Programming Assignment 3

3

CS 474

Steven Hernandez and Adam Lychuk

Date due: 11/18/2020

Date handed in: 11/18/2020

Division of work:

Adam Lychuk implemented experiment 2 and 3.

Adam Lychuk wrote the theory, implementation and results on experiment 3 and helped write the theory, implementation and results on experiment 2.

Steven Hernandez implemented experiment 1 and 2.

Steven Hernandez wrote the theory, implementation and results on experiment 1 and helped write the theory, implementation and results on experiment 2.

Experiment 2 had contributions from both teammates due to its challenging implementation.

Theory

Experiment 1

The fourier transform allows the user to take any function f(x) and transform it to a new interpolating polynomial based on the fourier series. This transformation allows the user to decompose the original function f(x) in the frequency domain. This decomposition is possible because of the fourier series nature as a trigonometric interpolating polynomial, or in simpler terms the integral of the sum of cos and sin functions. Each portion of the integral is therefore a separate waveform and their addition results in f(x) transformed to the frequency domain. In our case the Fourier transform is an alternative to convolution for performing image processing tasks. Making transformations in the frequency domain is often less computationally expensive and may provide insight or opportunities to make changes impossible using just convolution. For experiment 1 we used the Discrete version of the transform. Compared to the Continuous Fourier Transform, the DFT (Discrete Fourier Transform) uses the summation instead of the integral of f(x)e^-j2piux/N. Just like the continuous version, DFT also has a forward and inverse version of the transformation. Like the continuous version the only difference is a negative sign in the exponent of e, but here we scale by 1/N for the forward transformation since we are no longer dealing with an infinite series.

$$F(u)=\frac{1}{N}\sum_{x=0}^{N-1}f(x)e^{\frac{-j2\pi ux}{N}},\quad u=0,1,2,...,N-1$$

$$f(x)=\sum_{u=0}^{N-1}F(u)e^{\frac{j2\pi ux}{N}},\quad x=0,1,2,...,N-1$$

Figure 1: Top formula is the Forward DFT
While the bottom is the Inverse DT.

Experiment 2

The fourier transformation can be extended in the discrete case for 2D functions and applied easily to image, as an image can be easily represented as a 2D f(x,y) function. The 2D case is identical to the discrete case except we instead sum along the rows of the f(x,y) function and then sum along the columns of the f(x,y). This combined solution is the full 2D transformation. In the inverse case we also still scale by 1/N due to the nature of the double summation.

$$F(u,v)=\frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)e^{-j2\pi(\frac{ux+vy}{N})},$$

$$f(x,y)=\frac{1}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}F(u,v)e^{j2\pi(\frac{ux+vy}{N})},$$

Figure 2: 2D Forward DFT on top
and 2D Inverse DFT on the bottom

Experiment 3

If the fourier transformation is applied to an image we can separate the magnitude information from the phase information in the frequency domain and then visualize it in the real. To visualize the magnitude we set the real part to the square root of the sum of the square of the real and imaginary parts and the imaginary part to zero and then take the inverse transformation. To visualize the phase we set the real part to the cos of theta and the imaginary part to the sin of theta where theta is the inverse tangent of the imaginary part over the real part. Then again the inverse transformation is taken.

Implementation

Experiment 1

Experiment 1 had three parts to implement. The first one had us understand how to implement the given fft function on a small signal. Hence, an array was made to hold the given values, but they were placed in odd elements of the array such as data[1], data[3] and data[5] as these elements are to hold the real values and the even elements of the array hold the imaginary values. This way of storing data in arrays will be used for the rest of the project. The real values were then multiplied by '-1', property $f(x)(-1)^x \leftrightarrow F(u-N/2)$, in order to center the frequencies. Once this was done, the fft function was called and the respective values were passed on. The values would then be printed out in the terminal to show that the function did work. But it wasn't until the fft function was called once again, but set with a 1 in the isign parameter for the inverse fft that there was confirmation that the program worked since the results were the same real values as the original.

Part two of the experiment had us use values from a given function which in this case was $f(x) = \cos(2\pi ux)$. With 128 samples to use, an array of 257 was made to hold the 128 real and the 128 imaginary values with one extra element of the array not being used due to the nature of the fft function. Once the values were calculated and centered, they were passed on to the fft to perform the forward ft. Part three is implemented very closely like that of part two; however, the values were given in the 'Rect_128.dat'.

Experiment 2

This experiment had us implement the 2D DFT with the separability property in order to transform the images properly. The read and write functions given by the professor were also used in order to implement the program. This program starts by declaring variables to store the size of the dimensions of the image which in this case were 512 pixels and made 2D float arrays that will be used to hold the real and imaginary values of the image. Once made, the pixel values of the image were stored in the real value 2D array and the imaginary 2d array was filled with zeros. Next, the 2D fft function was implemented which is where the separability property was written. In the 2D fft, two 1 dimensional arrays were declared to be the size of the 2 times the length size of row or column and added one. These arrays were made this way since they would be used in the given fft function.

