Assignment #2 Report - 2D FFT and Fourier Domain Processing

Daniel Maienza

Professor Megherbi

Digital Image Processing and Computer Vision

EECE 5841 (Fall 2023)

Date Submitted: 10/19/2023

Date Due: 10/31/2023

I. Objective

The objective of this assignment is to understand the 2-D Fast Fourier Transform (FFT) and its inverse (IFFT), focusing on their applications in image processing (and its computational advantage of $N\log_2 N$ over N^2) . The project begins with an exploration of how to implement these mathematical tools on a synthesized "cake" image. This foundational exercise leads into a more complex task of applying a Butterworth Low Pass Filter (BWLP) to an MRI image. The assignment uses complex numbers in image data arrays while requiring the implementation of data normalization techniques. Additionally, the project places an emphasis on the correct centering of the Fourier Transform and the Butterworth filter in the frequency domain. The aim is to produce smoothed, low-pass-filtered output images while evaluating the effects of different filter parameters. This paper will provide a report detailing the methods, results, and observations, furthering the understanding of Fourier domain processing and its utility in image smoothing.

II. Algorithms/Functions

As this assignment is more sophisticated than the previous one, a Makefile is necessary to manage the several files for this lab's operation. In short, a Makefile is a simple way to manage build systems. It contains a set of rules that specify how to build a software project. In the case of this Makefile, here is the breakdown of its structure:

all: TestFour2floats Lab2_Part1 Lab2_Part2

This line defines a target named "all" that depends on the 'TestFour2floats', 'Lab2_Part1', and 'Lab2_Part2' targets. When "make" is run, it builds these three targets.

TestFour2floats: TestFour2floats.o Four2.o
Lab2_Part1: Lab2_Part1.o Four2.o
Lab2_Part2: Lab2_Part2.o Four2.o

These lines specify that each of the targets depend on 'Four2.o' and their respective object files. The command below each line compiles these object files to create the executable file.

Four2.o: Four2.c Four2.h Four2_Private.h
TestFour2floats.o: TestFour2floats.c Four2.h
Lab2_Part1.o: Lab2_Part1.c Four2.h
Lab2_Part2.o: Lab2_Part2.c Four2.h

The first line specifies that ‘Four2.o’ depends on ‘Four2.c’, ‘Four2.h’, and ‘Four2_Private.h’. If any of these files change, ‘Four2.o’ will be recompiled. This is how the following three lines operate.

clean:

This is a phony target that doesn't represent a file. It's used to clean up the project by removing all .o files.

The rest of the other source files in this assignment’s operation include ‘Four2.h’, ‘Four2_Private.h’, ‘cplx.h’, ‘Four2.c’, ‘TestFour2floats.c’, ‘Lab2_Part1.c’, ‘Lab2_Part2.c’, and the two pgm images, ‘cake.pgm’ and ‘mri.pgm’.

The ‘Four.h’ header file defines the API for the 2D FFT library. It declares the function ‘fft_Four2’, which is the main function for computing 2D FFT and inverse 2D FFT. This function is used in the main programs (TestFour2floats.c, Lab2_Part1.c, and Lab2_Part2.c) to perform the Fourier transforms.

The ‘Four2_Private.h’ header file contains private definitions and function declarations that are used internally within ‘Four2.c’. These are not meant to be accessed directly by external code or the user. It has function declarations for FFT transformations along the X and Y dimensions. They take an integer size argument and return a pointer to an array of complex numbers. It performs FFT shifting, which moves the zero frequency component to the center of the spectrum.

The ‘cplx.h’ header file defines a complex number type and is included in all files that need to work with complex numbers. The ‘typedef struct’ defines a new data type called ‘cplx’, which represents a complex number with real and imaginary parts as floats.

The ‘Four.c’ is the core of the 2D FFT operation. The global variables in this file include, ‘fft_image’: pointer to the working image, ‘fft_sizeX’ and ‘fft_sizeY’: dimensions of the image, and ‘fft_rr’: a working counter. The functions ‘cplx *fft_X(int i)’ and ‘cplx *fft_Y(int i)’ return pointers to complex numbers that represent the elements in the working image along the X and Y axes, respectively. The function ‘void fft_shift()’ shifts the frequency spectrum so that the zero frequency is centered. The main function for the 2D FFT is ‘void fft_Four2(float *im, unsigned sX, unsigned sY, bool inv)’. It first updates the global variables with the image and its dimensions. It then checks if the FFT is forward or inverse (inv flag). Following this, it calls fft_shift() to prepare the image for the FFT. Next, it executes 1D FFTs along each row and each column of the image. If it's an inverse FFT, it shifts the frequencies back and corrects the magnitude.

