Elie Daou

Assignment #2

Due Monday 6, March 2017

The objective of this assignment was multiple parts. The first part was to properly typecast and otherwise manipulate data using C. The next, was to familiarize ourselves with how to use a Fourier transform function (FFT) to manipulate data in an array. The last part was to create and apply a Butterworth Low Pass Filter.

The background of the assignment is that a Fast Fourier Transform (FFT) can be used to get the spectrum of an image. This can be done in a 2-dimensional array of pixel values. Also, a Butterworth Low Pass Filter can be used to filter an image.

The algorithms used for this lab were implementation of the FFT. A 1-dimensional Fast Fourier Transform function was used to implement a 2-dimenstional transform. First the 1D transform was used on the rows of the image. Next, the 1D transform was used on the columns of the image. The resulting values were then normalized to be within 0 and 255. This normalized array was then centered, so the spectrum was in the center and not the four corners. These values were then typecasted to be unsigned chars and placed back into an output image to view. The next algorithm used was the Butterworth Low Pass Filter. This is an equation that is used on the spectral array. The “buttered” array is then inverse-FFT and placed back into an output image to view.

Results

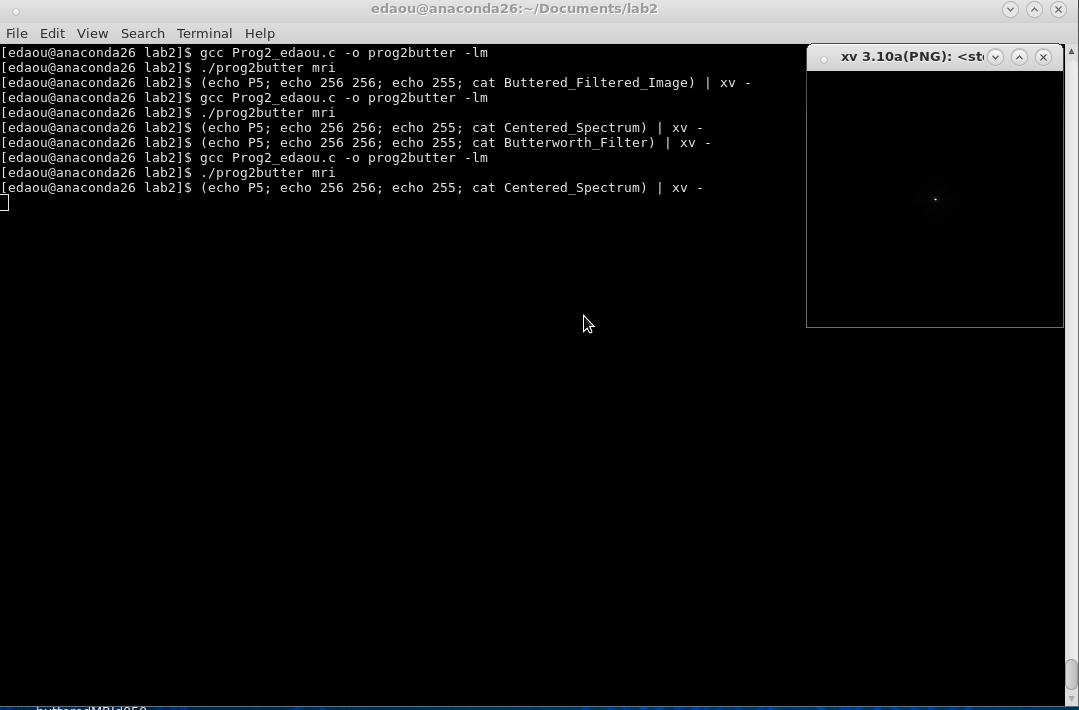
Figure 1 (MRI Spectrum)

