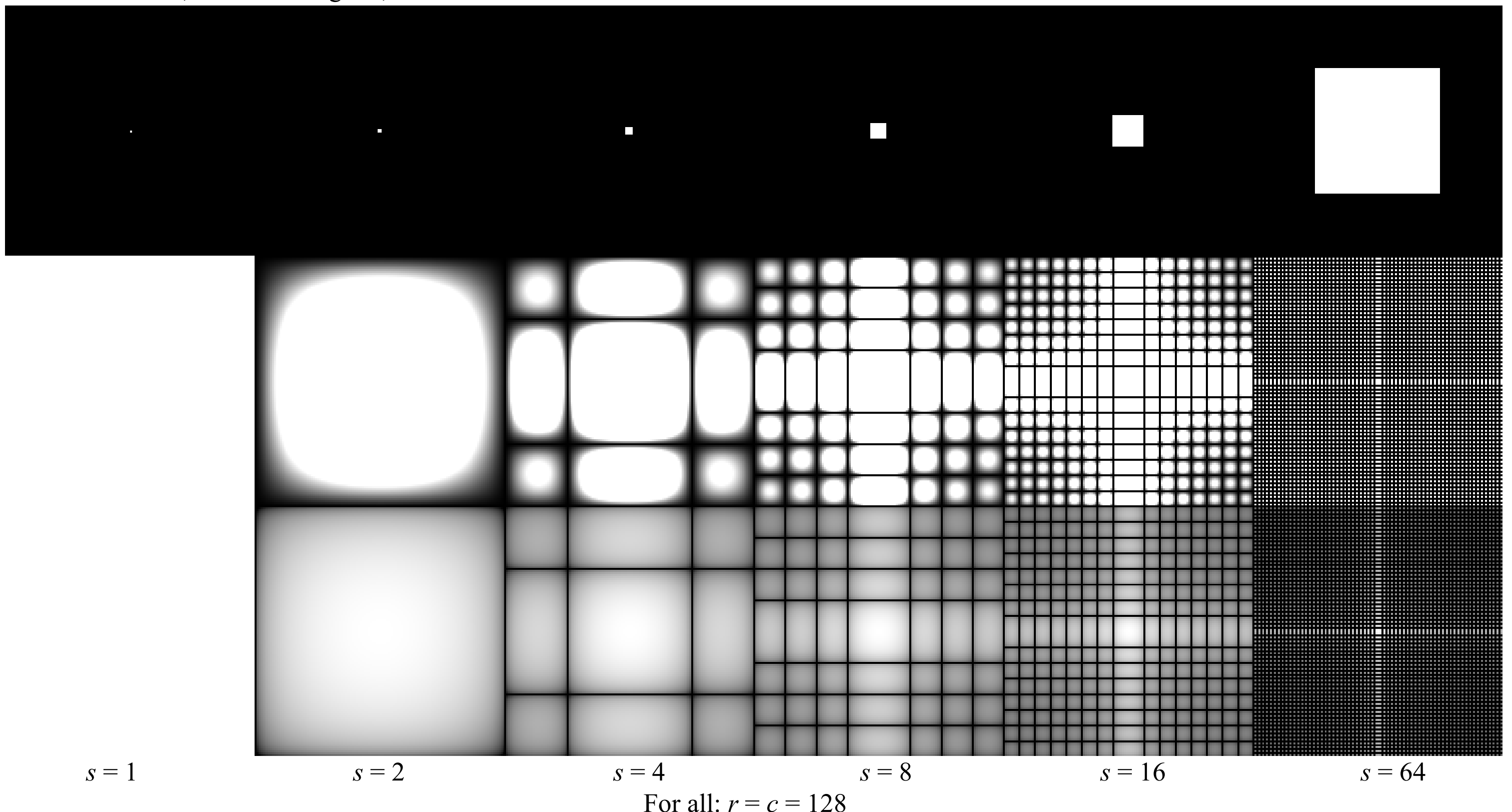

Fig. 1. Each column corresponds to a differently sized square. Row one shows the image of the square, row two shows the magnitude of DFT of the image, and row three shows the log-scaled version of the row above it. The scaling factor in the log transformation was determined such that the maximum intensity would be 255.

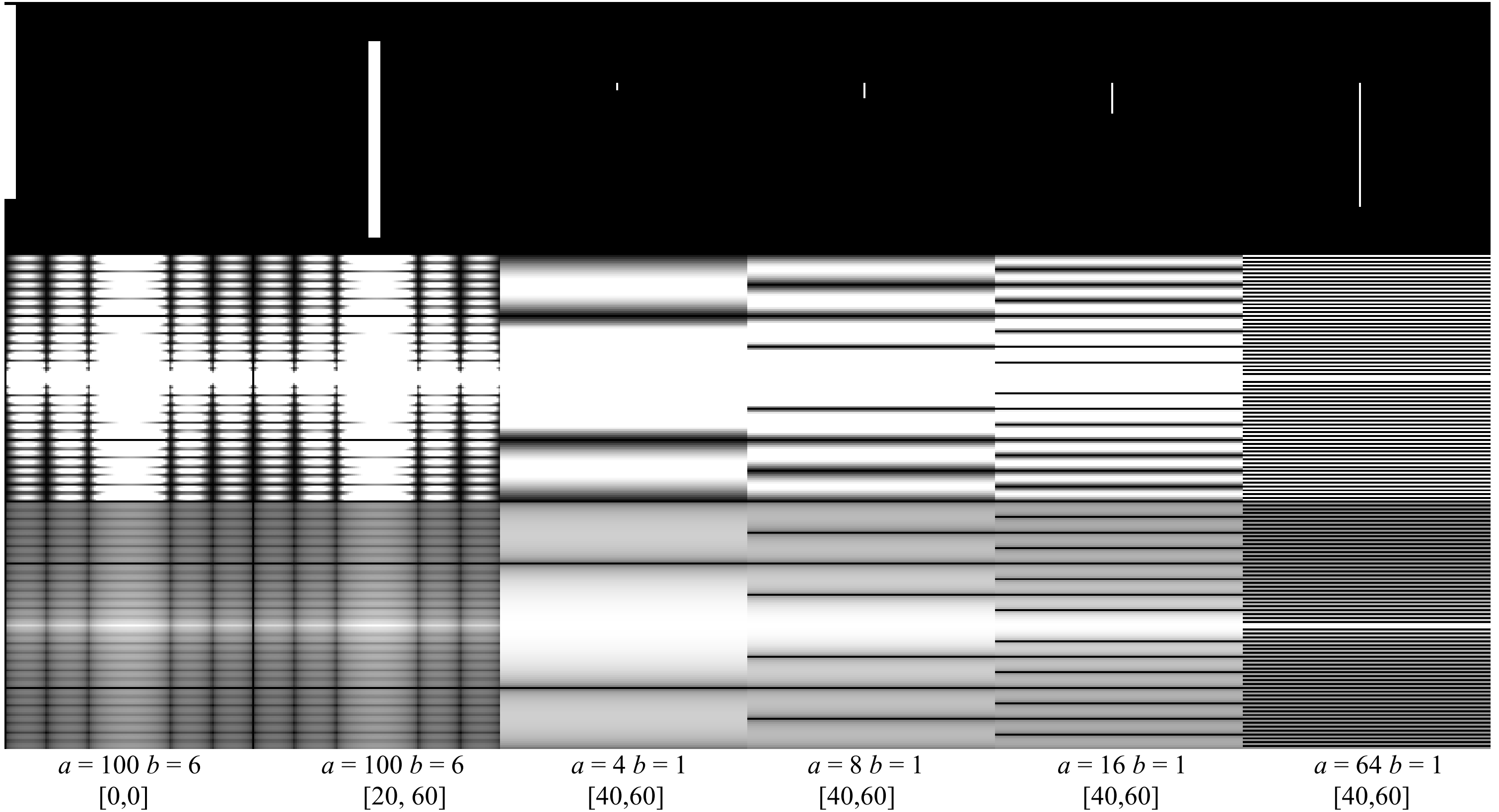

Fig. 2. Each column corresponds to a different line. Row one shows the image of the line, row two shows the magnitude of DFT of the image, and row three shows the log-scaled version of the row above it. The scaling factor in the log transformation was determined such that the maximum intensity would be 255.