A nested for-loop is started to go through the values and the array for rows is filled first with the respective real and imaginary numbers. Once done, it calls the fft function to pass the array and have it get processed. After the array has the new values, it passes the values into the array to hold. This gets repeated until all the rows of the real and imaginary vectors have been calculated. This whole process gets done once again, but this time around it is for the columns of the image. Returning to the main program, the values from both the real and imaginary array are combined to put them in an array that holds the new values for the new image. After that is done, the setPixelValue is called within a nested for-loop to set the new values.

Experiment 3

In implementing experiment 3 we simply needed to add logic for removing the phase information and logic for removing the magnitude information. The 2D fft is reused from experiment 2 for both the forward and inverse fourier transform. Based on a command line argument the program either removes the phase information or the magnitude information after transforming to the frequency domain with the forward fourier transform. In the case of phase removal each part of the real image is set to the magnitude and then each part of the imaginary image is set to zero. Then the inverse fourier transform transforms the image back and we write it to a file. In the case of magnitude removal we after the forward fourier transform calculate the inverse tangent using atan2() of the imaginary image over the real image for each i,j. We then in the same loop use that theta to make the real part cos(theta) and the imaginary part sin(theta). The resultant image is then written out to a file after we take the inverse fourier transform.

Results and Discussion

Experiment 1

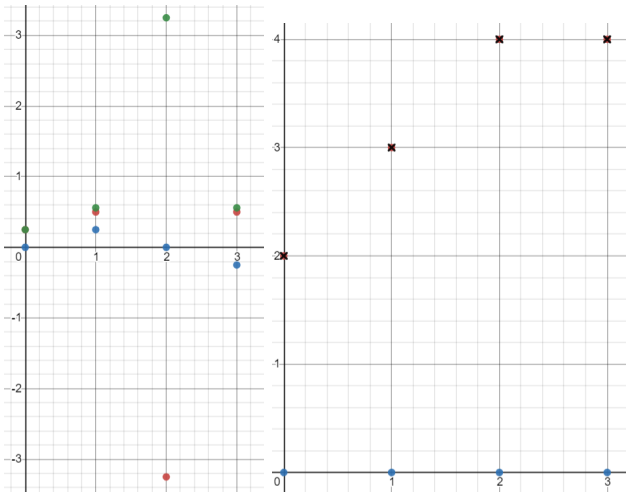

Figure 3: (a) plot for forward dft with red values being real, blue imaginary and green magnitude (b) plot for inverse dft with red values being real, blue imaginary and black Xs are magnitude

The first part of the experiment had us test the fft function with a small sample to make sure the function was working correctly. The first thing to do was the forward dft which gave the results seen in Figure 3(a). Once these values were made, the new values were processed with the inverse dft which gave back the original real value results of 2,3,4,4 as seen in the plot of Figure 3(b).

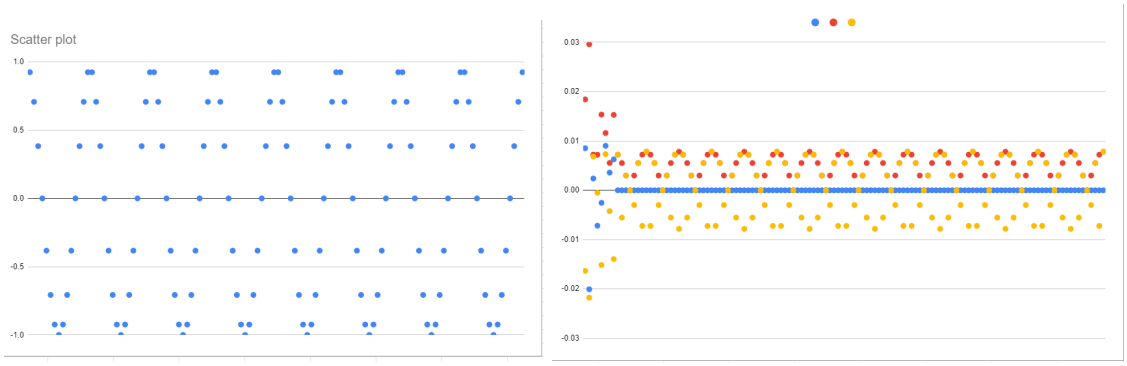

Figure 4: (a) results from the given cosine function (b) real values(yellow), imaginary (blue) and magnitude (red)

Figure 5: Phase of the cosine function

The second part of experiment one gave the function $\cos(2\pi ux)$ to be sampled as seen in Figure 4(a) which was then processed with the dft function. The result as seen in Figure 4(b) shows how the values were able to retain a form of a wave and one of the things that stuck out for us was having to make sure that the magnitude was centered to that of the frequency domain by using the property of $f(x)(-1)^x \leftrightarrow F(u-N/2)$. Figure 5 displays the phase of the values which seem to be quite larger than that of the real, imaginary and magnitude values.