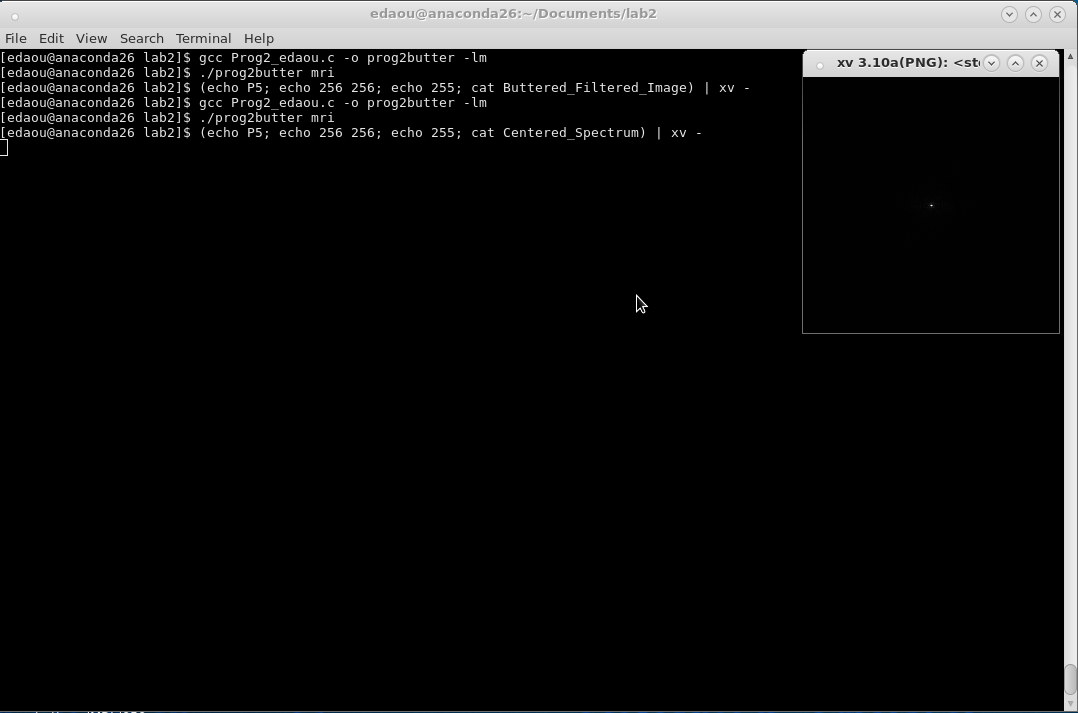
Figure 2 (MRI Spectrum)

Figure 1 and figure 2 seem identical, but in fact they are the outputs of the program running at two different times. Figure 1 is run when D0 (for the Butterworth Filter) is set to 10, and figure 2 is when D0 is set to 50. They look identical because they are. The Butterworth Filter does not affect the spectral analysis at all.

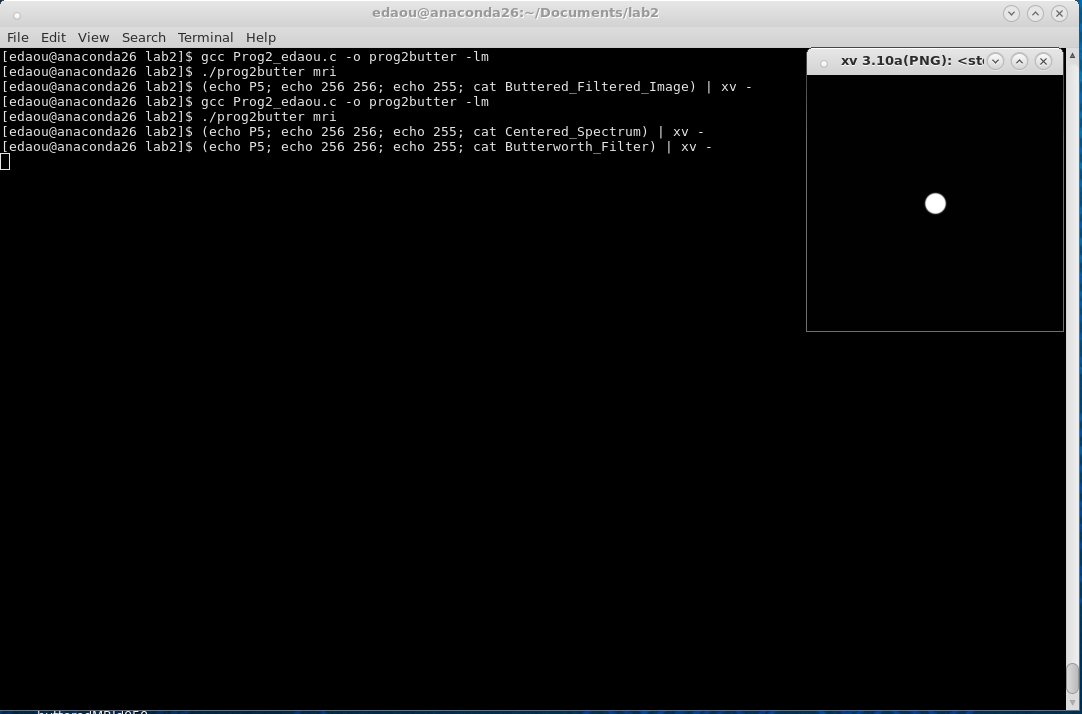
Figure 3 (BLPF D0=10)