similar to a sinc function orthogonal to the spatial domain's line. Note, this variation is similar to the square's DFT's variations too, just with an axis's effect held to a constant. As with the square, the number of humps grew with a/r and b/c since the numerator became more frequent than the denominator. And if either u or v evaluated its Dirichlet to zero, the DFT at that entire row or column equaled zero. Last, the circular shift m is not in (10). Thus, locating a line differently did not change its magnitude plot. A line, unlike a square, offered the ability for different characteristics in the DFT horizontally and vertically. Other than that freedom, the results were quite similar to a square's DFT.

II. Low-pass Filtering

A. Spatial Filtering

Gaussian filtering used the non-normalized masks below, though normalization took place in the convolution function.

$$M_{5x5} = \begin{pmatrix} .012 & .023 & .029 & .023 & .012 \\ .023 & .045 & .057 & .045 & .023 \\ .029 & .057 & .071 & .057 & .029 \\ .023 & .045 & .057 & .045 & .023 \\ .012 & .023 & .029 & .023 & .012 \end{pmatrix} \quad (11)$$

$$M_{7x7} = \begin{pmatrix} .0013 & .0039 & .0077 & .0096 & .0077 & .0039 & .0013 \\ .0039 & .0120 & .0233 & .0291 & .0233 & .0120 & .0039 \\ .0077 & .0233 & .0454 & .0566 & .0454 & .0233 & .0077 \\ .0096 & .0291 & .0566 & .0707 & .0566 & .0291 & .0096 \\ .0077 & .0233 & .0454 & .0566 & .0454 & .0233 & .0077 \\ .0039 & .0120 & .0233 & .0291 & .0233 & .0120 & .0039 \\ .0013 & .0039 & .0077 & .0096 & .0077 & .0039 & .0013 \end{pmatrix} \quad (12)$$

Fig. 3 shows the results from filtering with Gaussian masks and from filtering with a non-linear median filter. The median filter was better at removing scratches since the scratch's extreme whiteness almost never made it to the middle of the ordered pixel values for any given neighborhood. The Gaussian filter, on the other hand, did a weighted average wherein the extreme whiteness still left the resulting average skewed for each neighborhood. As a result, almost no scratches were removed and the images were left blurred. The Gaussian also had a ceiling to its effect relative to mask size since the nature of the Gaussian is to reduce quickly to near-zero values. This ceiling was seen by the 7x7 filtering adding little difference to the 5x5 filtering. Median filtering, differently, rapidly grew in effect with the mask size.

The median was able to produce such powerful results due to its nonlinearity. Due to it, it could nonlinearly focus in on certain types of noise reduction: salt and pepper. Linear systems cannot nonlinearly pinpoint extremes like ordered-

Fig. 3. The top row is Gaussian Blurring, and the bottom row is median filtering. The dimensions below correspond to the mask size used for each spatial filtering process.

statistical methods can, and it is a common error in linear (and even squared) systems to overemphasize extremities.

B. Frequency Filtering

We applied Gaussian and Butterworth filters to match the effects of the mask sizes from fig. 3 with the results shown in fig. 4. A smaller D_0 corresponded with a larger mask, because larger masks filtered out more frequency via convolution. To find the best parameters to match the spatial convolution, we minimized the squared value of the difference of the spatially filtered image with the frequency-filtered image, using integer values for the parameters (to minimize the manual search). From left to right, the power ratios for the Gaussian were 98.32, 97.22 and 96.74, and they were 98.98, 98.1, and 96.37 for the Butterworth filter. Interestingly, the brute-forced parameters resulted in similar power ratios for each method. Given the visual similarities, given the similarities in power ratios after searching for the best parameters, and given the small squared error in the difference after optimization, frequency domain can match spatial domain techniques, which is predicted and expected by the convolution-multiplication duality of the DFT.

Fig. 4. The top row is Gaussian frequency filtering with, from left to right, D_0 = 96, 65, and 57. The bottom row is Butterworth filtering with, from left to right, D_0 = 104, 70, and 69. The orders, from left to right, are 2, 2, and 1.

III. High-pass Filtering

A. Spatial Filtering

We realized edge detection of a checkered pattern with noise in space via convolution with the Laplacian mask and a Sobel mask. Further, we also used arithmetic averaging in an unsharp masking procedure to sharpen the noisy image. Fig. 5 shows the results of these filters unto the checkered pattern.

The Sobel filter, through a 2nd root, was nonlinear whereas the Laplacian used only convolution, thus making it linear. Further, the Laplacian was isotropic whereas the Sobel combined, nonlinearly, two anisotropic processes. The Laplacian result had thinner, more precise lines at the expense of the boldness and certainty provided by the Sobel filter while the Sobel paid full in precision for its boldness.

Bigger masks in the unsharp masking procedure resulted in sharper lines. In this example, however, it was difficult to see the differences between the original and any of the three output images.

B. Frequency Filtering

Fig. 5 also shows the results of the high-pass filtering in frequency. These results showed that the image has mostly low-frequency content. When the cutoff frequency let through enough low-frequency content, a gray colored the background for both filters.

The Gaussian filter achieved power ratios, from left to right in the diagram, of 34.29, 4.29, and 1.25 whereas the first-order Butterworth resulted in, from left to right, 34.45, 4.98, and 1.65.

IV. Convolution and Correlation

A. Efficiency in Space and in Frequency

Fig. 6 shows 5x5 and 25x25 arithmetic average filters being used in space and in frequency. It also shows the differences,