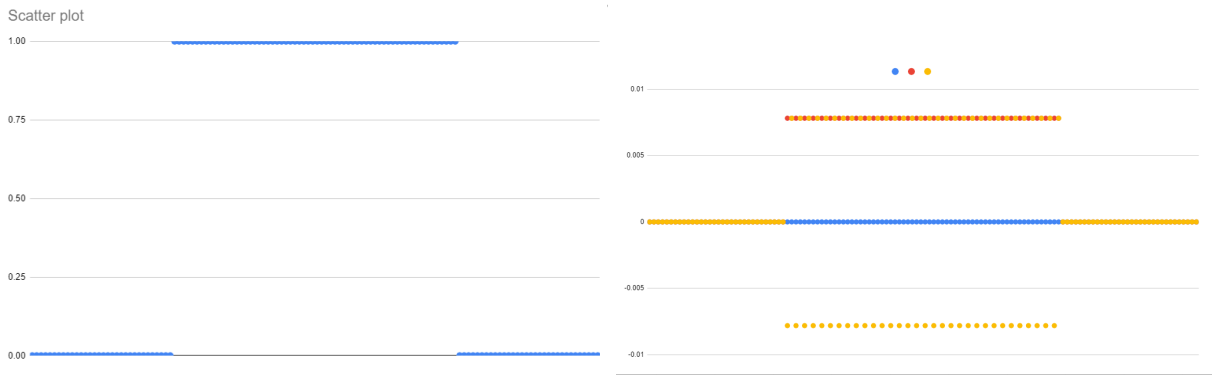

Figure 6: (a) Scatter plot of the Rect_128.dat data (b) Scatter plot of the real values (yellow), imaginary values (blue) and magnitude (red)

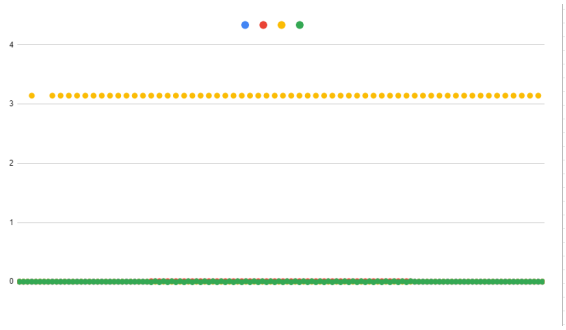

Figure 7: Rect_128 with phase shown in yellow

Part three had us do something very similar to that of part two, but instead of using a cosine function, the values of the rectangle function with 128 samples were given. Figure 6(a) displays the values and the shape of the rectangle function can be seen. After using the DFT, the values, seen in Figure 6(b), seem to reflect what the function will look like in the frequency domain since the real values in yellow have a bigger amplitude towards the middle and reach near zero as it expands from the center. The imaginary values kept close to zero and the magnitude had positive values. The phase can be seen in Figure 7 where the values came out to be larger compared to the rest.