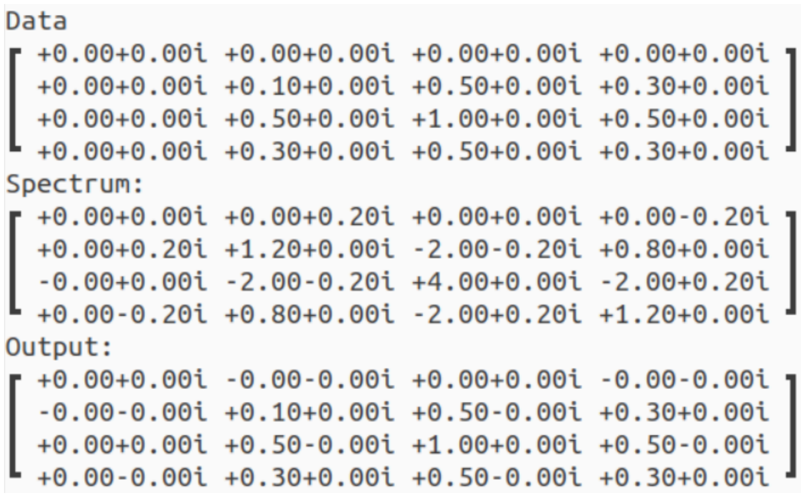
This ‘TestFour2floats.c’ sample program provided by the instructor serves to demonstrate how to perform a 2D FFT and its inverse using the FFT utility functions provided in Four2.h. The program uses a simple 4x4 data set for the demonstration. The ‘void printvector(float *data, const unsigned n)’ function prints a 2D array (vector) of complex numbers to the console. The complex numbers are stored in a single-dimensional array with interleaved real and imaginary parts. The function also adds special characters to make the printed array look like a matrix. In the main program, ‘int main(int argc, char *argv[])’, ‘n’ is set to 4, a 4x4 2D array, and three float arrays ‘data’, ‘spectrum’, and ‘output’ are dynamically allocated. These arrays will hold the original data, the result of the FFT, and the result of the inverse FFT, respectively. A small set of data points is populated into the ‘data’ array. The ‘spectrum’ array is populated with the contents of ‘data’ because the FFT function will overwrite its input with its output. The ‘fft_Four2’ function is called with ‘spectrum’ as the input and ‘false’ to indicate that it's a forward FFT. The ‘fft_Four2’ function is called again, this time with ‘output’ (copied from ‘spectrum’) as the input and ‘true’ to indicate that it's an inverse FFT. The ‘data’, ‘spectrum’, and ‘output’ are then printed out

The ‘Lab2_Part1.c’ program performs a 2D FFT on an image, normalizes the magnitude spectrum, and then performs an inverse FFT to get the image back in the spatial domain. The code is designed to read a grayscale image in PGM format. Opens a file called ‘cake.pgm’ and reads the image dimensions and pixel data into the ‘image’ array. The real part of the ‘spectrum’ array is populated with the image data, and the imaginary part is set to zero. The function ‘fft_Four2()’ is called to perform a 2D FFT on the spectrum array. The ‘spectrum’ array is copied to the ‘output’ array to prepare for the inverse FFT. The magnitude of each complex number in the spectrum is calculated and stored in ‘magnitude_spectrum’. The magnitude of the spectrum is then normalized by finding the ‘max’ and ‘min’ values and scaling the magnitude of the spectrum into the specified range of [0, 255] for an 8-bit grayscale image. The normalized spectrum is written to a new PGM file called ‘cake_spectrum.pgm’. The ’fft_Four2()’ function is called to perform an inverse FFT on the ‘output’ array from earlier. To finally write the real image (result of the inverse FFT) to a new PGM file called ‘cake_real.pgm’, the real components from the ‘output’ array need to be extracted.