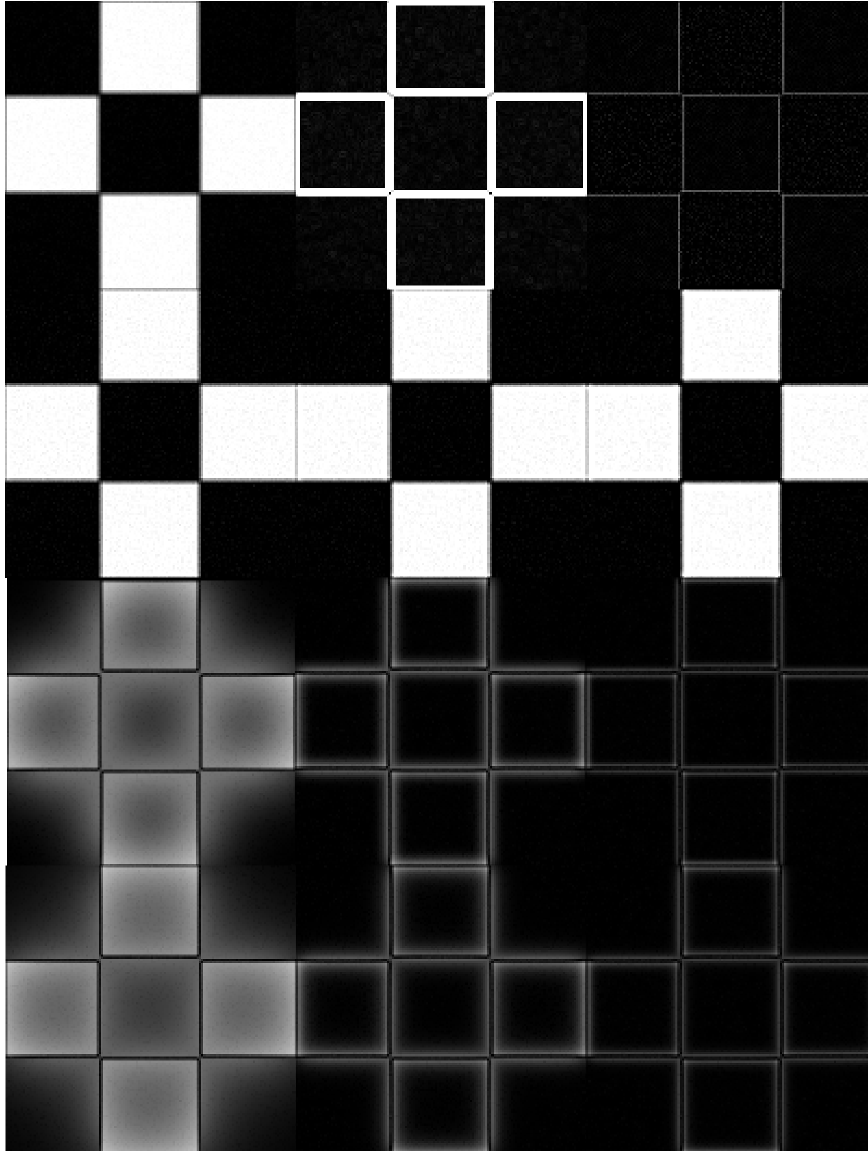

Fig. 5. From left to right on the top row: Original image, Sobel mask, Laplacian mask. Row 2: Unsharp masking with 3x3, 5x5, and 7x7 arithmetic averaging filter. Row 3: High-pass Gaussian frequency filtering with, from left to right, D_0 = 5, 25, and 50. The bottom row is first-order, high-pass Butterworth filtering with, from left to right, D_0 = 5, 25, and 50.

both unscaled and log-scaled, of the results for spatial and frequency filtering. The automatic log function, which brought the highest value to 255, outputted a white image for both cases. Hence, the errors were so similar that after dynamic compression, they were all nearly exactly the same still, which attested to the convolution-multiplication duality of the DFT. The sum of the entire absolute value difference image for the 5x5 case was only 1.4 whereas for the 25x25 case it was only 6.69! To the naked eye, the difference images were also purely black. Table I shows the time spent computing each of these effects. DFT methods, though slower for smaller images, quickly become more effective since they hardly worsened at all when the image size increased whereas spatial time worsened almost exponentially with the number of pixels.

B. *Autocorrelation*

Fig. 6 also shows the autocorrelation of the image in space and in frequency. The autocorrelation was just the correlation of the image with a normalized version of itself (to avoid all white). The frequency output had very faint white regions at its corners. Along the same tone as in *IV.B*, when the correlation was spatial with the 256x256 mask, the time exploded whereas the frequency method almost matched exactly with the 25x25 mask.

TABLE I
TIMINGS FOR METHODS

Method	5x5	25x25	Autocorrelation (256x256)
Space	.02 s	.61 s	37.64 s
Frequency	.78 s	.77 s	.78 s

V. HOMOMORPHIC FILTERING

Fig. 6 shows the results from the homomorphic filtering. The image gained contrast enhancement with dynamic range compression at the same time. The filter was a high-pass Gaussian with $\gamma_{low} = .5$, $\gamma_{high} = 4$, and $D_0 = 50$. A high γ_{high} ensured the fuzzy edges came in clearly whereas a decently low γ_{low} reduced the brightness in the original image.

VI. CONCLUSION

The efficiency of frequency methods for linear operations was seen by its ability to replicate spatial methods with better efficiency. Further, the frequency abstraction was shown to help in understanding the filtering process. Spatially nonlinear methods showed its use over linear DFT processes, too. Future work could be to implement even more robust and efficient algorithms to perform the operations.

-2

Fig. 6. The top row: original image, average filtered spatially 5x5, frequency average filtered 5x5, spatially average filtered 25x25. Second row: difference between spatial and frequency 25x25 log-scaled, difference between spatial and frequency 5x5 log-scaled, difference between spatial and frequency 5x5, frequency average filtered 25x25. Bottom row: difference between spatial and frequency for 25x25, spatial autocorrelation, frequency autocorrelation, and homomorphic filtering .