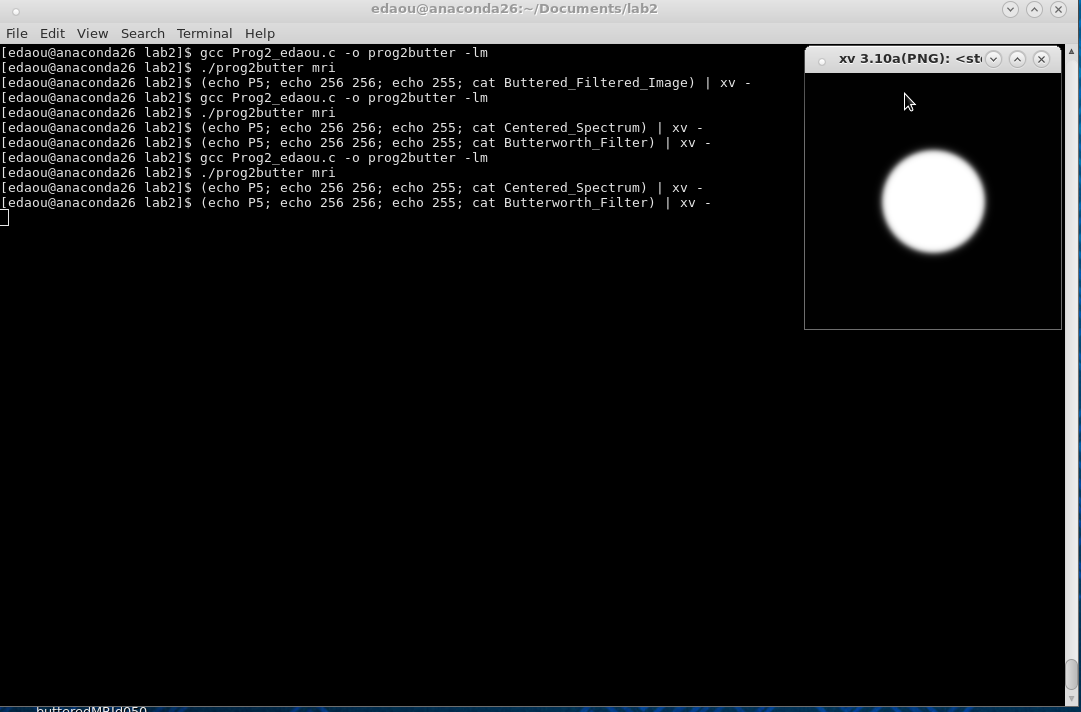
Figure 4 (BLPF D0=50)

Figure 5 (Image after BLPF D0=10)