Lastly, the ‘Lab2_Part2.c’ program performs a 2D FFT on an MRI image, applies a Butterworth Low Pass Filter to the Fourier spectrum, and then performs an inverse FFT to revert the image back to the spatial domain. The ‘create_2D_array’ function allocates a 2D array of floats of given rows and columns. It is used to store the Butterworth filter coefficients. The ‘create_butterworth_filter’ function generates a Butterworth low-pass filter in the frequency domain. It calculates the coefficients based on a given cutoff frequency, ‘D_0’. The function ‘apply_butterworth_filter’ multiplies the real and imaginary parts of the Fourier spectrum by the coefficients of the Butterworth filter. This effectively applies the low-pass filter to the Fourier-transformed image. The function ‘calculate_total_energy’ calculates the total energy in the frequency domain within a circular region of radius D_0 after filtering. The main function is similar to the Part1 program except those three helper functions are called in between the forward FFT on the spectrum and when the spectrum is copied to the output. This causes the output image after the IFFT to now have a blurring effect depending on the extent of the cutoff frequency.

III. Results

The output of ‘./TestFour2floats’ displays three 2D arrays representing different stages of 2D Fast Fourier Transform (FFT) and its inverse:

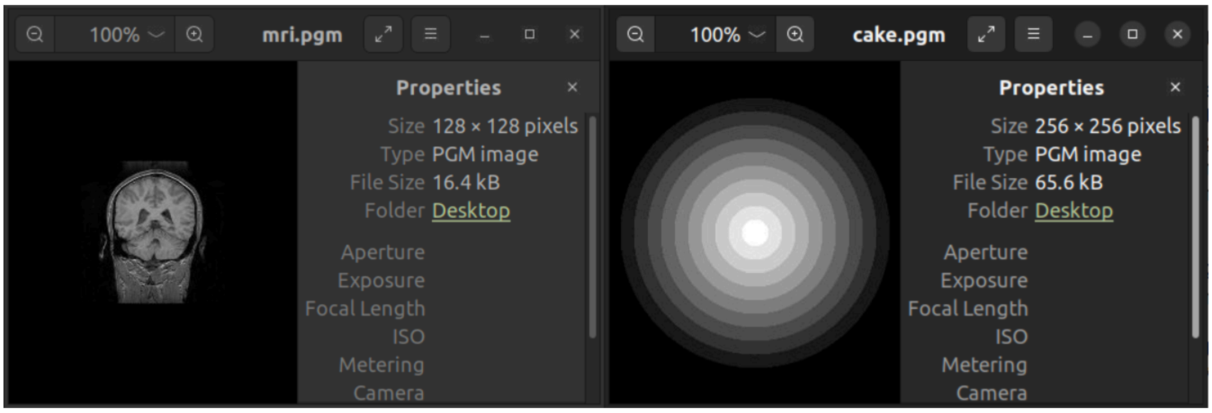


```
Data
┌ +0.00+0.00i +0.00+0.00i +0.00+0.00i +0.00+0.00i ┐
│ +0.00+0.00i +0.10+0.00i +0.50+0.00i +0.30+0.00i │
│ +0.00+0.00i +0.50+0.00i +1.00+0.00i +0.50+0.00i │
└ +0.00+0.00i +0.30+0.00i +0.50+0.00i +0.30+0.00i ┘
Spectrum:
┌ +0.00+0.00i +0.00+0.20i +0.00+0.00i +0.00-0.20i ┐
│ +0.00+0.20i +1.20+0.00i -2.00-0.20i +0.80+0.00i │
│ -0.00+0.00i -2.00-0.20i +4.00+0.00i -2.00+0.20i │
└ +0.00-0.20i +0.80+0.00i -2.00+0.20i +1.20+0.00i ┘
Output:
┌ +0.00+0.00i -0.00-0.00i +0.00+0.00i -0.00-0.00i ┐
│ -0.00-0.00i +0.10+0.00i +0.50-0.00i +0.30+0.00i │
│ +0.00+0.00i +0.50-0.00i +1.00+0.00i +0.50-0.00i │
└ +0.00-0.00i +0.30+0.00i +0.50-0.00i +0.30+0.00i ┘
```

Figure 1: The original 2D array in spatial/time domain, the frequency domain representation after 2D FFT, and the recovered 2D array after inverse FFT.

The ‘Data’ section is the original 2D array, which is essentially the input data. Each cell contains a complex number represented as ‘+x.xx+y.yyi’. In this case, the imaginary parts are all zero (+0.00i), and the real parts contain the values of the original 2D data array. These values are used for the 2D FFT operation. The ‘Spectrum’ section shows the 2D FFT of the original data. The values are again complex numbers. FFT transforms the data into the frequency domain, where the magnitude of each complex number represents the "strength" or "amplitude" of a particular frequency, and the angle (phase) of the complex number indicates the phase offset of that frequency component. The 'Output' section shows the result of the 2D inverse FFT on the spectrum. The imaginary parts are zero and the real parts are equal to tthe original data, which means the inverse FFT was successful in recovering the original data from its frequency domain representation.