APPENDIX

A. Project3.cpp (main)

```
#include "Dip.h"
#include <iostream>
#include <cstring>
#include <ctime>
#include <cmath>
#include <cstdlib>
#include <cstdio>
using namespace std;

#define USAGE "Project3 inImg outImg taskNumber[optional]\n"

Image flip(const Image& IN_IMG)
{
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    for(int row = 0; row < IN_IMG.getRow(); ++row)
        for(int col = 0; col < IN_IMG.getCol(); ++col)
            outImg(row, col) = IN_IMG(IN_IMG.getRow() - 1 - row, IN_IMG.getCol() - 1 - col);

    return outImg;
}

int main(int argc, char **argv)
{
    /*
     * Manage input
     */
    char taskNumber[3];
    if (argc < 3)
    {
        cout << USAGE;
        exit(3);
    }
    else if(argc != 4)
    {
        cout << "1: Square" << endl;
        cout << "2: Line" << endl;
        cout << "3: Lowpass filtering" << endl;
        cout << "4: Highpass filtering" << endl;
        cout << "5: Convolution and correlation" << endl;
        cout << "6: Homomorphic filter" << endl;

        cout << "Selection : ";

        cin >> taskNumber;
    }
    else
    {
        taskNumber[0] = argv[3][0];
        taskNumber[1] = argv[3][1];
    }

    if(taskNumber[1] != '\0' || taskNumber[0] > '6' || taskNumber[0] < '1')
    {
        cout << "Invalid selection." << endl;
        exit(3);
    }

    const Image IN_IMG = readImage(argv[1]);

    // for naming certain outputs
    const int NAME_SIZE = strlen(argv[2]);
    char* outputName = new char[NAME_SIZE + 40];
    strcpy(outputName, argv[2]);
    outputName[NAME_SIZE - 4] = '\0';

    /*
     * Perform selected task
     */
    switch(taskNumber[0])
    {
        case '1' : // Square
        {
            int imgSize, sqrSize;

            cout << "Enter square image's side length: ";
            cin >> imgSize;
            cout << "Enter size of square: ";
            cin >> sqrSize;

            Image whiteSquare = square(imgSize, imgSize, sqrSize);
            Image mag2(imgSize, imgSize);
            Image phase2(imgSize, imgSize);

            fft(whiteSquare, mag2, phase2);

            writeImage(whiteSquare, "blackWhiteSquare.pgm");
            writeImage(mag2, "blackWhiteSquareMag.pgm");
            automaticLog(mag2);
            writeImage(mag2, "BlackWhiteSquareLogMag.pgm");

        }
        break;
        case '2' : // line
        {
            int imgSize, lineRow, lineCol, lineLength, lineThickness;

            cout << "Enter rows and columns in image (the same amount): ";
            cin >> imgSize;
            cout << "Enter row and col coordinates of the top of the line: ";
            cin >> lineRow >> lineCol;
            cout << "Enter line length and line thickness: ";
            cin >> lineLength >> lineThickness;

            Image whiteLine = line(imgSize, imgSize, lineRow, lineCol, lineLength, lineThickness);
            Image mag2(imgSize, imgSize);
```

```
        Image phase2(imgSize, imgSize);

        fft(whiteLine, mag2, phase2);
        writeImage(whiteLine, "blackLine.pgm");
        writeImage(mag2, "blackLineMag.pgm");
        automaticLog(mag2);
        writeImage(mag2, "blackLineLogMag.pgm");
    }
    break;

    case '3' : //lowpass filtering
    {
        // Do 3x3 5x5 and 7x7 gauss filter sigma = 1.5
        const float SD = 1.5;
        for(int WINDOW_SIZE = 3; WINDOW_SIZE < 8; WINDOW_SIZE += 2)
        {
            char trialNum[2];
            sprintf(trialNum,"%d", WINDOW_SIZE);
            strcat(outputName, "SpatialGauss");
            strcat(outputName, trialNum);
            strcat(outputName, ".pgm");

            writeImage(lpGaussian(IN_IMG, WINDOW_SIZE, SD), outputName);

            outputName[NAME_SIZE - 4] = '\0';
        }

        // Do median 3x3, 5x5, and 7x7
        for(int WINDOW_SIZE = 3; WINDOW_SIZE < 8; WINDOW_SIZE += 2)
        {
            char trialNum[2];
            sprintf(trialNum,"%d", WINDOW_SIZE);
            strcat(outputName, "median");
            strcat(outputName, trialNum);
            strcat(outputName, ".pgm");

            writeImage(median(IN_IMG, WINDOW_SIZE), outputName);

            outputName[NAME_SIZE - 4] = '\0';
        }

        float D_0[3];
        float n[3];
        // Do Gaussian frequency filtering
        cout << "Enter 3 D_0s for Gaussian: ";
        cin >> D_0[0] >> D_0[1] >> D_0[2];
        for(size_t freq = 0; freq < 3; ++freq)
        {
            char trialNum[2];
            sprintf(trialNum,"%d", freq);
            strcat(outputName, "LPfreqGaussian");
            strcat(outputName, trialNum);
            strcat(outputName, ".pgm");

            writeImage(lpGaussian(IN_IMG, D_0[freq]), outputName);

            outputName[NAME_SIZE - 4] = '\0';
        }
        // Do butterworth frequency filtering
        cout << "Enter 3 D_0s for Butter: ";
        cin >> D_0[0] >> D_0[1] >> D_0[2];
        cout << "Enter 3 orders for Butter: ";
        cin >> n[0] >> n[1] >> n[2];
        for(size_t trial = 0; trial < 3; ++trial)
        {
            char trialNum[2];
            sprintf(trialNum,"%d", trial);
            strcat(outputName, "LPbutterworth");
            strcat(outputName, trialNum);
            strcat(outputName, ".pgm");

            writeImage(lpButterworth(IN_IMG, D_0[trial], n[trial]), outputName);

            outputName[NAME_SIZE - 4] = '\0';
        }

        delete [] outputName;
    }
    break;

    case '4' : // high pass filtering
    {
        //sobel
        strcat(outputName, "sobel");
        strcat(outputName, ".pgm");

        writeImage(sobel(IN_IMG), outputName);

        outputName[NAME_SIZE - 4] = '\0';

        //um
        int sizes[3];
        cout << "Enter three sizes for um: ";
        cin >> sizes[0] >> sizes[1] >> sizes[2];
        for(size_t trial = 0; trial < 3; ++trial)
        {
            char trialNum[2];
            sprintf(trialNum,"%d", trial);
            strcat(outputName, "um");
            strcat(outputName, trialNum);
            strcat(outputName, ".pgm");

            writeImage(um(IN_IMG, sizes[trial]), outputName);

            outputName[NAME_SIZE - 4] = '\0';
        }

        //laplace
        strcat(outputName, "laplacian");
        strcat(outputName, ".pgm");

        writeImage(laplacian(IN_IMG), outputName);

        outputName[NAME_SIZE - 4] = '\0';
```

```
    int D_0[3], n[3];
    // Do Gaussian frequency filtering
    cout << "Enter 3 D_0s for Gaussian: ";
    cin >> D_0[0] >> D_0[1] >> D_0[2];
    for(size_t freq = 0; freq < 3; ++freq)
    {
        char trialNum[2];
        sprintf(trialNum,"%d", freq);
        strcat(outputName, "HPfreqGaussian");
        strcat(outputName, trialNum);
        strcat(outputName, ".pgm");

        writeImage(hpGaussian(IN_IMG, D_0[freq]), outputName);

        outputName[NAME_SIZE - 4] = '\0';
    }

    // Do butterworth frequency filtering
    cout << "Enter 3 D_0s for Butter: ";
    cin >> D_0[0] >> D_0[1] >> D_0[2];
    cout << "Enter 3 orders for Butter: ";
    cin >> n[0] >> n[1] >> n[2];
    for(size_t trial = 0; trial < 3; ++trial)
    {
        char trialNum[2];
        sprintf(trialNum,"%d", trial);
        strcat(outputName, "HPbutterworth");
        strcat(outputName, trialNum);
        strcat(outputName, ".pgm");

        writeImage(hpButterworth(IN_IMG, D_0[trial], n[trial]), outputName);

        outputName[NAME_SIZE - 4] = '\0';
    }
}
break;

case '5' : //convolution and correlation timing
    // convolution
    for(int maskSize = 5; maskSize < 26; maskSize += 20)
    {
        // do spatial, time it
        clock_t startTime = clock();
        Image spatial = average(IN_IMG, maskSize);
        double time = double(clock() - startTime)/double(CLOCKS_PER_SEC);

        char trialNum[3];
        sprintf(trialNum,"%d", maskSize);
        strcat(outputName, "spatialAverage");
        strcat(outputName, trialNum);
        strcat(outputName, ".pgm");
        cout << outputName << ": " << time << endl;

        writeImage(spatial, outputName);

        outputName[NAME_SIZE - 4] = '\0';

        // do frequency, time it
        startTime = clock();
        Image mask(maskSize, maskSize);
        mask.initImage(1);
        const float SIZE_SIZE = maskSize*maskSize;
        for(int row = 0; row < maskSize; ++row)
            for(int col = 0; col < maskSize; ++col)
                mask(row, col) = mask(row, col)/SIZE_SIZE;
        Image freq = freqSpatialMaskFilter(IN_IMG, mask);
        time = double(clock() - startTime)/double(CLOCKS_PER_SEC);

        strcat(outputName, "frequencyAverage");
        strcat(outputName, trialNum);
        strcat(outputName, ".pgm");
        cout << outputName << ": " << time << endl;

        writeImage(freq, outputName);

        outputName[NAME_SIZE - 4] = '\0';

        // show difference between the pair (log and not)
        Image difference(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                difference(row, col) = abs(spatial(row, col) - freq(row, col));

        float sum = 0;
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                sum += difference(row, col);
        cout << "SUM of abs(differnce): " << sum << endl;

        strcat(outputName, "difference");
        strcat(outputName, trialNum);
        strcat(outputName, ".pgm");
        writeImage(difference, outputName);
        outputName[NAME_SIZE - 4] = '\0';

        strcat(outputName, "autoLogDifference");
        strcat(outputName, trialNum);
        strcat(outputName, ".pgm");
        automaticLog(difference);
        writeImage(difference, outputName);
        outputName[NAME_SIZE - 4] = '\0';
    }

    // now do correlation
    // do spatial, time it
    {
        // create a normalized input image for autocorrelation (used in frequency too)
        Image normalizedInImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());
        float sum = 0;
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                sum += IN_IMG(row, col);
        for(int row = 0; row < IN_IMG.getRow(); ++row)
```

```
                for(int col = 0; col < IN_IMG.getCol(); ++col)
                    normalizedInImg(row, col) = IN_IMG(row, col)/sum;

            clock_t startTime = clock();
            Image autoCorr = conv(IN_IMG, flip(normalizedInImg), false);
            double time = double(clock() - startTime)/double(CLOCKS_PER_SEC);

            strcat(outputName, "spatialCorrelation");
            strcat(outputName, ".pgm");
            cout << outputName << ": " << time << endl;

            writeImage(autoCorr, outputName);

            outputName[NAME_SIZE - 4] = '\0';

            // now do frequency
            startTime = clock();
            Image freqCorr = freqSpatialMaskFilter(IN_IMG, normalizedInImg, false);
            time = double(clock() - startTime)/double(CLOCKS_PER_SEC);

            strcat(outputName, "frequencyCorrelation");
            strcat(outputName, ".pgm");
            cout << outputName << ": " << time << endl;

            writeImage(freqCorr, outputName);
        }
        break;
    case '6' : //homomorphic filter
    {
        float var[3];
        cout << "Enter high, low, and cutoff: ";
        cin >> var[0] >> var[1] >> var[2];

        strcat(outputName, "HOMOMORPHIC");
        strcat(outputName, ".pgm");

        writeImage(homomorphic(IN_IMG, var[0], var[1], var[2]), outputName);
    }
    break;
    default:   // uh oh
        cout << "Error in switch statement\n";
        exit(3);
    }

    /*
     * Success!
     */
    return 0;
}
```