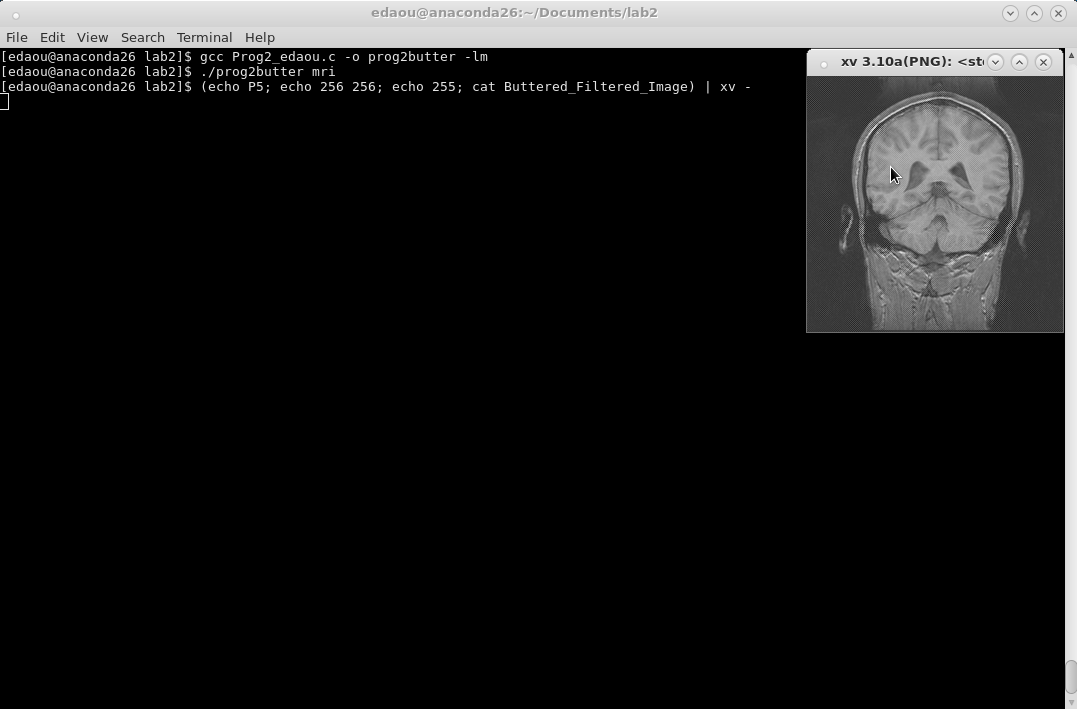


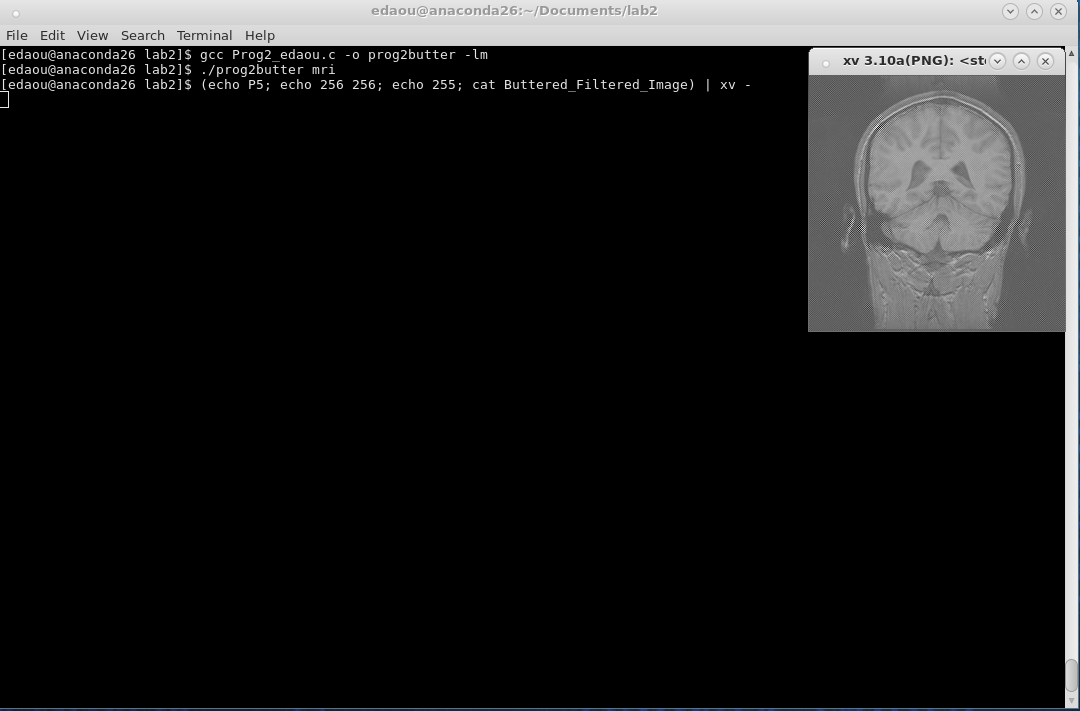
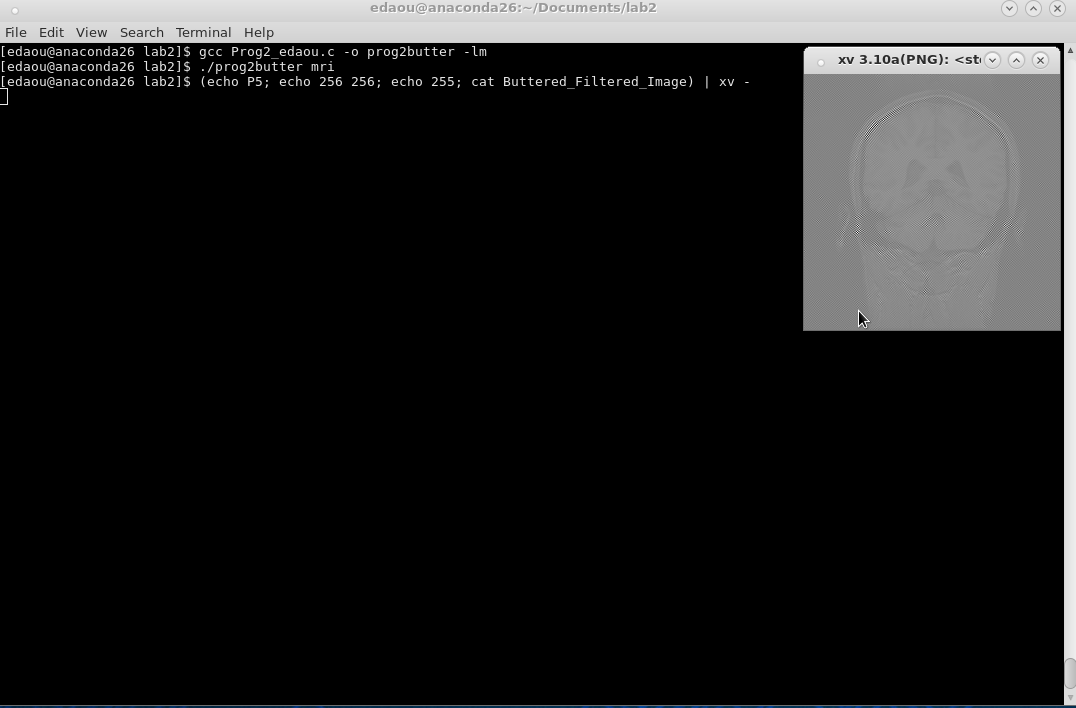
Figure 6 (Image after BLPF D0=20)

Figure 7 (Image after BLPF D0=50)

**

The only real observations I have from this lab is that it is very easy to mess up the filters and the spectral analysis. I produced more outputs that were simply just “noise” than I would care to admit. I spent many hours banging my head against the wall trying to make this work. The order in which you apply certain filters and normalizations is extremely important.

In conclusion, this lab was very hard. However, after many hours of work, I am very excited about the implications of these filters and how they can be applied to other image processing projects.

**Source Code:**

/\*

\* Program 2 - Elie Daou

\* Due Monday 6 March, 2017 9:00am

\* This program reads an input file and outputs 4 files

\* 1. The spectrum of that file

\* 2. The original output file, after it has gone through FFT2d/ FFT2d^-1

\* 3. The butterworth filter

\* 4. The original image after it has had the butterworth filter applied to it.

