# Lab 2 Report

**Author:** *Pablo Ruiz*

**Due Date:** *11/09/2022 9:00 AM*

## Objective:

The objective of this lab was to apply the Fast Fourier Transform (FFT) algorithm to a gray-scale input image in order to obtain its frequency domain spectrum. This was then used to create and apply a low pass Butterworth filter to the spectrum. Next, the inverse FFT algorithm was used to convert the filtered spectrum back to the spatial domain. Finally, the new filtered/smoothed image was saved as a new file and visualized.

# Algorithms / Functions:

- Read_PGM:

```
/// @brief Reads an image from a file
/// @param image output array of the image intensities
/// @param fileName full name of the image to read, ie. "image1.pgm"
/// @param pgmType output representing type of pgm encoding used P1-P6
/// @param sizeX output x dimension of the image intensity array
/// @param sizeY output y dimension of the image intensity array
/// @return If succesful return 0, otherwise return error code
unsigned Read_PGM (const char *fileName,
                   unsigned char *&image,
                   char *&pgmType,
                   unsigned &sizeX,
                   unsigned &sizeY,
                   unsigned normal=255)
```

- Write_PGM:

```
/// @brief Creates a .pgm image file with the passed name and writes
///        the passed image data to it
/// @param fileName full name of the image to write, ie. "image1.pgm"
/// @param image image data
/// @param pgmType type of pgm encoding used P1-P6
/// @param sizeX x dimension of the image data passed
/// @param sizeY y dimension of the image data passed
/// @param level max level range of the image data passed
/// @return If successful return 0, otherwise return error code
unsigned Write_PGM (const char *fileName,
                    unsigned char *image,
                    char *pgmType,
                    unsigned sizeX,
                    unsigned sizeY,
                    unsigned level=255)
```

- Unsigned_2_Cplx:

```
/// @brief Converts an array of unsigned values to an array of clpx objects
/// @param data array of unsigned values
/// @param size size of the array
/// @return the clpx array
cplx* Unsigned_2_Cplx(unsigned char *data, unsigned size)
```

- Cplx_2_Unsigned

```
/// @brief Converts an array of cplx objects to an array of unsigned values
/// @param data array of cplx objects
/// @param size size of the array
/// @return the unsigned array
unsigned char *Cplx_2_Unsigned(cplx *data, unsigned size)
```

- Float_2_Unsigned

```
/// @brief Converts an array of floats to an array of unsigned chars
/// @param data Input array of floats
/// @param size Size of the input array
/// @return A pointer to the output array of unsigned chars
unsigned char *Float_2_Unsigned(float *data, unsigned size)
```

- Copy_Cplx:

```
/// @brief Make a copy of a cplx array
/// @param source cplx array to copy
/// @param size size of the array
/// @return pointer to the new array
cplx *Copy_Cplx(cplx *source, unsigned size)
```

- Print_2D_Array:

```
/// @brief Print a 1D array as a 2D array of floats
/// @param data 1D input array of floats of size sizeX*sizeY
/// @param sizeX X dimension of the 2D array
/// @param sizeY Y dimension of the 2D array
void Print_2D_Array(float *data, unsigned sizeX, unsigned sizeY)
```

```
/// @brief Print a 1D cplx array as a 2D array of complex numbers
/// @param data 1D input array of cplx of size sizeX*sizeY
/// @param sizeX X dimension of the 2D array
/// @param sizeY Y dimension of the 2D array
void Print_2D_Array(cplx *data, unsigned sizeX, unsigned sizeY)
```

- Norm:

```
/// @brief normalize an array of floats with a known max value (upper limit)
/// @param data input array of floats, they will be normalized in place
/// @param size size of the input array
/// @param max current max value of the data in the array
/// @param norm upper limit of the output array, defaults to 255
void Norm(float *data, float max, unsigned size, float norm = 255)
```

```
/// @brief normalize an array of floats
/// @param data input array of floats, they will be normalized in place
/// @param size size of the input array
/// @param norm upper limit of the output array, defaults to 255
void Norm(float *data, unsigned size, float norm = 255)
```

- Mag:

```
/// @brief calculate the magnitude of a cplx array and return
///        its corresponding array of magnitudes
/// @param data input cplx array
/// @param size size of the input array
/// @return the magnitude array
float *Mag(cplx *data, unsigned size)
```

- Total_Power:

```
/// @brief Calculate the total power (sum) of a 1-D array
/// @param data Input 1-D array of floats
/// @param size Size of input array
/// @return The total power (sum)
float Total_Power(float *data, unsigned size)
```