For reference, the two images that are worked on in this assignment are shown below.



Properties

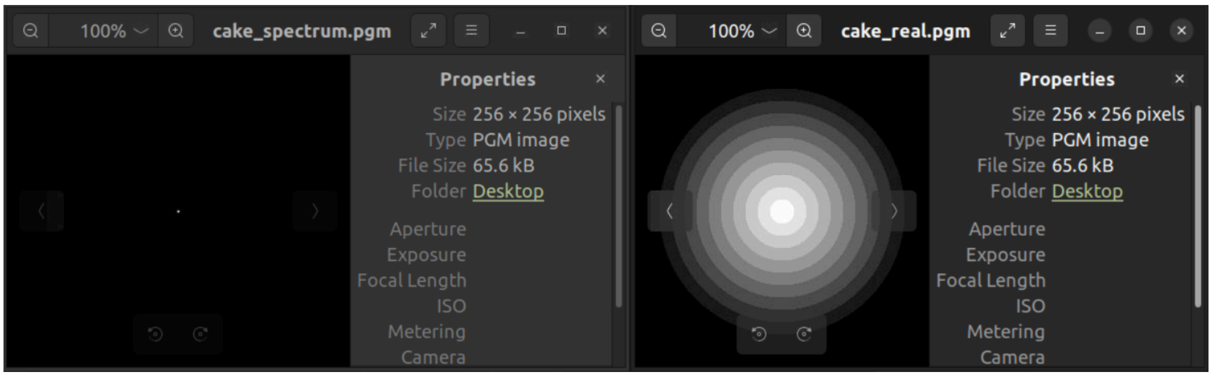
Size 128 × 128 pixels
Type PGM image
File Size 16.4 kB
Folder Desktop
Aperture
Exposure
Focal Length
ISO
Metering
Camera

Properties

Size 256 × 256 pixels
Type PGM image
File Size 65.6 kB
Folder Desktop
Aperture
Exposure
Focal Length
ISO
Metering
Camera

Figure 2: The MRI image is on the left with a height and width of 128 pixels. The cake image is on the right with a height and width of 256 pixels.

The first part of the assignment asked to read the provided synthesized cake image, obtain its 2-D Fourier Transform, and 2-D Fourier inverse the result to re-obtain the original image. The resulting normalized spectrum image and the re-obtained image are shown in the below figure.



Properties

Size 256 × 256 pixels
Type PGM image
File Size 65.6 kB
Folder Desktop
Aperture
Exposure
Focal Length
ISO
Metering
Camera

Properties

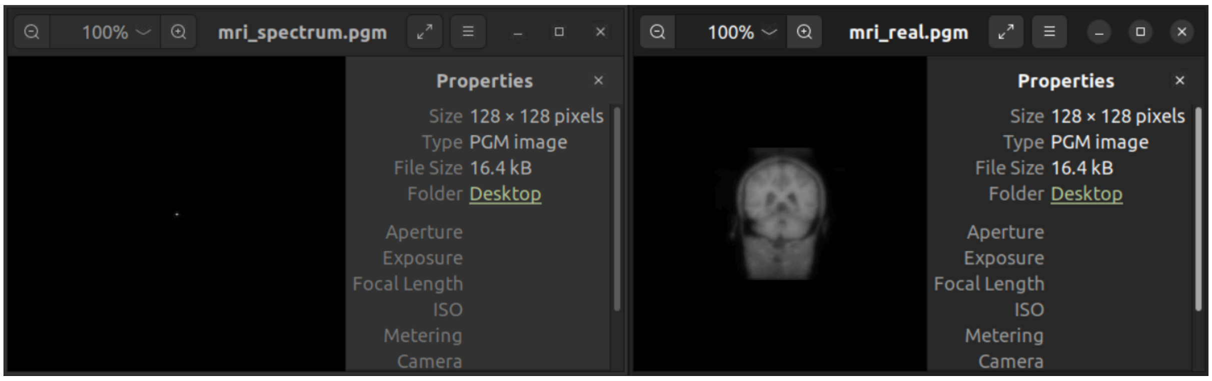
Size 256 × 256 pixels
Type PGM image
File Size 65.6 kB
Folder Desktop
Aperture
Exposure
Focal Length
ISO
Metering
Camera

Figure 3: The normalized Fourier spectrum of the cake on the left shows a single white dot in the center. The re-obtained image of the cake on the right after the IFFT shows a copy of the original.