\*/

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

int const n = 256;

// This is the 1D FFT given to us. It was not modified at all.

four1(data,nn,isign)

float data[];

int nn,isign;

{

int 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=2\*mmax;

theta=6.28318530717959/(isign\*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;

}

}

// get the minimum and maximum of the array to be able to normalize it

float getMaximumOfArray(float \*\*inputtedArray);

float getMinimumOfArray(float \*\*inputtedArray);

//find the magnitude of the FFT (real and imaginary combined)

void findMagnitude(float \*\*inputArray, float \*\*outputArray);

// do fourier transform in 2D

void four2d(float \*\*inArray);

// do inverse fourier in 2D

void four2dInverse(float \*\*inArray);

// normalize the data based on 255 being the max possible value

void normalize(float \*\*inputArray, float \*\*outputArray, float min, float max);

// function to output the files

void outputFile(char nameOfFile[256], float \*\*dataArray);

// main function

void main(int argc, char \*argv[]){

int row, column;

//dynamic memory allocation is sooooo0o0o00o0o much better, easier to pass into funtions and clean to work with. be careful with segmentation faults!!!

float \*\*data, \*\*spectrum, \*\*newSpectrum, \*\*realValuesOnly, \*\*butter, \*\*newButter, \*\*butteredSpectrum, \*\*centeredSpectrum;

char outputFileName[256];

unsigned char pixel; // represent one pixel at a time, to be able to typecast

FILE \*input;

int D0 = 50;

int smoothness = 30; //"smoothness factor" or "gradient"

float minimum; // minimum pixel value to normalize

float maximum; // maximum pixel value to normalize

// 2-d array to hold original image data

data =(float\*\*) calloc (n, sizeof (float \*));

for (row = 0; row < n; row++) {

data[row] = (float \*) calloc (n, sizeof (float));

}

// spectrum array to hold fourier transform image data

spectrum =(float\*\*) calloc (n, sizeof (float \*));

for (row = 0; row < n; row++) {

spectrum[row] = (float \*) calloc (2\*n, sizeof (float));

}

// new spectrum to hold the combined spectrum data

newSpectrum =(float\*\*) calloc (n, sizeof (float \*));

for (row = 0; row < n; row++) {

newSpectrum[row] = (float \*) calloc (n, sizeof (float));

}

// Store and output the centered spectrum

centeredSpectrum =(float\*\*) calloc (n, sizeof (float \*));

for (row = 0; row < n; row++) {

centeredSpectrum[row] = (float \*) calloc (n, sizeof (float));

}

// holds real data to output from the FFT

realValuesOnly =(float\*\*) calloc (n, sizeof (float \*));

for (row = 0; row < n; row++) {

realValuesOnly[row] = (float \*) calloc (n, sizeof (float));

}

// the butterworth LPF

butter =(float\*\*) calloc (n, sizeof (float \*));

for (row = 0; row < n; row++) {

butter[row] = (float \*) calloc (n, sizeof (float));

}

// normalized butterworth filter to be outputted

newButter =(float\*\*) calloc (n, sizeof (float \*));

for (row = 0; row < n; row++) {

newButter[row] = (float \*) calloc (n, sizeof (float));

}

//take the spectrum through the LPF

butteredSpectrum =(float\*\*) calloc (n, sizeof (float \*));

for (row = 0; row < n; row++) {

butteredSpectrum[row] = (float \*) calloc (2\*n, sizeof (float));

}

// Read the inputted image to data, same as lab 1

input = fopen(argv[1], "rb");

for (row = 0; row < n; row++) {

for (column = 0; column < n; column++) {

fread(&pixel, sizeof(char), 1, input);

data[row][column]= (float)pixel;

}

}

// close input image since it's not needed anymore

fclose(input);

// store data into spectrum because you can't use FFT without losing the info passed into it

for (row = 0; row < n; row++) {

for (column = 0; column < n; column++) {

spectrum[row][column\*2] = data[row][column];

}

}

four2d(spectrum);

// find the magnitude of the fourier transform

findMagnitude(spectrum, newSpectrum);

// This way of centering the information didn't work, but i kept it in because i did try it and i can't figure out why it won't work

/\*for (row=0; row<n; row++) {

for (column=0; column<n; column++) {

centeredSpectrum[row][column] = newSpectrum[row][column] \* pow((-1), (row+column));