- BWF:

```
/// @brief Apply a butterworth lowpass filter to an input array and
///        generates a new filtered output array
/// @param n degree for the filter
/// @param Do filter radius
/// @param B Output float array which contains the butterworth filter
/// @param data array to be filtered
/// @param N size of one vertice of the array (ie. N=64 for a 64x64 filter)
/// @return filtered array
cplx* BWF(unsigned n, unsigned Do, float *&B, cplx *data, unsigned N)
```

- fft_Four2:

```
/// @brief Compute the 2d FFT or IFFT of an image
/// @param im input 1-D image array to be transformed in place
/// @param sX input image x-axis dimension
/// @param sY input image y-axis dimension
/// @param inv if true compute the IFFT otherwise compute the FFT
/// @return spectrum of the image
void fft_Four2(float *im, //image
               unsigned sX, //width
               unsigned sY, //height
               bool inv){ //inverse fft when true
```

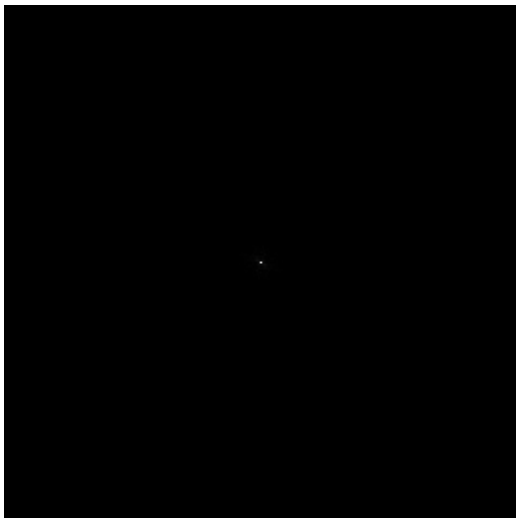
# Result:

- **Total power before filter =**

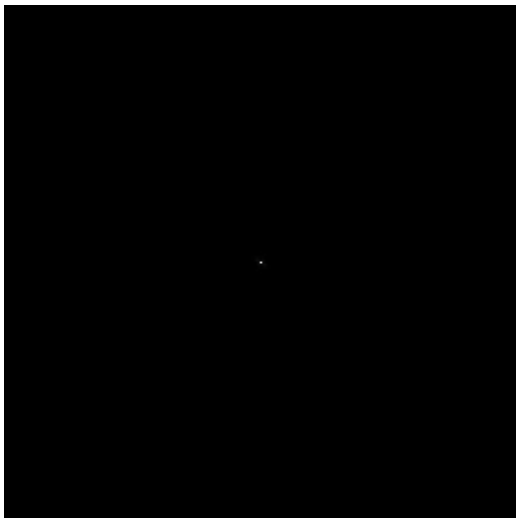
- **When n=1 and $D_0$ = 5 :**

  - Total Power Before Filter = 260236144.000000
  - Total Power After Filter = 29619746.000000
  - Remaining Power: 11.381873%

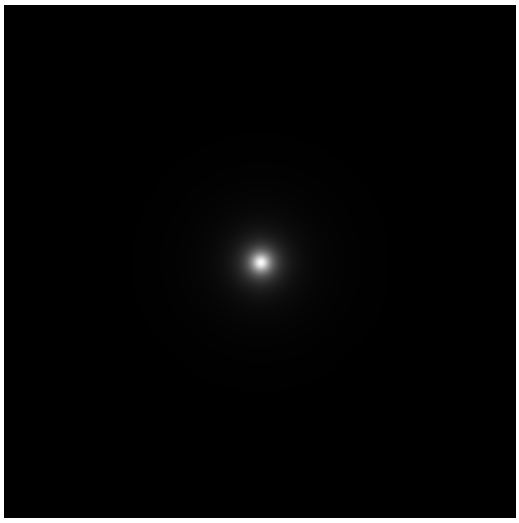
  - Magnitude before filter:



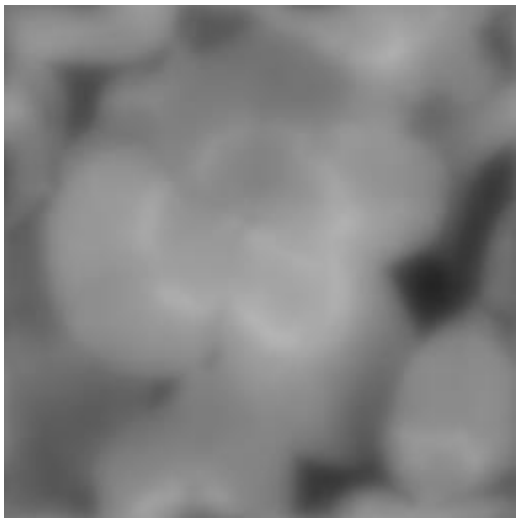
  - Magnitude after filter:



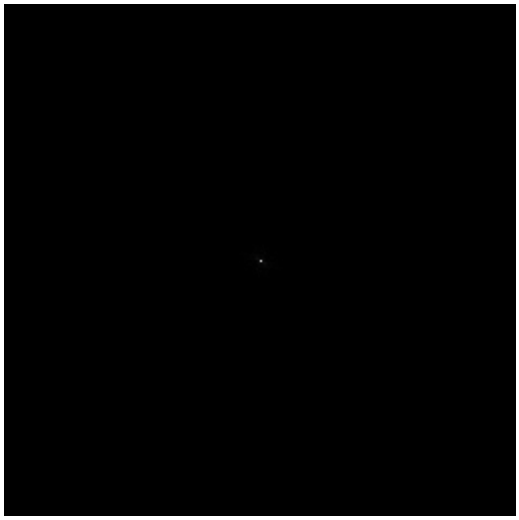
- Butterworth filter:



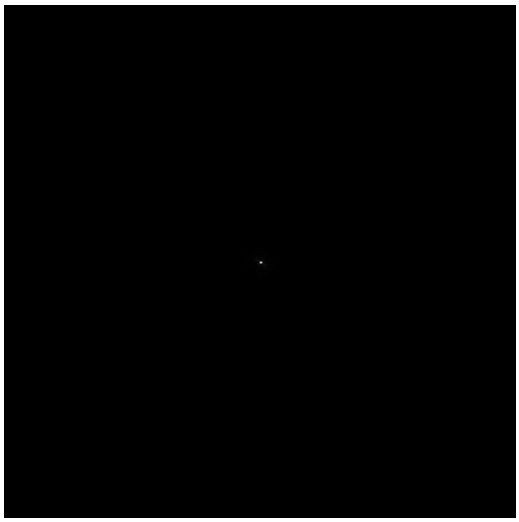
- Filtered image:



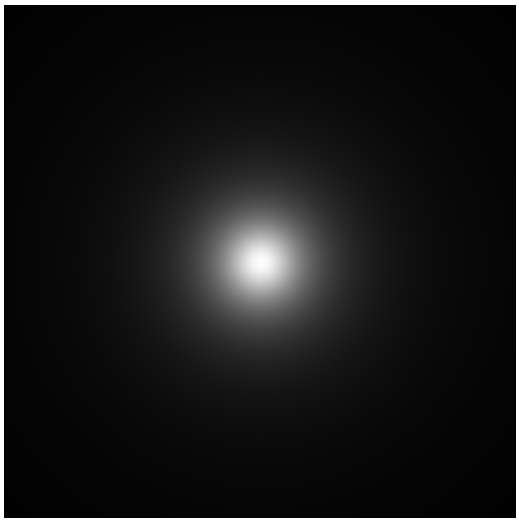
- **When n = 1 and $D_0$ = 20 :**
  - Total Power Before Filter = 260236144.000000
  - Total Power After Filter = 78034328.000000
  - Remaining Power: 29.985968%
  - Magnitude before filter:



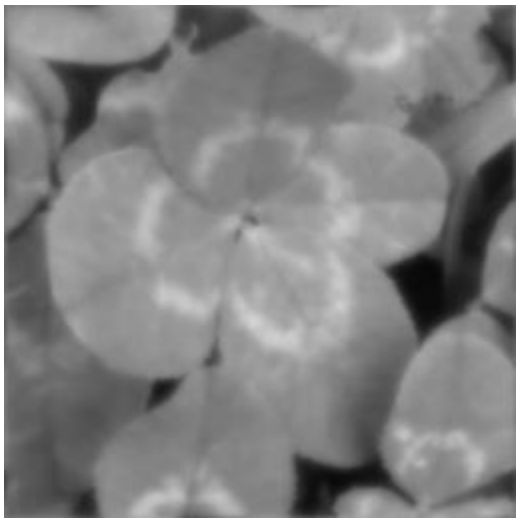
  - Magnitude after filter:



- Butterworth filter:

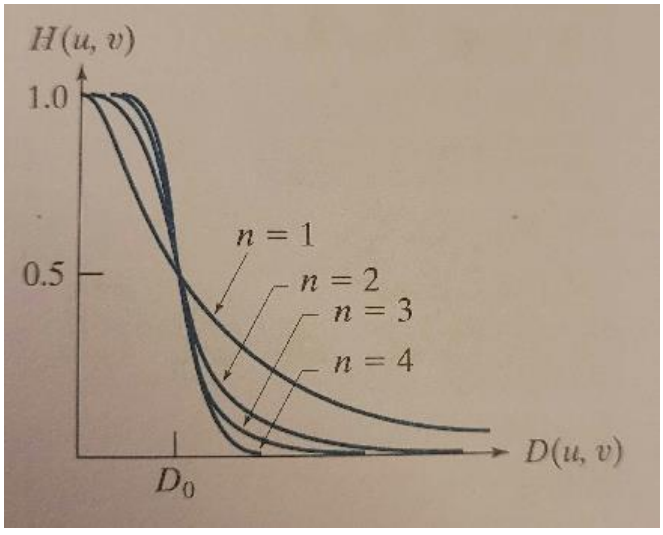


- Filtered image:



# Observations:

As the radius ($D_0$) of the filter increases the less frequencies are attenuated and thus the less blurring occurs in the spatial domain. Additionally, higher order filters approach the ideal low pass filter and have a sharper cutoff as shown in the following graph, but generate more ringing. Furthermore, normalizing and storing the Butterworth filter as an image was very helpful to debugging the filter values and making sure that the filter was performing the expected function.




# Conclusions:

The fast fourier transform is a very useful and ubiquitous algorithm in signal processing and has allowed engineers to filter and manipulate signals much more efficiently in the frequency domain. Additionally, the Butterworth low pass filter is a relatively simple frequency domain filter that can effectively and selectively smooth an image and reduce its total power. Both of these techniques are very powerful in image processing and other signal processing fields.