Experiment 2

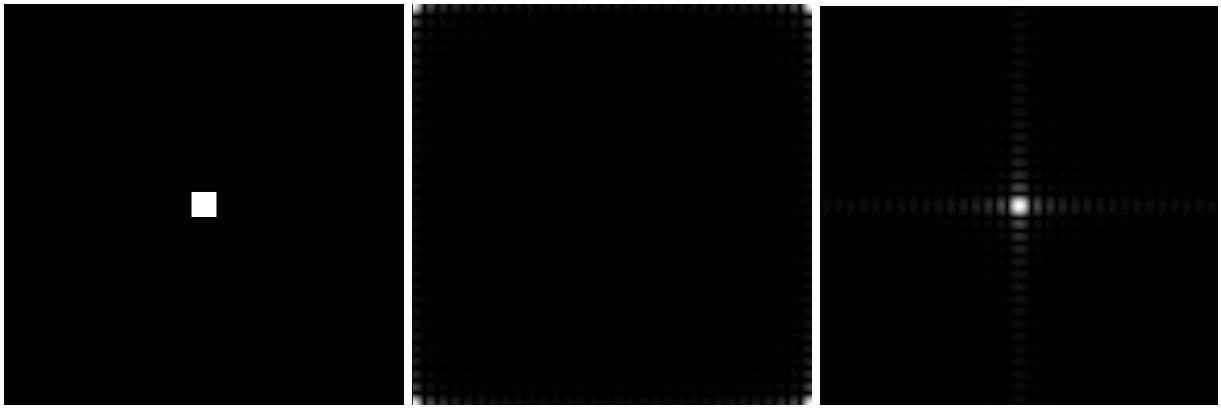

Figure 8: (a) 32x32 square (b)non-centered magnitude (c)Centered magnitude

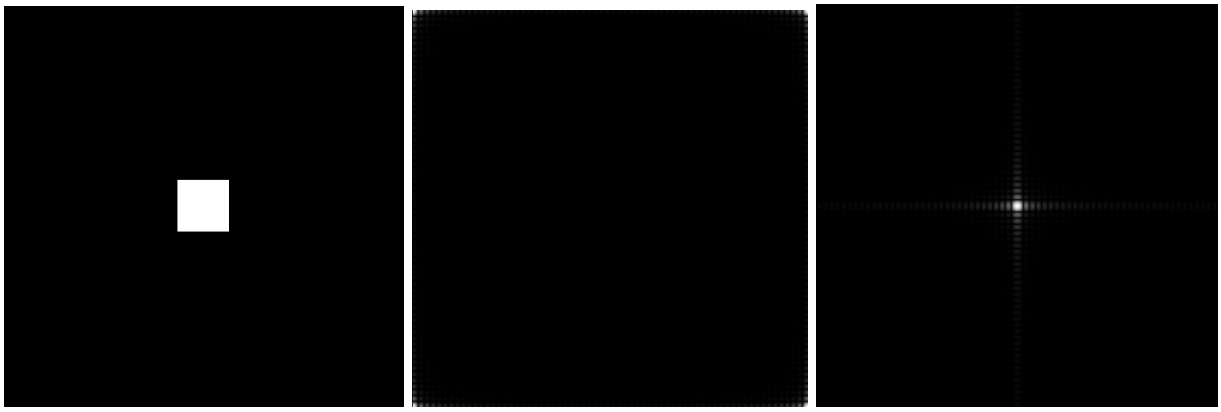

Figure 9: (a) 64x64 square (b) non-centered magnitude (c)Centered magnitude

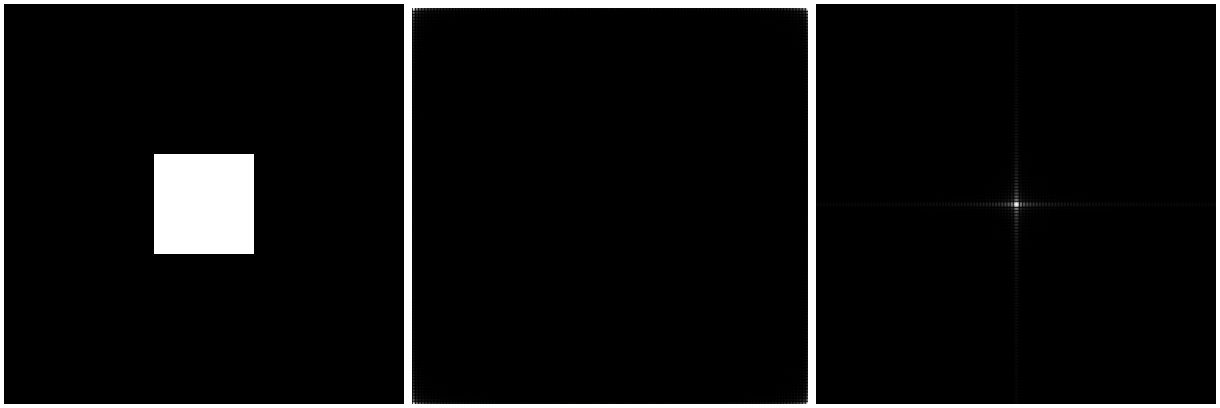

Figure 10: (a) 128x128 square (b) non-centered magnitude (c)Centered magnitude

This experiment had us generate 3 images of size 512x512 with what seems like a black background and a white square in the center. The first part of this experiment had us use a 32x32 square as seen in Figure 8(a) which then had it go through the 2D DFT without shifting the magnitude to the center of the frequency domain. The result of this gave us what seems like a completely black square, Figure 8(b), but if you look closely at the corners of the square, there are hints of the spectrum. Now looking at Figure 8(c), when the magnitude gets shifted so it is centered to the frequency domain, the spectrum is seen very clearly on the center, but like the sinc function, it seems that this spectrum decays the further away it is from the center. Part 2 and 3 of experiment 2 had us do the same process to the image with the 64x64 square, Figure 9(a), and the 128x128 square, Figure 10(a), to be able to compare the results when the magnitude is not shifted and when it is. The results in the non-shifted images, Figure 9(b) and Figure 10(b), show the spectrum is harder to see in the corners than the 64x64 image since there are more pixels in the square. But it is in the third images, Figure 9(c) and 10(c), where it is easier to see what is going on with the spectrum. When comparing this to the sinc function and recalling how it decays in amplitude as 1/x, it helps make more sense as to why the spectrum is more faint as the number of samples is higher.