}

}

maximum = getMaximumOfArray(centeredSpectrum);

minimum = getMinimumOfArray(centeredSpectrum);

// normalize the data based on the max and min values with the new max being 255

normalize(centeredSpectrum, centeredSpectrum, minimum, maximum);\*/

// find the max and min values pre-normalization

maximum = getMaximumOfArray(newSpectrum);

minimum = getMinimumOfArray(newSpectrum);

// normalize the data based on the max and min values with the new max being 255

normalize(newSpectrum, newSpectrum, minimum, maximum);

// center the normalized data.

for (row=0; row<n/2; row++) {

for (column=0; column<n/2; column++) {

float temp = newSpectrum[row][column];

centeredSpectrum[row][column] = newSpectrum[row+n/2][column+n/2];

centeredSpectrum[row+n/2][column+n/2] = temp;

temp = newSpectrum[row + n/2][column];

centeredSpectrum[row + n/2][column] = newSpectrum[row][column+n/2];

centeredSpectrum[row][column+n/2] = temp;

}

}

// save the centered spectrum

sprintf(outputFileName, "Centered\_Spectrum");

outputFile(outputFileName, centeredSpectrum);

// make the butterworth filter, just the BLPF equation

for (row=0; row<n; row++) {

for (column=0; column<n; column++) {

float center = sqrt(((column-128)\*(column-128)) + ((row-128)\*(row-128)));

butter[row][column] = 1/(1+(sqrt(2)-1)\*pow((center/D0),smoothness));

}

}

// repeat steps for min/ max/ normalize for the butterworth filter

maximum = getMaximumOfArray(butter);

minimum = getMinimumOfArray(butter);

normalize(butter, newButter, minimum, maximum);

// output the butterworth filter

sprintf(outputFileName, "Butterworth\_Filter");

outputFile(outputFileName, newButter);

// apply the butterworth filter to the spectrum

for (row=0; row<n; row++) {

for (column=0; column<n; column++) {

butteredSpectrum[row][column\*2] = spectrum[row][column\*2] \* newButter[row][column];

butteredSpectrum[row][column\*2+1] = spectrum[row][(column\*2)+1] \* newButter[row][column];

}

}

// reverse the FFT2D for both the buttered and unbuttered spectrum

four2dInverse(spectrum);

four2dInverse(butteredSpectrum);

// ignore all the imaginary numbers and only save the real values

for (row=0; row<n; row++) {

for (column=0; column<2\*n; column++) {

if (column%2 == 0) {

realValuesOnly[row][column/2] = spectrum[row][column];

}

}

}

// get the max and min to be able to normalize (we do this every time)

maximum = getMaximumOfArray(realValuesOnly);

minimum = getMinimumOfArray(realValuesOnly);

// normalize so you can......

normalize(realValuesOnly, realValuesOnly, minimum, maximum);

// ...... properly output the data. told you it happens a lot. that's why there's a function for it.

sprintf(outputFileName, "Inverted\_Original\_Image");

outputFile(outputFileName, realValuesOnly);

// do that again for the buttered spectrum

for (row=0; row<n; row++) {

for (column=0; column<2\*n; column++) {

if (column%2 == 0) {

realValuesOnly[row][column/2] = butteredSpectrum[row][column];

}

}

}

// and again. min/ max/ normalize/ output

maximum = getMaximumOfArray(realValuesOnly);

minimum = getMinimumOfArray(realValuesOnly);

normalize(realValuesOnly, realValuesOnly, minimum, maximum);

sprintf(outputFileName, "Buttered\_Filtered\_Image");

outputFile(outputFileName, realValuesOnly);

// free up all dynamically allocated memory

for (row=0; row< n; row++) {

free(data[row]);

free(spectrum[row]);

free(newSpectrum[row]);

free(realValuesOnly[row]);

free(butter[row]);

free(newButter[row]);

free(butteredSpectrum[row]);

free(centeredSpectrum[row]);

}

free(data);

free(spectrum);

free(newSpectrum);

free(realValuesOnly);

free(butter);

free(newButter);

free(butteredSpectrum);

free(centeredSpectrum);