For both spatial domain / frequency domain,
instead of normalize image, you should pad the image
before convolution. -1.

B. Dip.h

```
/********************************************************************
 * Dip.h - header file of the Image processing library
 *
 * Author: Hairong Qi, hqi@utk.edu, ECE, University of Tennessee
 * Author: Ali Ghezawi, aghezawi@utk.edu (all except cs())
 *
 * Created: 01/22/06
 *
 * Modification:
 *  9/24-25/11 added convolution and frequency stuff
 *  9/5/11 added gaussian noise function and point-based image enhancement
 *  8/22/11 added oil, swirl, edge, pixelMatrixMult
 ********************************************************************/

#ifndef DIP_H
#define DIP_H

#include "Image.h"

#define PI 3.1415926


/*
 * Frequency filtering
 */
//uses high pass guassian as base to do homomorphic filtering
Image homomorphic(const Image&,       // input image to filter
    const float&,                     // high frequency
    const float&,                     // low frequency
    const float&);                    // cutoff frequency

// filters image in freq domain based on spatial mask
Image freqSpatialMaskFilter(const Image&,     // image to filter (regular size)
    const Image&,                             // mask (spatial domain, regular size) size(arg1) > size(arg2)
    const bool& isCONV = true);               // whether to convolve or correlalate

// convolves IN_IMG with MASK in frequency domain using lp butterworth
Image lpButterworth(const Image&,    //image to convolve mask with
    const float&,                    //cutoff freq
    const int&);                     //order

// convolves IN_IMG with MASK in frequency domain using lp gaussian
Image lpGaussian(const Image&,    // image to convolve mask with
    const float&);                // cutoff freq

// convolves IN_IMG with MASK in frequency domain using hp butterworth
Image hpButterworth(const Image&,    //image to convolve mask with
    const float&,                    //cutoff freq
    const int&);                     //order

// convolves IN_IMG with MASK in frequency domain using hp gaussian
Image hpGaussian(const Image&,    // image to convolve mask with
    const float&);                // cutoff freq

// computes the 'convolution' and outputs power ratio
Image filter(const Image&,                  //regular-sized image to 'convolve' with
    const Image&,                           //padded mask in frequency domain
    const bool& SUPPRESS = true,            // whether to suppress the power ratio output
    const Image& MASK_PHASE = Image(1,1),   // mask phase, defaults to 1by1, indicating not to use it
    const bool& isCONV = true,              //whether to convole or correlate
    const int& EXTRACT_OFFSET = 0);         //offset in extracting the final image to get 'same' convolution.

Image padPow2(const Image&,            //input image to resize with zeros (also makes sure power of 2)
    const Image& GOAL = Image(1,1));   //defines a goal row by col to resize to (1,1) means none

/*
 * Spatial filtering
 */
Image average(const Image&, // image to perform regular average filter on
    const int&);            // mask defining neighborhood size

Image um(const Image& IN_IMG,  // input image to filter
    const int& MASK_SIZE);     // filter size for average

Image laplacian(const Image& IN_IMG); // input image to filter

Image sobel(const Image&,          // image to filter
    const bool& APPROX = false);   // whether to approximate (for speed)

Image lpGaussian(const Image&, // image to filter
    const int&,                // mask size for gassuain mask
    const float&);             // standard deviation of gaussian filter

Image median(const Image&,     // image to filter
    const int&);               // mask size for median search

// computes median of a single neighborhood.
float median(const Image&,    // image to compute the median within
    const int&,               // mask size (area to compute median within)
    const int&,               // row to center around
    const int&,               // col to center around
    const int&);              // channel to work in
/*
 * Convolution
 */
// convolves IN_IMG with MASK, dividing each neighborhood multiplication by COEF
// NOTE: assume flipping the mask results in the mask
Image conv(const Image&,             //image to convolve mask with
    const Image&,                    //mask
    const bool&);                    //normalize mask first?

// sum(sum(MASK*neighborhoodOfPixel)). Mask nbyn, n%2 !=0
float pixelMatrixMult(const Image&, //input image where operation is taking place
    const Image&,                   //mask that defines the area of the operation
    const int&,                     //row in param1 where operation is centered around
    const int&,                     //col in param1 where operation is centered around
    const int&);                    //the channel under which the operation is taking place

/*
 * Line
 */
```

```
// returns black image with white vertical line in it
Image line(const int&, // rows in image
    const int&,        // cols in image
    const int&,        // row coordinate of top of line
    const int&,        // col coordinate of top of line
    const int&,        // length of line (vertical)
    const int&);       // thickness of line (horizontal)

/*
 * White Square
 */
// returns arg1 with white square at center sized arg2 by arg2.
Image square(const int&, // row# of picture
    const int&,          // col# of picture
    const int&);        // length of square to put at the center of image.

// calculates a scaling factor then does log transfer on input. Input will then
// contain a scaled, log-transformed image.
void automaticLog(Image&);

/*
 * Gaussian Noise Functions
 */
// Returns a single sample of a Gaussian random variable with mean 0
float gaussianNoise(const float&);

// returns input image with noise from gaussianNoise added to it.
Image addGaussNoise(const Image&, const float&);    // input image

/*
 * point-based image enhancement processing
 * note: all can work for RGB pictures, but the
 * algorithms are designed for graycale.
 */
// samples an image by S
Image sampling(const Image&,  // input image
    const int&);              // S

// quantizes an image's levels
Image quantization(const Image&,  // input image
    const int&);                  // # of levels, > 1

// performs log transformation on each pixel
Image logtran(const Image&,       // input image
    const float&);                // scaling factor

// performs power law (gamma) transformation on each pixel
Image powerlaw(const Image&,  // input image
    const float&,             // scaling factor
    const float&);            // gamma

// performs contrast stretching
Image cs(const Image &,
    const float&,                  // slope
    const float&);                 // intercept

// performs threshold effect
Image threshold(const Image&,      // input image
    const float&,                   // bottom threshold
    const float&);                  // top threshold

// performs bitplane slicing
Image bitplane(const Image&,     // input image
    const int&);                  // BIT to slice

// performs histogram equalization
Image histeq(const Image& ); // input image

// saves histogram of image to a file
void saveHist(const Image&,   // input image to take hist of
    const int&,               // what channel to operate in
    char*);                   // filename to save hist data to

// performs localized histogram equalization. Each pixel
// has histeq() done to it using a ROW by COL box around
// that pixel. ROW & COL must be odd & > 1
Image localHistEq(const Image&,     // input image
    const int&,                     // ROW
    const int&);                    // COL

// Swirls an image about its center.
Image swirl(const Image&,            // Input image
    const float&);                   // swirl coefficient

// outputs an image with edges highlighted.
Image edge(const Image& IN_IMG);     // Input image

// Makes an image look as if it were painted with oil paints.
Image oil(const Image&,              // Input image
    const int&);                     // arg2 = (s-1)/2 of search sqr
#endif
```