The white dot in the middle of the 2D Fourier spectrum represents the DC component (zero frequency component) of the image. An FFT converts an image from its spatial domain to its frequency domain. The resulting 'spectrum' image visualizes the frequencies present in the original image. The low frequencies are located at the corners, and the high frequencies are in the center. After performing the FFT shift, the low frequencies move to the center. A white dot in the center of the normalized spectrum indicates a concentration of low-frequency components. The DC component represents the average brightness of the entire image. Images with smooth variations and less detail usually have strong low-frequency components which is demonstrated in this spectrum image.

The re-obtained cake image shows a copy of the original input image which is what one would expect. The image was read, applied to the real part of a spectrum, applied to a forward FFT, applied to a reverse FFT, and had the real part written to the output image shown. Nothing changed the input image since no filter was applied in the Fourier domain. The first part of this assignment was meant to understand and prove this process of loading an input image and applying an FFT in both directions for interpretability.

The second part of the assignment asked to repeat the steps of part one with the MRI image, except a Butterworth Low Pass Filter is applied in the Fourier domain at two different frequency cutoffs to see their effects. At a cutoff frequency of 10, the resulting normalized spectrum image and the output image in the spatial domain are shown in the below figure.



Properties

Size 128 × 128 pixels
Type PGM image
File Size 16.4 kB
Folder Desktop
Aperture
Exposure
Focal Length
ISO
Metering
Camera

Properties

Size 128 × 128 pixels
Type PGM image
File Size 16.4 kB
Folder Desktop
Aperture
Exposure
Focal Length
ISO
Metering
Camera

Figure 4: The normalized Fourier spectrum of the MRI on the left shows a single white dot in the center. The output image of the MRI on the right after the BWLP filter of cutoff frequency 10 and the IFFT shows a blurry copy of the original.

Applying a BWLP, in the frequency domain, primarily retains the low-frequency components while attenuating the high-frequency components. Low frequencies correspond to the larger, smoother structures in the image, while high frequencies correspond to the finer details and noise. After applying the BWLP (cutoff frequency of 10) to the clear input MRI image and performing an inverse Fourier transform, one obtains a smoothed, blurry version of the original MRI scan. The blurriness is because the filter has suppressed or eliminated the high-frequency components (fine details) of the image, leaving only the broader structures. Essentially, the filter averages out the intensity values over larger areas, leading to a loss of sharpness and detail. In medical image analysis, blurring an image would help in segmenting or identifying larger structures more easily while reducing noise. The normalized spectrum image again shows a white dot in the center. If zoomed in, it is actually slightly different than the spectrum image of the cake since the frequency components of the MRI are different.

Perhaps, the cutoff frequency is too low and should be raised in order to retain more of the high-frequency details. This move is shown in the figure below.

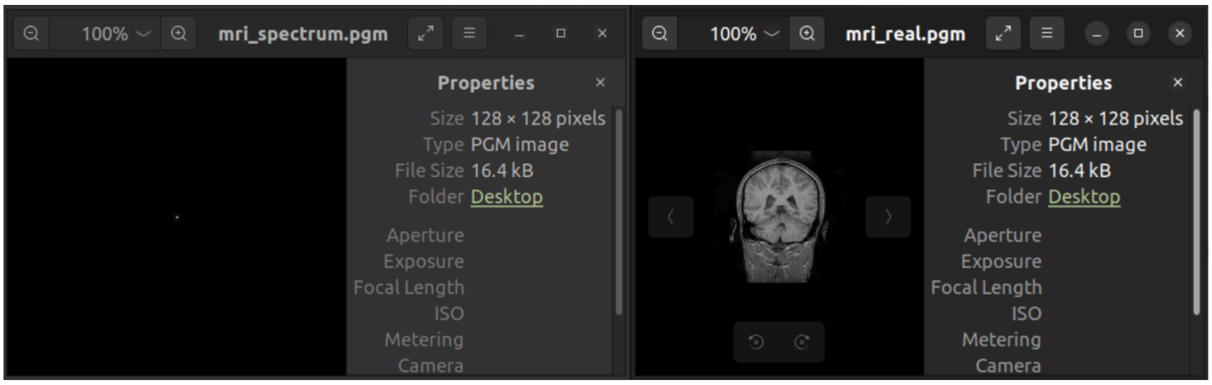


Figure 5: Again, the normalized Fourier spectrum of the MRI on the left shows a single white dot in the center. The output image of the MRI on the right after the BWLP filter of cutoff frequency 100 and the IFFT shows a blurry copy of the original.