}

// get the maximum of an array, self explanatory

float getMaximumOfArray(float \*\*inputtedArray){

int row, column;

int n = 256;

float max;

max = inputtedArray[0][0];

for (row=0; row<n; row++) {

for (column=0; column<n; column++) {

if (inputtedArray[row][column] > max) {

max = inputtedArray[row][column];

}

}

}

return max;

}

// same with the minimum, self- explanatory

float getMinimumOfArray(float \*\*inputtedArray){

int row, column;

int n = 256;

float min;

min = inputtedArray[0][0];

for (row=0; row<n; row++) {

for (column=0; column<n; column++) {

if (inputtedArray[row][column] < min) {

min = inputtedArray[row][column];

}

}

}

return min;

}

// to get the maginitude it's the sqrt of the real^2 times the imaginary^2

void findMagnitude(float \*\*inputArray, float \*\*outputArray){

int n = 256;

int row, column;

for ( row=0; row<n; row++) {

for (column=0; column<n; column++) {

outputArray[row][column] = sqrt(pow(inputArray[row][2\*column], 2) + pow(inputArray[row][(2\*column)+1], 2));

}

}

}

// find the min and the max, then scale it up/ down into the range of 0-255, because those are our pixel ranges

void normalize(float \*\*inputArray, float \*\*outputArray, float min, float max){

int n = 256;

int row, column;

for ( row=0; row<n; row++) {

for (column=0; column<n; column++) {

outputArray[row][column] = 255 \* (inputArray[row][column] - min)/(max - min);

outputArray[row][column] = ceil(outputArray[row][column]);

}

}

}

// we do the FFT2D twice, i think, so it's only logical to make it's own function

void four2d(float \*\*inArray){

int row, column;

int n = 256;

float \*temp;

temp =(float\*) calloc (2\*n, sizeof (float ));

// for each row, pass it into the FFT

for (row = 0; row < n; row ++) {

for (column = 0; column <2\*n; column++) {

temp[column] = inArray[row][column];

}

four1(temp-1, n, 1);

for (column=0; column<2\*n; column++) {

inArray[row][column] = temp[column];

}

}

// now column by column because that's the rule, hombre

for (column = 0; column<n; column++) {

for (row=0; row<n; row++) {

temp[row\*2] = inArray[row][column\*2];

temp[(row\*2)+1] = inArray[row][(column\*2)+1];

}

four1(temp-1, n, 1);

for (row=0; row<n; row++) {

inArray[row][column\*2] = temp[row\*2];

inArray[row][(column\*2)+1] = temp[(row\*2)+1];

}

}

free(temp);

}

void four2dInverse(float \*\*inArray){

int row, column;

int n = 256;

float \*temp;

temp =(float\*) calloc (2\*n, sizeof (float ));

// time for inverse fourier transform

//start with columns, since you started fourier with rows

for (column = 0; column<n; column++) {

for (row=0; row<n; row++) {

temp[row\*2] = inArray[row][column\*2];

temp[(row\*2)+1] = inArray[row][(column\*2)+1];

}

four1(temp-1, n, -1);

for (row=0; row<n; row++) {

inArray[row][column\*2] = temp[row\*2];

inArray[row][(column\*2)+1] = temp[(row\*2)+1];

}

}

// next do the rows chica, because that's what you did first and this rule isn't made to be broken

for (row=0; row<n; row++) {

for (column=0; column<n; column++) {

temp[(column\*2)] = inArray[row][(column\*2)];

temp[(column\*2)+1] = inArray[row][(column\*2)+1];

}

four1(temp-1, n, -1);

for (column=0; column<2\*n; column++) {

inArray[row][column] = temp[column];

}

}

free(temp);

}

// the most boring funtion here, all it does it outputs a file of bits..... yay

void outputFile(char nameOfFile[256], float \*\*dataArray){

unsigned char pixel;

FILE \*outputFile;

int column, row;

int n = 256;

outputFile = fopen(nameOfFile, "wb");

for (row=0; row<n; row++) {

for (column=0; column<n; column++) {

pixel = (unsigned char)(dataArray[row][column]); // this is the typecasting everyone was worried about

fwrite(&pixel, sizeof(char), 1, outputFile);

}

}

fclose(outputFile);

}