Experiment 3

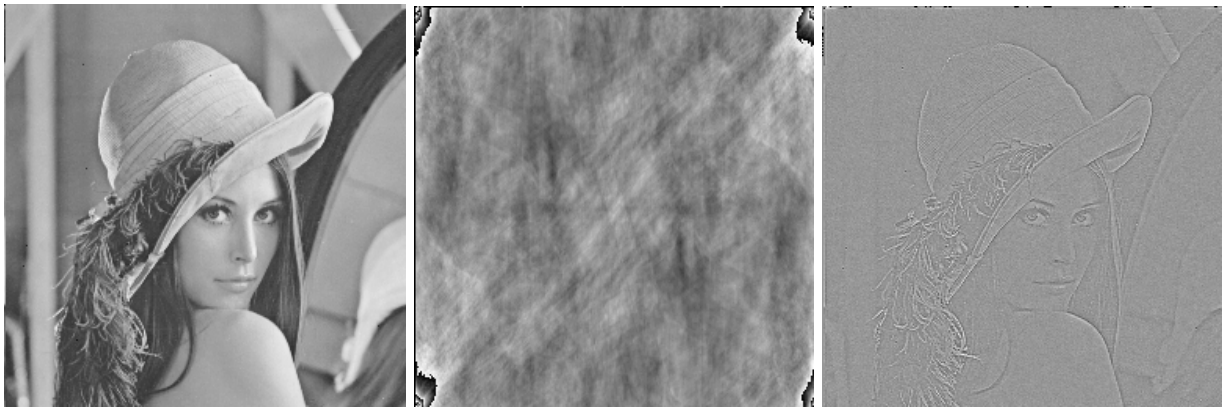

Figure 11: (a) original lenna (b) lenna with phase set to zero (c) lenna with magnitude set to 1

This experiment explored how image information is stored in the frequency domain, as well as the effects of manipulation of magnitude and phase in the frequency domain. In experiment (3.a) we removed the phase information by setting the imaginary part to zero and the real part to the image magnitude. As we can see in figure 11(b) the phase of an image stores its structure. When the phase is thrown out and set to zero, all that remains is the intensity values of the image as stored by the magnitude.

This is also shown in experiment (3.b) where we instead removed the magnitude values and left the phase intact. This was accomplished by setting the real part to cos(theta) and the imaginary part to sin(theta) where theta = tan-1(imag/real) and is shown in figure 11(c). My intuition is that this works because the fourier series is a combination of the addition of cos and sin functions. When we do the inverse fourier transform we have encoded the sin and cos values of the series in the real and imaginary numbers. Therefore we keep the phase information while also setting the magnitude to 1 as cos(theta)-sin(theta)=1. Interestingly the structure information included in the phase are the edges of the image, meaning through just the fourier transform we can obtain image edges without having to do a laplacian convolution.