At an increased cutoff frequency of 100, the output MRI image retains much more of its original detail than the cutoff frequency of 10. This is helpful in reducing some of the noise while still seeing the structure in detail.

The total energy within D_0=10 in the frequency domain after filtering was 5742457 and the total energy with D_0=100 was 33161514. This makes sense since as the cutoff frequency is decreased, more of the higher frequencies are cut off. This results in a lower total energy as shown with D_0=10.

IV. Observations/discussions

Most of my difficulty initially occurred with the working of the Makefile. I was not at all familiar with them before this project and had no idea how they worked. I then learned how they create dependables and executables. From this, I was able to edit the provided Makefile to allow two additional executables for the first part and second part of this assignment.

I spent the most time on the ‘Lab2_Part1’ program to understand the ‘Four.c’ source file it was calling along with how to read and write the cake image based on the code from the previous assignment. From this, a lot of time was spent debugging the program to set up the spectrum array before transformation, applying the forward FFT, calculating the spectrum magnitude, normalizing the spectrum, applying the IFFT, and finally writing the real part of the output to the output image. Overall, this program worked and carried out the desired expectations. The spectrum showed a white dot in the center representing the average intensity of the image and the re-obtained image was a copy to the input cake image.

Most of the time spent making the 'Lab2_Part2' program was involved in constructing the functions to create the Butterworth filter, applying it, and calculating the total energy of the real image after the IFFT. Again, the program worked and carried out the desired functions. The spectrum showed a white dot in the center representing the average intensity of the image and the output image was a blurred copy of the input MRI image with the high-frequency details attenuated.

V. Conclusions

This assignment provided an in-depth exploration of 2-D Fast Fourier Transforms (FFT) and its inverse (IFFT), concerning their application in image processing. Starting with a provided baseline code (TestFour2floats.c), the task extended to implement a Butterworth Low Pass Filter (BLPF) to produce a smoothed, low-pass-filtered output image. This helped in understanding the theory of Fourier domain processing and its practical applications. The assignment began with running 2-D FFT and IFFT operations on a synthesized cake image. This helped in gaining a foundational understanding of how Fourier domain transformations work and to then add on further processes. A critical step was to normalize the images to ensure they were integer-valued and within the 0-255 range. This was necessary for the accurate visualization of images. The crux of the assignment was implementing the Butterworth Low Pass Filter on an MRI image. This step involved crafting a 2-D filter, centering it in its spectrum, and applying it to both the real and imaginary parts of the FFT of the MRI image. Two different cutoff frequencies (D_0) for the Butterworth filter were experimented with. The total energy remaining after filtering was computed for both, demonstrating that a lower cutoff frequency decreases the total energy. The successful implementation and understanding of FFT, IFFT, and Butterworth Low Pass filtering techniques will aid in future assignments in this course and field.