C. square.cpp

```
/**************************************************************
 * square.cpp: generates black image with white square at center
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/23/11
 *
 **************************************************************/

#include "Dip.h"

/*
 * generates black img with white sqr at center
 * @param1 image row#
 * @param2 image col#
 * @param3 the length of the side; <= min(row, col) of input img
 * @return image with white square imposed unto it
 */
Image square(const int& ROW, const int& COL, const int& SIDE_LENGTH)
{
    if(ROW < SIDE_LENGTH || COL < SIDE_LENGTH)
    {
        std::cout << "Square::invalid input" << std::endl;
        return Image(ROW, COL);
    }

    Image outImg(ROW, COL);

    const int ROW_START = ROW/2 - SIDE_LENGTH/2;
    const int COL_START = COL/2 - SIDE_LENGTH/2;
    const int ROW_END = ROW_START + SIDE_LENGTH;
    const int COL_END = COL_START + SIDE_LENGTH;

    for(int row = ROW_START; row < ROW_END; ++row)
        for(int col = COL_START; col < COL_END; ++col)
            outImg(row, col) = 255;

    return outImg;
}

// calculates a scaling factor then logscales
void automaticLog(Image& inImg)
{
    float max = 0;
    for(int row = 0; row < inImg.getRow(); ++row)
        for(int col = 0; col < inImg.getCol(); ++col)
            if(max < inImg(row, col))
                max = inImg(row,col);

    float c = 255/std::log(max);

    for(int row = 0; row < inImg.getRow(); ++row)
        for(int col = 0; col < inImg.getCol(); ++col)
            inImg(row, col) = std::floor(c*std::log(inImg(row, col)) + .5);
}
```

D. line.cpp

```
/***************************************************************
 * line.cpp: generates vertical white line on black image.
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/23/11
 *
 ***************************************************************/

#include "Dip.h"

/*
 * generates black img with white sqr at center
 * @param1 image row#
 * @param2 image col#
 * @param3 top row coordinate of line
 * @param4 top col coordinate of the line (thickness expands to the right of this point)
 * @param5 line length
 * @param6 line thickness
 * @return image with white square imposed unto it
 *
 * e.g. 128 by 128 img with black line on far left edge with 2 pixel thickness:
 * line(128,128, 0, 0, 128, 2)
 */
Image line(const int& IMG_ROW, const int& IMG_COL, const int& LINE_ROW, const int& LINE_COL, const int& LENGTH, const int& THICKNESS)
{
    // check input
    if(IMG_COL < LINE_COL + THICKNESS || IMG_ROW < LINE_ROW + LENGTH)
    {
        std::cout << "line::invalid inputs" << std::endl;
        return Image(IMG_ROW, IMG_COL);
    }

    Image outImg(IMG_ROW, IMG_COL);

    const int END_ROW = LINE_ROW + LENGTH;
    const int END_COL = LINE_COL + THICKNESS;

    for(int row = LINE_ROW; row < END_ROW; ++row)
        for(int col = LINE_COL; col < END_COL; ++col)
            outImg(row, col) = 255;

    return outImg;
}
```

E. conv.cpp

```
/***************************************************************
 * conv.cpp: convolves image with mask
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 08/22/11
 *
 * Modified 9/24/11 to change the mask being an array to an Image.
 *                 Also changed name from pixelMatrixMult to conv => added conv()
 *
 ***************************************************************/

#include "Dip.h"

/*
 * convolves IN_IMG with MASK, dividing each neighborhood multiplication by COEF
 * @param1 image to convolve mask with row & col of arg1 >= those of arg2
 * @param2 mask
 * @param3 whether to noramlize mask
 * @return convolved image
 * NOTE: assume flipping the mask results in the mask. Hence, conv() is correlation
 */
Image conv(const Image& IN_IMG, const Image& MASK, const bool& NORMALIZE)
{
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());
    Image newMask(MASK.getRow(), MASK.getCol(), MASK.getChannel());

    if(NORMALIZE)
    {
        float sum = 0;
        for(int row = 0; row < MASK.getRow(); ++row)
            for(int col = 0; col < MASK.getCol(); ++col)
                sum += MASK(row, col);
        if(!sum)
            std::cout << "conv::Impossible to normalize" << std::endl;
        else
            for(int row = 0; row < MASK.getRow(); ++row)
                for(int col = 0; col < MASK.getCol(); ++col)
                    newMask(row,col) = MASK(row,col)/sum;
    }
    else
        newMask = MASK;

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col, chan) = pixelMatrixMult(IN_IMG, newMask, row, col, chan);

    return outImg;
}

/*
 * Performs neighborhood multiplication then sums it.
 * @param1 input image where operation is taking place
 * @param2 mask that defines the area of the operation
 * @param3-param4 are row/col in param1 where operation is centered around
 * @param5 is the channel under which the operation is taking place
 * Invalid alignments of param1&2 are taken to be zero.
 */
float pixelMatrixMult(const Image& IN_IMG, const Image& MASK, const int& ROW, const int& COL, const int& CHAN)
{
    float pixelOut = 0.0;

    // if mask size is even, add 1 to it, calculate all points except for starting ones as if it were even + 1
    // e.g. a mask of 2x2 is the same as a 3x3 except ++row_start and ++col_start.
    int evenSize;
    if(MASK.getRow()%2)
        evenSize = MASK.getRow();
    else
        evenSize = MASK.getRow() + 1;

    const int SHIFT =  (evenSize - 1)/2;

    const int ROW_END = SHIFT + ROW < IN_IMG.getRow() ? SHIFT + ROW + 1 : IN_IMG.getRow();
    const int COL_END = SHIFT + COL < IN_IMG.getCol() ? SHIFT + COL + 1 : IN_IMG.getCol();
    int rowStart;
    int colStart;
    if(MASK.getRow()%2)
    {
        rowStart = ROW - SHIFT > -1 ? ROW - SHIFT : 0;
        colStart = COL - SHIFT > -1 ? COL - SHIFT : 0;
    }
    else
    {
        rowStart = ROW - SHIFT + 1 > -1 ? ROW - SHIFT + 1 : 0;
        colStart = COL - SHIFT + 1 > -1 ? COL - SHIFT + 1 : 0;
    }

    //mapping into mask is different for even or odd mask
    const int ROW_FACTOR = MASK.getRow()%2 ? -ROW + SHIFT : -ROW + SHIFT - 1;
    const int COL_FACTOR = MASK.getCol()%2 ? -COL + SHIFT : -COL + SHIFT - 1;

    for(int row = rowStart; row < ROW_END; ++row)
        for(int col = colStart; col < COL_END; ++col)
            pixelOut += IN_IMG(row, col, CHAN)*MASK(row + ROW_FACTOR, col + COL_FACTOR);

    return pixelOut;
}
```