Source Code

Experiment 1

```
//Steven Hernandez and Adam Lychuk
//CS 474
//Project 3 part 1a,1b and 1c
#include <iostream>
#include <fstream>
#include <cmath>
#include <math.h>
#include <complex>
using namespace std;

#define SWAP(a, b) \
  tempr = (a);     \
  (a) = (b);       \
  (b) = tempr
/*

   The real part is stored in the odd locations of the array
   (data[1], data[3], data[5]. etc) and the imaginary part
   in the even locations (data[2], data[4], data[6], etc.

   The elements in the array data must be stored from data[1]
   to data[2*nn] -  data[0] is not used!

   nn is the length of the input which should be power of 2.
```

```
   Warning: the program does not check this condition.

   isign: -1 Forward FFT, isign: 1  Inverse FFT (based on our definition)

   Warning: the FFT routine provided does not multiply by the normalization
   factor 1/N that appears in the forward DFT equation; you should do this
   yourself (see page 506 from the fft handout).

*/
void fft(float data[], unsigned long nn, int isign)
{
  unsigned long n, mmax, m, j, istep, i;
  double wtemp, wr, wpr, wpi, wi, theta;
  float tempr, tempi;

  n = nn << 1;
  j = 1;
  for (i = 1; i < n; i += 2)
  {
    if (j > i)
    {
      SWAP(data[j], data[i]);
      SWAP(data[j + 1], data[i + 1]);
    }
    m = n >> 1;
    while (m >= 2 && j > m)
    {
      j -= m;
      m >>= 1;
    }
    j += m;
  }
  mmax = 2;
  while (n > mmax)
  {
    istep = mmax << 1;
    theta = isign * (6.28318530717959 / mmax);
    wtemp = sin(0.5 * theta);
    wpr = -2.0 * wtemp * wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;
    for (m = 1; m < mmax; m += 2)
    {
      for (i = m; i <= n; i += istep)
      {
        j = i + mmax;
```

```
        tempr = wr * data[j] - wi * data[j + 1];
        tempi = wr * data[j + 1] + wi * data[j];
        data[j] = data[i] - tempr;
        data[j + 1] = data[i + 1] - tempi;
        data[i] += tempr;
        data[i + 1] += tempi;
      }
      wr = (wtemp = wr) * wpr - wi * wpi + wr;
      wi = wi * wpr + wtemp * wpi + wi;
    }
    mmax = istep;
  }
}
#undef SWAP

int main(int argc, char *argv[])
{
  ////////////////////////////   part a   ////////////////////////////////////

  //signal of 2,3,4,4
  float data[] = {0, 2, 0, 3, 0, 4, 0, 4, 0};

  //multiply real numbers by -1 to center for frequency domain
  for (int i = 0; i < 4; ++i)
  {
    if ((i + 1) % 2)
    {
      data[2 * i + 1] *= -1;
    }
  }

  //calls fft function, passes array, power 4 and signal for forward FT
  fft(data, 4, -1);

  //Complex float variable
  std::complex<float> complex_num;

  //i increases by 2 to only go over real numbers in array
  for (int i = 1; i < 9; i += 2)
  {
    //stores the absolute value of the complex value a+bi
    complex_num = std::complex<float>(data[i], data[i + 1]);
    //Normalize
    complex_num /= 4;
    //stores real value in odd elements of array
    data[i] = complex_num.real();
    //stores imaginary value in even elements of array
```

```
    data[i + 1] = complex_num.imag();
  }

  std::ofstream fout;
  fout.open("function_f.dat");
  for (int i = 0; i < 4; ++i)
  {
    //Magnitude from slides is sqrt((R^2(u))+(I^2(u)))
    //store real, imaginary and magnitude
    fout << data[2 * i + 1] << "," << data[2 * i + 2] << "," << sqrt(pow(data[2 * i
+ 1], 2) + pow(data[2 * i + 2], 2)) << "\n";
  }
  fout.close();
  fout.clear();

  //Compute Inverse FT with the current data
  fft(data, 4, 1);
  for (int i = 0; i < 4; ++i)
  {
    if ((i + 1) % 2)
    {
      data[2 * i + 1] *= -1;
    }
  }

  ////////////////////////////////// end part a //////////////////////////////////

  ////////////////////////////////   part b   //////////////////////////////////
  std::cout << " \n PART B: COSINE FUNCTION" << std::endl;
  //float array of size 257 since the first element is not used
  float f_cos[257];

  //Store the function result in odd elements of array and 0 in even(imaginary)
  for (int i = 0; i < 128; ++i)
  {
    f_cos[2 * i + 1] = cos(2.0 * 3.14159265359 * 8.0 * (float)(i + 1) / 128.0);
    f_cos[2 * i + 2] = 0;
  }
  fout.open("function_cos_128.dat");

  //store values
  for (int i = 0; i < 128; ++i)
  {
    fout << f_cos[2 * i + 1];
  }
  fout.close();
  fout.clear();
```

```
std::cout << std::endl;

//multiply real numbers by -1 to center frequency domain
for (int i = 0; i < 128; ++i)
{
  if ((i + 1) % 2)
  {
    f_cos[2 * i + 1] *= -1;
  }
}

//Forward FT of the cosine results
fft(f_cos, 7, -1); //had 128 in the second parameter

//Goes through the array for odd elements
for (int i = 1; i < 257; i += 2)
{
  //stores the absolute value of the complex value a+bi
  complex_num = std::complex<float>(f_cos[i], f_cos[i + 1]);
  //normalize by 128
  complex_num /= 128.0;
  //store real num
  f_cos[i] = complex_num.real();
  //store imag num
  f_cos[i + 1] = complex_num.imag();
}

//Experiment does not require  inverse FT
fout.open("function_cos_128_fft.dat");

//store values
for (int i = 0; i < 128; ++i)
{
  fout << f_cos[2 * i + 1] << endl;
}
fout << endl;
for (int i = 0; i < 128; ++i)
{
  fout << f_cos[2 * i + 2] << endl;
}
fout << endl;
for (int i = 0; i < 128; ++i)
{
  fout << sqrt(pow(f_cos[2 * i + 1], 2) + pow(f_cos[2 * i + 2], 2)) << endl;
}
fout << endl;
for (int i = 0; i < 128; ++i)
```

```
{
  fout << atan2(f_cos[2 * i + 2], f_cos[2 * i + 1]) << endl;
}
fout.close();
fout.clear();
std::cout << std::endl;

//////////////////////////////// end part b ////////////////////////////////

////////////////////////////////   part c    ////////////////////////////////
std::ifstream fin;
float rect[257];
//Uses data from the given file
fin.open("Rect_128.dat");
//Stores data in the odd elements of the array (real numbers)
if (fin.is_open())
{
  for (int i = 0; i < 128; ++i)
  {
    fin >> rect[2 * i + 1];
    rect[2 * i + 2] = 0;
  }
  fin.close();
}
else
{
  std::cout << "error" << std::endl;
}

//Centers values by using -1
for (int i = 0; i < 128; ++i)
{
  if ((i + 1) % 2)
    rect[2 * i + 1] *= -1;
}

//Applies Forward FT
fft(rect, 7, 1);

//Goes over odd elements of array
for (int i = 1; i < 257; i += 2)
{
  //stores the absolute value of the complex value a+bi
  complex_num = std::complex<float>(rect[i], rect[i + 1]);
  //normalize by 128 (num of values in file),don't normalize if doing inverse
  complex_num /= 128.0;
  //store real num
```

```
    rect[i] = complex_num.real();
    //store imag num
    rect[i + 1] = complex_num.imag();
  }

  std::cout << std::endl;

  fout.open("Rect_128_results.dat");
  //store values
  for (int i = 0; i < 128; ++i)
  {
    fout << rect[2 * i + 1] << endl;
  }
  fout << endl;
  for (int i = 0; i < 128; ++i)
  {
    fout << rect[2 * i + 2] << endl;
  }
  fout << endl;
  for (int i = 0; i < 128; ++i)
  {
    fout << sqrt(pow(rect[2 * i + 1], 2) + pow(rect[2 * i + 2], 2)) << endl;
  }
  fout << endl;
  for (int i = 0; i < 128; ++i)
  {
    fout << atan2(rect[2 * i + 2], rect[2 * i + 1]) << endl;
  }
  fout.close();
  fout.clear();

  ////////////////////////////// end part c ////////////////////////////////////
  return 0;
}
```