Appendix

Lab2_Part1.c

```
//Code to run 2D FFT of an image
//Fall 2023
//Daniel Maienza

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "Four2.h"

int main(int argc, char *argv[]){
  unsigned sizeX; //image width
  unsigned sizeY; //image height
  unsigned char *image; //image array
  unsigned levels;
  unsigned i;
  float min_val; //max of spectrum magnitude
  float max_val; //min of spectrum magnitude

    /* Read Image */
  FILE *iFile = fopen("cake.pgm","r");
  if(iFile==0) return 1;
  if(3!=fscanf(iFile, "P5 %d %d %d ", &sizeX, &sizeY, &levels)) return 1;
  image=(unsigned char *) malloc(sizeX*sizeY);
  fread(image,sizeof(unsigned char),sizeX*sizeY,iFile);
  fclose(iFile);

  unsigned n=sizeX; //size of 2D image in both directions

  /* 3 arrays with the following meaning */
  /* image --> FFT --> spectrum --> IFFT --> output */
  float *spectrum, *output, *magnitude_spectrum;

  spectrum=(float*)malloc(sizeof(float)*2*n*n);
//NOTE: DO NOT CONFUSE Spectrum which is just a variable name we give here for the
complex Fourrier transform, with  the image spectrum as we define it in class
  output=(float*)malloc(sizeof(float)*2*n*n);
  magnitude_spectrum = (float*)malloc(sizeof(float)*n*n);

  if(spectrum==0 || output==0) return 1;

  /* since fourl destroys the input array with the output */
  /* use the output of the FFT as the input value */
  for(i=0; i<n*n; i++){
    spectrum[2*i]=image[i];
    spectrum[2*i+1]=0;
  }

  //call 2D fft
  fft_Four2(spectrum,n,n,false);

  /*copy spectrum into output and call inverse FFT on output later */
  for(i=0; i<2*n*n; i++){
    output[i]=spectrum[i];
  }

  //Find magnitude of spectrum from real and imaginary part
  for(i=0; i<n*n; i++){

magnitude_spectrum[i]=sqrt(spectrum[2*i]*spectrum[2*i]+spectrum[2*i+1]*spectrum[2*i+1]);
  }

  //Find min and max
  min_val = magnitude_spectrum[0];
  max_val = magnitude_spectrum[0];

  for(i=0; i<n*n; i++) {
    if (magnitude_spectrum[i] < min_val) {
      min_val = magnitude_spectrum[i];
      }
    if (magnitude_spectrum[i] > max_val) {
      max_val = magnitude_spectrum[i];
      }
  }

  //Normalize the spectrum
  unsigned char *normalized_image;
  normalized_image = (unsigned char*) malloc(n*n*sizeof(unsigned char));

  for(i=0; i<n*n; i++) {
    normalized_image[i] = (255 * (magnitude_spectrum[i]-min_val) / (max_val-min_val));
  }

  /*write spectrum image to file*/
  iFile = fopen("cake_spectrum.pgm","w");
  if(iFile==0) return 1; //error handling
  fprintf(iFile, "P5 %d %d %d ",sizeX,sizeY,255);//write header
  fwrite(normalized_image, sizeof(unsigned char), sizeX*sizeY, iFile); //write binary image
  fclose(iFile);

  //call the 2D inverse FFT on output
  fft_Four2(output,n,n,true);
  // Extract real components for display
  unsigned char *real_image;
  real_image = (unsigned char*) malloc(n*n*sizeof(unsigned char));
  for(i=0; i<n*n; i++){
    real_image[i] = (unsigned char) (output[2*i]);
  }

  /*write real image to file*/
  iFile = fopen("cake_real.pgm","w");
  if(iFile==0) return 1; //error handling
  fprintf(iFile, "P5 %d %d %d ",sizeX,sizeY,255);//write header
  fwrite(real_image, sizeof(unsigned char), sizeX*sizeY, iFile); //write binary image
  fclose(iFile);
  return 0;
}
```