F. spatialFilter.cpp

```cpp
/*************************************************************
 * spatialFilter.cpp: has spatial filters in it
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/24/11
 *          09/25/11 added sobel/laplacian/unsharp masking/average filter
 *
 *************************************************************/

#include "Dip.h"
#include <cmath>
#include <algorithm>

/*
 * filters image with gaussian average filter
 * @param1 image to filter
 * @param2 size of gaussian mask used
 * @param3 standard deviation in the generation of the mask
 * @return filtered image
 */
Image lpGaussian(const Image& IN_IMG, const int& MASK_SIZE, const float& SD)
{
    Image mask(MASK_SIZE, MASK_SIZE);
    const float SHIFT = (MASK_SIZE-1)/2;

    const float SDSD2 = 2*SD*SD;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = std::exp(-(std::pow(row-SHIFT,2)+std::pow(col-SHIFT,2))/SDSD2)/PI/SDSD2;

    std::cout << mask << std::endl;

    return conv(IN_IMG, mask, true);
}

/*
 * Applies median filtering to input image.
 * @param1 input image to filter
 * @param2 square neighborhood size to use during filtering (odd)
 */
Image median(const Image& IN_IMG, const int& MASK_SIZE)
{
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col, chan) = median(IN_IMG, MASK_SIZE, row, col, chan);

    return outImg;
}

/*
 * Returns the median of a neighborhood defined about a point in an image
 * @param1 image neighborhood resides in
 * @param2 square neighborhood size (odd)
 * @param3 & @param4 & parm5 row, col, chan to operate in/around
 */
float median(const Image& IN_IMG, const int& MASK_SIZE, const int& ROW, const int& COL, const int& CHAN)
{
    const int SHIFT =  (MASK_SIZE - 1)/2;
    const int SHIFT_COL = SHIFT + COL;
    const int SHIFT_ROW = SHIFT + ROW;
    const int ROW_END = SHIFT_ROW < IN_IMG.getRow() ? SHIFT_ROW + 1 : IN_IMG.getRow();
    const int COL_END = SHIFT_COL < IN_IMG.getCol() ? SHIFT_COL + 1 : IN_IMG.getCol();
    const int ROW_START = ROW - SHIFT > -1 ? ROW - SHIFT : 0;
    const int COL_START = COL - SHIFT > -1 ? COL - SHIFT : 0;

    const int VOTERS = (ROW_END - ROW_START)*(COL_END - COL_START);

    float* voters = new float[VOTERS];

    int voteNumber = 0;
    for(int row = ROW_START; row < ROW_END; ++row)
        for(int col = COL_START; col < COL_END; ++col)
            voters[voteNumber++] = IN_IMG(row, col, CHAN);

    std::sort(voters, voters + VOTERS);

    float pixelValue = VOTERS%2 ? voters[VOTERS/2] : (voters[VOTERS/2] + voters[VOTERS/2 - 1])/2;

    delete [] voters;

    return pixelValue;
}

/*
 * Filters image with sobel mask
 * @param1 input image
 * @param2 whether to approximate (default fase)
 * @return filtered image
 */
Image sobel(const Image& IN_IMG, const bool& APPROX)
{
    Image outImgX,
        outImg; // stores outImg and outImgy

    Image sobelxOry(3,3);

    // x first
    for(int row = 0; row < 3; ++row)
        for(int col = 0; col < 3; ++col)
            sobelxOry(row, col) = -1 + col + (col - 1)*(row%2);

    outImgX = conv(IN_IMG, sobelxOry, false);

    // now y
    for(int row = 0; row < 3; ++row)
        for(int col = 0; col < 3; ++col)
            sobelxOry(row, col) = -1 + row + (row - 1)*(col%2);
```

```
    outImg = conv(IN_IMG, sobelxOry, false);

    if(APPROX)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col) = std::abs(outImg(row, col)) + std::abs(outImgX(row, col));
    else
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                outImg(row, col) = std::sqrt(outImg(row, col)*outImg(row, col) + outImgX(row, col)*outImgX(row, col));

    return outImg;
}

/*
 * Performs unsharp masking filtering
 * @param1 input image to filter
 * @param2 mask size of filter
 */
Image um(const Image& IN_IMG, const int& MASK_SIZE)
{
    return IN_IMG + (IN_IMG - average(IN_IMG, MASK_SIZE));
}

/*
 * Performs Laplacian filtering
 * @param1 input image to filter
 * @return filtered image
 */
Image laplacian(const Image& IN_IMG)
{
    Image mask(3,3);
    for(int row = 0; row < 3; ++row)
        for(int col = 0; col < 3; ++col)
            mask(row,col) = ((row%2)|(col%2)) - 5*(row%2)*(col%2);

    return conv(IN_IMG, mask, false);
}

/*
 * filters image with standard average filter
 * @param1 image to filter
 * @param2 size of mask to use
 * @return filtered image
 */
Image average(const Image& IN_IMG, const int& MASK_SIZE)
{
    Image mask(MASK_SIZE, MASK_SIZE);
    mask.initImage(1);

    return conv(IN_IMG, mask, true);
}
```