Experiment 2

```
for (int i = 0; i < rows; ++i)
{
  //add a new col for each row
  image_real[i + 1] = new float[cols + 1];
  image_imag[i + 1] = new float[cols + 1];

  for (int j = 0; j < cols; ++j)
  {
    //gets the value of the pixel from image
    image.getPixelVal(i, j, val);
```

```
    //insert value into real num array and have 0 on imaginary
    image_real[i + 1][j + 1] = val;
    image_imag[i + 1][j + 1] = 0;

    // image_real[i+1][j+1] *= pow(-1, ((i+1)+(j+1)));
    // if multiplied by -1, it is centered
  } // if multiplied by 1, then it isn't centered
}

int N = rows;
int M = cols;
//call 2D fft and passes the size of row,col, the 2d arrays for
//real and imaginary numbers along with the isign, -1 = forward,
//1 = Inverse
fft2D(M, N, image_real, image_imag, -1);
std::complex<float> complex_num;
for (int i = 1; i < N + 1; ++i)
{
  for (int j = 1; j < N + 1; ++j)
  {
    complex_num = std::complex<float>(image_real[i][j], image_imag[i][j]);
    complex_num /= N * N;

    image_real[i][j] = complex_num.real();
    image_imag[i][j] = complex_num.imag();
  }
}
float **image_out = new float *[rows];
for (int i = 0; i < rows; ++i)
{
  image_out[i] = new float[cols];
  for (int j = 0; j < cols; ++j)
  {
    image_out[i][j] = log(1. + sqrt(pow(image_real[i + 1][j + 1], 2.) +
pow(image_imag[i + 1][j + 1], 2.)));
  }
}

for (int i = 0; i < rows; ++i)
{
  for (int j = 0; j < cols; ++j)
  {
    image.setPixelVal(i, j, image_out[i][j]);
  }
}
writeImage(argv[2], image);
```