Lab2_Part2.c

```
//Code to run 2D FFT and then Butterworth Low Pass Filter of an image
//Fall 2023
//Daniel Maienza

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "Four2.h"

// Function to allocate a 2D array of floats
float **create_2D_array(int rows, int cols) {
  float **array = (float **)malloc(rows * sizeof(float *));
  if(array == NULL) {
      // Handle memory allocation failure
      exit(1);
  }
  for(int i = 0; i < rows; ++i) {
      array[i] = (float *)malloc(cols * sizeof(float));
      if(array[i] == NULL) {
          // Handle memory allocation failure
          exit(1);
      }
  }
  return array;
}

// Function to create a Butterworth filter
void create_butterworth_filter(float **filter, int n, float D0) {
    int center_x = n / 2;
    int center_y = n / 2;
    for (int x = 0; x < n; ++x) {
        for (int y = 0; y < n; ++y) {
            float distance = sqrt((x - center_x) * (x - center_x) + (y - center_y) * (y - center_y));
            filter[x][y] = 1.0 / (1.0 + pow(distance / D0, 2));
        }
    }
}

// Function to apply the Butterworth filter
void apply_butterworth_filter(float *spectrum, float **filter, int n) {
    for (int x = 0; x < n; ++x) {
        for (int y = 0; y < n; ++y) {
            int index = x * n + y;
            // Multiply real and imaginary parts by filter
            spectrum[2 * index] *= filter[x][y];
            spectrum[2 * index + 1] *= filter[x][y];
        }
    }
}

// Function to calculate the total energy in the frequency domain within a circle of radius D0
float calculate_total_energy(float *magnitude_spectrum, int n, float D0) {
    float energy = 0.0;
    int center_x = n / 2;
    int center_y = n / 2;

    for (int x = 0; x < n; ++x) {
        for (int y = 0; y < n; ++y) {
            float distance = sqrt((x - center_x) * (x - center_x) + (y - center_y) * (y - center_y));
            if (distance <= D0) {
                int index = x * n + y;
                energy += magnitude_spectrum[index];
            }
        }
    }
    return energy;
}



int main(int argc, char *argv[]){
  unsigned sizeX; //image width
  unsigned sizeY; //image height
  unsigned char *image; //image array
  unsigned levels;
  unsigned i;
  float min_val; //max of spectrum magnitude
  float max_val; //min of spectrum magnitude
  float D0 = 100; //cutoff frequency

    /* Read Image */
  FILE *iFile = fopen("mri.pgm","r");
  if(iFile==0) return 1;
  if(3!=fscanf(iFile, "P5 %d %d %d ", &sizeX, &sizeY, &levels)) return 1;
  image=(unsigned char *) malloc(sizeX*sizeY);
  fread(image,sizeof(unsigned char),sizeX*sizeY,iFile);
  fclose(iFile);

  unsigned n=sizeX; //size of 2D image in both directions

  /* 3 arrays with the following meaning */
  /* image --> FFT --> spectrum --> IFFT --> output */
  float *spectrum, *output, *magnitude_spectrum;

  spectrum=(float*)malloc(sizeof(float)*2*n*n);
//NOTE: DO NOT CONFUSE Spectrum which is just a variable name we give here for the
complex Fourrier transform, with  the image spectrum as we define it in class
  output=(float*)malloc(sizeof(float)*2*n*n);
  magnitude_spectrum = (float*)malloc(sizeof(float)*n*n);

  if(spectrum==0 || output==0) return 1;

  /* since fourl destroys the input array with the output */
  /* use the output of the FFT as the input value */
  for(i=0; i<n*n; i++){
    spectrum[2*i]=image[i];
    spectrum[2*i+1]=0;
  }

  //call 2D fft
  fft_Four2(spectrum,n,n,false);

  // Create a 2D array for the filter
  float **filter = create_2D_array(n, n);

  // Create the Butterworth filter
  create_butterworth_filter(filter, n, D0);

  // Apply the Butterworth filter to the spectrum
  apply_butterworth_filter(spectrum, filter, n);

  /*copy spectrum into output and call inverse FFT on output later */
  for(i=0; i<2*n*n; i++){
    output[i]=spectrum[i];
  }
```

```
 //Find magnitude of spectrum from real and imaginary part
 for(i=0; i<n*n; i++){

magnitude_spectrum[i]=sqrt(spectrum[2*i]*spectrum[2*i]+spectrum[2*i+1]*spectrum[2*i+1]);
 }

 //Calculate the total energy remaining after filtering with the cutoff
 float total_energy = calculate_total_energy(magnitude_spectrum, n, D0);
 printf("Total energy with D0=%.0f: %.0f\n", D0, total_energy);

 //Find min and max
 min_val = magnitude_spectrum[0];
 max_val = magnitude_spectrum[0];

 for(i=0; i<n*n; i++) {
   if (magnitude_spectrum[i] < min_val) {
     min_val = magnitude_spectrum[i];
     }
   if (magnitude_spectrum[i] > max_val) {
     max_val = magnitude_spectrum[i];
     }
 }

 //Normalize the spectrum
 unsigned char *normalized_image;
 normalized_image = (unsigned char*) malloc(n*n*sizeof(unsigned char));

 for(i=0; i<n*n; i++) {
   normalized_image[i] = (255 * (magnitude_spectrum[i]-min_val) / (max_val-min_val));
 }

 /*write spectrum image to file*/
 iFile = fopen("mri_spectrum.pgm","w");
 if(iFile==0) return 1; //error handling
 fprintf(iFile, "P5 %d %d %d ",sizeX,sizeY,255);//write header
 fwrite(normalized_image, sizeof(unsigned char), sizeX*sizeY, iFile); //write binary image
 fclose(iFile);

 //call the 2D inverse FFT on output
 fft_Four2(output,n,n,true);

 // Extract real components
 unsigned char *real_image;
 real_image = (unsigned char*) malloc(n*n*sizeof(unsigned char));
 for(i=0; i<n*n; i++){
   real_image[i] = (unsigned char) (output[2*i]);
 }
```

```
  /*write real image to file*/
  iFile = fopen("mri_real.pgm","w");
  if(iFile==0) return 1; //error handling
  fprintf(iFile, "P5 %d %d %d ",sizeX,sizeY,255);//write header
  fwrite(real_image, sizeof(unsigned char), sizeX*sizeY, iFile); //write binary image
  fclose(iFile);
  return 0;
}
```