G. *freqFilter.cpp*

```
/***************************************************************
 * freqFilter.cpp: has frequency filters in it
 *
 * Author: Ali Ghezawi (C) aghezawi@utk.edu
 *
 * Created: 09/24/11
 *          09/25/11 added freqSpatialMaskFilter
 *
 ***************************************************************/

#include <complex>
#include "Dip.h"


/*
 * convolves IN_IMG with MASK in frequency domain using lp butterworth
 * @param1 image to convolve mask with
 * @param2 cutoff freq
 * @param3 order
 * @return convolved image
 */
Image lpButterworth(const Image& IN_IMG, const float& D_0, const int& N)
{
    // generate filter
    const int ROW = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (mask.getRow()-1)/2;
    const float COL_SHIFT = (mask.getCol()-1)/2;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = 1/(1 + std::pow(std::abs(std::complex<float>(row - ROW_SHIFT, col - COL_SHIFT))/D_0, 2*N));

    // filter
    std::cout << D_0 << " and " << N <<  ": ";

    return filter(IN_IMG, mask, false);
}

/*
 * convolves IN_IMG with MASK in frequency domain using lp gaussian
 * @param1 image to convolve mask with
 * @param2 cutoff freq
 * @return convolved image
 */
Image lpGaussian(const Image& IN_IMG, const float& D_0)
{
    // generate filter
    const int ROW = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (mask.getRow()-1)/2;
    const float COL_SHIFT = (mask.getCol()-1)/2;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = std::exp(-std::pow(std::abs(std::complex<float>(row - ROW_SHIFT, col - COL_SHIFT))/D_0, 2)/2);

    // filter
    std::cout << D_0 << ": ";

    return filter(IN_IMG, mask, false);
}

/*
 * convolves IN_IMG with MASK in frequency domain using hp gaussian
 * @param1 image to convolve mask with
 * @param2 cutoff freq
 * @return convolved image
 */
Image hpGaussian(const Image& IN_IMG, const float& D_0)
{
    // generate filter
    const int ROW = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (mask.getRow()-1)/2;
    const float COL_SHIFT = (mask.getCol()-1)/2;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = 1 - std::exp(-std::pow(std::abs(std::complex<float>(row - ROW_SHIFT, col - COL_SHIFT))/D_0, 2)/2);

    // filter
    std::cout << D_0 << ": ";

    return filter(IN_IMG, mask, false);
}

/*
 * convolves IN_IMG with MASK in frequency domain using lp butterworth
 * @param1 image to convolve mask with
 * @param2 cutoff freq
 * @param3 order
 * @return convolved image
 */
Image hpButterworth(const Image& IN_IMG, const float& D_0, const int& N)
{
    // generate filter
    const int ROW = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (mask.getRow()-1)/2;
    const float COL_SHIFT = (mask.getCol()-1)/2;
    for(int row = 0; row < mask.getRow(); ++row)
        for(int col = 0; col < mask.getCol(); ++col)
            mask(row, col) = 1/(1 + std::pow(D_0/std::abs(std::complex<float>(row - ROW_SHIFT, col - COL_SHIFT)), 2*N));

    // filter
    std::cout << D_0 << " and " << N <<  ": ";

    return filter(IN_IMG, mask, false);
}
```

```
/*
 * computes the 'convolution' and outputs power ratio
 * @param1 regular-sized image to 'convolve' with
 * @param2 padded mask in frequency domain (power of 2 >= 2*row and 2*col)
 * @param3 whether to suppress power ratio calculation
 * @param4 the phase of the mask (Image(1,1) means ignore phase of mask)
 * @param5 whether the operation is convolution or correlation
 * @return filtered image, regular size
 */
Image filter(const Image& IN_IMG, const Image& PAD_MASK, const bool& SUPPRESS, const Image& MASK_PHASE, const bool& isCONV, const int& EXTRACT_OFFSET)
{
    float PT = 0, P = 0;

    Image padImg = padPow2(IN_IMG);

    // fft the image
    Image mag(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    Image phase(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    fft(padImg, mag, phase);

    // 'convolve'
    for(int row = 0; row < PAD_MASK.getRow(); ++row)
        for(int col = 0; col < PAD_MASK.getCol(); ++col)
        {
            PT += mag(row,col)*mag(row,col);
            padImg(row, col) = mag(row, col)*PAD_MASK(row, col); // padImg represents output img freq
            P += padImg(row,col)*padImg(row,col);
        }

    // inverse transform
    Image padOutImg(padImg.getRow(), padImg.getCol(), padImg.getChannel());
    if(MASK_PHASE.getRow() == 1) // meaning no mask phase given (for butterworht etc.)
        ifft(padOutImg, padImg, phase);
    else if(isCONV)
    {
        Image outPhase = phase + MASK_PHASE;
        ifft(padOutImg, padImg, outPhase);
    }
    else
    {
        Image outPhase = phase - MASK_PHASE;
        ifft(padOutImg, padImg, outPhase);
    }

    // truncate results
    Image outImg(IN_IMG.getRow(), IN_IMG.getCol(), IN_IMG.getChannel());
    const int END_ROW = EXTRACT_OFFSET + IN_IMG.getRow(); // to get 'same' convolution
    const int END_COL = EXTRACT_OFFSET + IN_IMG.getCol();  // to get 'same' convolution
    for(int row = EXTRACT_OFFSET; row < END_ROW; ++row)
        for(int col = EXTRACT_OFFSET; col < END_COL; ++col)
            outImg(row - EXTRACT_OFFSET, col - EXTRACT_OFFSET) = padOutImg(row, col);

    // output power ratio
    if(!SUPPRESS)
        std::cout << 100*P/PT << std::endl;

    return outImg;
}

/*
 * filters image in freq domain based on spatial mask
 * @param1 image (regular size)
 * @param2 mask (spatial, regular size, must be square)
 */
Image freqSpatialMaskFilter(const Image& IN_IMG, const Image& MASK, const bool& isCONV)
{
    // create frequency version of mask then pass to filter()
    Image padMask = padPow2(MASK, IN_IMG);

    Image magMask(padMask.getRow(), padMask.getCol(), padMask.getChannel());
    Image phaseMask(padMask.getRow(), padMask.getCol(), padMask.getChannel());

    fft(padMask, magMask, phaseMask);

    return filter(IN_IMG, magMask, true, phaseMask, isCONV, int(MASK.getRow()/2));
}

/*
 * Resizes an image to ROW by COL
 * @param1 input image to resize 2 fold if a power of 2 else the next highest power of 2
 * @param2 arg1 will resize to 2*arg1 (or closest pwr of 2) Image(1,1) means no goal
 */
Image padPow2(const Image& IN_IMG, const Image& GOAL)
{
    int padRow, padCol;
    if(GOAL.getRow() == 1)
    {
        padRow = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
        padCol = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    }
    else
    {
        padRow = std::pow(2, float(std::ceil(std::log(float(GOAL.getRow()*2))/std::log(float(2)))));
        padCol = std::pow(2, float(std::ceil(std::log(float(GOAL.getCol()*2))/std::log(float(2)))));
    }

    Image padImg(padRow, padCol, IN_IMG.getChannel());

    for(int chan = 0; chan < IN_IMG.getChannel(); ++chan)
        for(int row = 0; row < IN_IMG.getRow(); ++row)
            for(int col = 0; col < IN_IMG.getCol(); ++col)
                padImg(row, col) = IN_IMG(row, col);

    return padImg;
}

Image homomorphic(const Image& IN_IMG, const float& HI, const float& LO, const float& D0)
{
    // generate filter
    const int ROW = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getRow()*2))/std::log(float(2)))));
    const int COL = std::pow(2, float(std::ceil(std::log(float(IN_IMG.getCol()*2))/std::log(float(2)))));
    Image mask(ROW, COL, IN_IMG.getChannel());
    const float ROW_SHIFT = (ROW-1)/2;
```

```
    const float COL_SHIFT = (COL-1)/2;
    for(int row = 0; row < ROW; ++row)
        for(int col = 0; col < COL; ++col)
            mask(row, col) = (HI-LO)*(1 - std::exp(-std::pow(std::abs(std::complex<float>(row - ROW_SHIFT, col - COL_SHIFT))/D0, 2)))+ LO;

    // filter
    return filter(IN_IMG, mask);
}
```