Experiment 3

```
//Steven Hernandez and Adam Lychuk
//CS 474
//Project 3 part 2a,2b and 2c
//This part of the assignment has us use the separability property for the
//2D DFT.
#include <iostream>
#include <fstream>
#include <cmath>
#include <math.h>
#include <complex>
using namespace std;

#include "image.h"

int readImageHeader(char[], int &, int &, int &, bool &);
int readImage(char[], ImageType &);
int writeImage(char[], ImageType &);

void fft2D(int N, int M, float **real_Fuv, float **imag_Fuv, int isign);
//This from the function the prof gave us
#define SWAP(a, b) \
  tempr = (a);     \
  (a) = (b);       \
  (b) = tempr
/*

   The real part is stored in the odd locations of the array
   (data[1], data[3], data[5]. etc) and the imaginary part
   in the even locations (data[2], data[4], data[6], etc.

   The elements in the array data must be stored from data[1]
   to data[2*nn] -  data[0] is not used!

   nn is the length of the input which should be power of 2.
   Warning: the program does not check this condition.

   isign: -1 Forward FFT, isign: 1  Inverse FFT (based on our definition)

   Warning: the FFT routine provided does not multiply by the normalization
   factor 1/N that appears in the forward DFT equation; you should do this
   yourself (see page 506 from the fft handout).

*/
void fft(float data[], unsigned long nn, int isign)
```

```
{
  unsigned long n, mmax, m, j, istep, i;
  double wtemp, wr, wpr, wpi, wi, theta;
  float tempr, tempi;

  n = nn << 1;
  j = 1;
  for (i = 1; i < n; i += 2)
  {
    if (j > i)
    {
      SWAP(data[j], data[i]);
      SWAP(data[j + 1], data[i + 1]);
    }
    m = n >> 1;
    while (m >= 2 && j > m)
    {
      j -= m;
      m >>= 1;
    }
    j += m;
  }
  mmax = 2;
  while (n > mmax)
  {
    istep = mmax << 1;
    theta = isign * (6.28318530717959 / mmax);
    wtemp = sin(0.5 * theta);
    wpr = -2.0 * wtemp * wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;
    for (m = 1; m < mmax; m += 2)
    {
      for (i = m; i <= n; i += istep)
      {
        j = i + mmax;
        tempr = wr * data[j] - wi * data[j + 1];
        tempi = wr * data[j + 1] + wi * data[j];
        data[j] = data[i] - tempr;
        data[j + 1] = data[i + 1] - tempi;
        data[i] += tempr;
        data[i + 1] += tempi;
      }
      wr = (wtemp = wr) * wpr - wi * wpi + wr;
      wi = wi * wpr + wtemp * wpi + wi;
    }
```

```
    mmax = istep;
  }
}
#undef SWAP

void min_max_norm(float **image, int w, int h)
{
  float min = 255, max = 0;
  for (int i = 1; i < h+1; ++i)
  {
    for (int j = 1; j < w+1; ++j)
    {
      float pixelval = image[i][j];
      if (min > pixelval)
      {
        min = pixelval;
      }
      if (max < pixelval)
      {
        max = pixelval;
      }
    }
  }
  float scale = 255.0f / (max - min);
  float shift = -scale * min;
  for (int i = 0; i < h+1; ++i)
  {
    for (int j = 0; j < w+1; ++j)
    {
      image[i][j] = scale * image[i][j] + shift;
    }
  }
}

int main(int argc, char *argv[])
{
  int rows, cols, bytes;
  bool type;
  int val;
  readImageHeader(argv[1], rows, cols, bytes, type);
  ImageType image(rows, cols, bytes);
  readImage(argv[1], image);
  //2 double arrays using pointers to store real and imaginary numbers
  float **image_real = new float *[rows + 1];
  float **image_imag = new float *[rows + 1];
  //initialize the first element of the arrays
  image_real[0] = new float[cols + 1];
```

```
image_imag[0] = new float[cols + 1];

for (int i = 0; i < rows; ++i)
{
  //add a new col for each row
  image_real[i + 1] = new float[cols + 1];
  image_imag[i + 1] = new float[cols + 1];

  for (int j = 0; j < cols; ++j)
  {
    //gets the value of the pixel from image
    image.getPixelVal(i, j, val);

    //insert value into real num array and have 0 on imaginary
    image_real[i + 1][j + 1] = val;
    image_imag[i + 1][j + 1] = 0;
    // image_real[i+1][j+1] *= pow(-1, ((i+1)+(j+1)));
    // if multiplied by -1, it is centered
  } // if multiplied by 1, then it isn't centered
}

int N = rows;
int M = cols;
//call 2D fft and passes the size of row,col, the 2d arrays for
//real and imaginary numbers along with the isign, -1 = forward,
//1 = Inverse
fft2D(M, N, image_real, image_imag, -1);

std::complex<float> complex_num;
for (int i = 1; i < N + 1; ++i)
{
  for (int j = 1; j < N + 1; ++j)
  {
    complex_num = std::complex<float>(image_real[i][j], image_imag[i][j]);
    complex_num /= N * N;

    image_real[i][j] = complex_num.real();
    image_imag[i][j] = complex_num.imag();
  }
}
if (atoi(argv[3])==0) {
  // Set phase to zero
  for (int i = 1; i < N+1; ++i) {
    for (int j = 1; j < N+1; ++j) {
      image_real[i][j] = sqrt(pow(image_real[i][j],2) + pow(image_imag[i][j],2));
      image_imag[i][j] = 0;
    }
```

```
    }
  } else {
    // set magnitude to 1
    float theta;
    for (int i = 1; i < N+1; ++i) {
      for (int j = 1; j < N+1; ++j) {
        theta = atan2(image_imag[i][j],image_real[i][j]);
        image_real[i][j] = cos(theta);
        image_imag[i][j] = sin(theta);
      }
    }
  }
  fft2D(M,N, image_real, image_imag, 1);

  if (atoi(argv[3])!=0) {
    min_max_norm(image_real,N,M);
  }

  for (int i = 0; i < rows; ++i)
  {
    for (int j = 0; j < cols; ++j)
    {
      image.setPixelVal(i, j, image_real[i][j]);
    }
  }
  writeImage(argv[2], image);
  return 0;
}

void fft2D(int N, int M, float **real_fuv, float **imag_fuv, int isign)
{
  float *buffer = new float[2 * N + 1];
  for (int i = 0; i < N; ++i)
  {
    for (int j = 0; j < N; ++j)
    {
      buffer[2 * j + 1] = real_fuv[i + 1][j + 1];
      buffer[2 * j + 2] = imag_fuv[i + 1][j + 1];
    }
    fft(buffer, N, isign);
    for (int j = 0; j < N; ++j)
    {
      real_fuv[i + 1][j + 1] = buffer[2 * j + 1];
      imag_fuv[i + 1][j + 1] = buffer[2 * j + 2];
    }
  }
  for (int j = 0; j < N; ++j)
```

```
  {
    for (int i = 0; i < N; ++i)
    {
      buffer[2 * i + 1] = real_fuv[i + 1][j + 1];
      buffer[2 * i + 2] = imag_fuv[i + 1][j + 1];
    }
    fft(buffer, N, isign);
    for (int i = 0; i < N; ++i)
    {
      real_fuv[i + 1][j + 1] = buffer[2 * i + 1];
      imag_fuv[i + 1][j + 1] = buffer[2 * i + 2];
    }
  }
}
```