# Source Code:

```
/*
Computer Vision
Assignment 2 Program 1
Author: Pablo Ruiz
*/

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

/// @brief Reads an image from a file
/// @param image output array of the image intensities
/// @param fileName full name of the image to read, ie. "image1.pgm"
/// @param pgmType output representing type of pgm encoding used P1-P6
/// @param sizeX output x dimension of the image intensity array
/// @param sizeY output y dimension of the image intensity array
/// @return If succesful return 0, otherwise return error code
unsigned Read_PGM (const char *fileName,
unsigned char *&image,
char *&pgmType,
unsigned &sizeX,
unsigned &sizeY,
unsigned normal=255)
{
unsigned levels, size;
pgmType = (char*)malloc(sizeof(char) * 3);

// Read Image
FILE *iFile = fopen(fileName,"r");
if(iFile == 0) return 1; //return read failure error code
if(4 != fscanf(iFile, "%s %u %u %u ", pgmType, &sizeX, &sizeY, &levels)) return 2; //return scan error code
```

```
size = sizeX * sizeY;
image = (unsigned char *) malloc(size);
fread(image,sizeof(unsigned char),size,iFile);
fclose(iFile);
return 0;
}


/// @brief normalize an array of floats with a known max value (upper limit)
/// @param data input array of floats, they will be normalized in place
/// @param size size of the input array
/// @param max current max value of the data in the array
/// @param norm upper limit of the output array, defaults to 255
void Norm(float *data, float max, unsigned size, float norm = 255)
{
// Normalize intensity values
float factor = norm / max;
for (unsigned i = 0; i < size; ++i)
{
data[i] *= factor;
}
}

/// @brief normalize an array of floats
/// @param data input array of floats, they will be normalized in place
/// @param size size of the input array
/// @param norm upper limit of the output array, defaults to 255
void Norm(float *data, unsigned size, float norm = 255)
{
float max = data[0];
//find max
for (unsigned i = 0; i < size; ++i)
{
max = data[i] > max ? data[i] : max;
}
Norm(data, max, size, norm);
}

/// @brief Creates a .pgm image file with the passed name and writes
/// the passed image data to it
/// @param fileName full name of the image to write, ie. "image1.pgm"
/// @param image image data
/// @param pgmType type of pgm encoding used P1-P6
/// @param sizeX x dimension of the image data passed
/// @param sizeY y dimension of the image data passed
```

```
/// @param level max level range of the image data passed
/// @return If successful return 0, otherwise return error code
unsigned Write_PGM (const char *fileName,
unsigned char *image,
char *pgmType,
unsigned sizeX,
unsigned sizeY,
unsigned level=255)
{
//Write image to file
FILE *iFile = fopen(fileName,"w");
if(iFile==0) return 1; //return error code failed to open
fprintf(iFile, "%s\n%u %u\n%u\n", pgmType,sizeX,sizeY,level);//write header
fwrite(image, sizeof(unsigned char), sizeX * sizeY, iFile); //write binary image
fclose(iFile);
return 0;
}

/// @brief Converts an array of unsigned values to an array of clpx objects
/// @param data array of unsigned values
/// @param size size of the array
/// @return the clpx array
cplx* Unsigned_2_Cplx(unsigned char *data, unsigned size)
{
cplx *out = (cplx*) calloc(sizeof(cplx), size);
for (unsigned i = 0; i < size; ++i)
{
out[i].real = data[i];
}
return out;
}

/// @brief Converts an array of cplx objects to an array of unsigned values
/// @param data array of cplx objects
/// @param size size of the array
/// @return the unsigned array
unsigned char *Cplx_2_Unsigned(cplx *data, unsigned size)
{
unsigned char *out = (unsigned char*) calloc(sizeof(unsigned char), size);
for (unsigned i = 0; i < size; ++i)
{
out[i] = data[i].real;
}
return out;
}
```

```
/// @brief Make a copy of a cplx array
/// @param source cplx array to copy
/// @param size size of the array
/// @return pointer to the new array
cplx *Copy_Cplx(cplx *source, unsigned size)
{
cplx *out = (cplx*) calloc(sizeof(cplx), size);
for(unsigned i = 0; i < size; ++i)
{
out[i] = source[i];
}
return out;
}

/// @brief calculate the magnitude of a cplx array and return
/// its corresponding array of magnitudes
/// @param data input cplx array
/// @param size size of the input array
/// @return the magnitude array
float *Mag(cplx *data, unsigned size)
{
float *out = (float*)calloc(sizeof(float), size);
for (unsigned i = 0; i < size; ++i)
{
out[i] = sqrtf(powf(data[i].real, 2) + powf(data[i].imag, 2));
}
return out;
}

/// @brief Converts an array of floats to an array of unsigned chars
/// @param data Input array of floats
/// @param size Size of the input array
/// @return A pointer to the output array of unsigned chars
unsigned char *Float_2_Unsigned(float *data, unsigned size)
{
unsigned char *out = (unsigned char*)calloc(sizeof(unsigned char), size);
for (unsigned i = 0; i < size; ++i)
{
out[i] = round(data[i]);
}
return out;
}

/// @brief Print a 1D array as a 2D array of floats
```

```
/// @param data 1D input array of floats of size sizeX*sizeY
/// @param sizeX X dimension of the 2D array
/// @param sizeY Y dimension of the 2D array
void Print_2D_Array(float *data, unsigned sizeX, unsigned sizeY)
{
for (unsigned i = 0; i < sizeY; ++i)
{
for (unsigned j = 0; j < sizeX; ++j)
{
printf("%.2f, ", data[sizeX * i + j]);
}
printf("\n");
}
printf("\n");
}

/// @brief Print a 1D cplx array as a 2D array of complex numbers
/// @param data 1D input array of cplx of size sizeX*sizeY
/// @param sizeX X dimension of the 2D array
/// @param sizeY Y dimension of the 2D array
void Print_2D_Array(cplx *data, unsigned sizeX, unsigned sizeY)
{
unsigned index;
for (unsigned i = 0; i < sizeY; ++i)
{
for (unsigned j = 0; j < sizeX; ++j)
{
index = sizeX * i + j;
printf("%.2f%.2fi", data[index].real, data[index].imag);
}
printf("\n");
}
printf("\n");
}

/// @brief Calculate the total power (sum) of a 1-D array
/// @param data Input 1-D array of floats
/// @param size Size of input array
/// @return The total power (sum)
float Total_Power(float *data, unsigned size)
{
float power = 0;
for (unsigned i = 0; i < size; ++i)
{
power += data[i];
```

```
}
return power;
}

/// @brief Apply a butterworth lowpass filter to an input array and
/// generates a new filtered output array
/// @param n degree for the filter
/// @param Do filter radius
/// @param B Output float array which contains the butterworth filter
/// @param data array to be filtered
/// @param N size of one vertice of the array (ie. N=64 for a 64x64 filter)
/// @return filtered array
cplx* BWF(unsigned n, unsigned Do, float *&B, cplx *data, unsigned N)
{
float D;
cplx *out = (cplx*)calloc(sizeof(cplx), N*N);
B = (float*)calloc(sizeof(float), N*N);

//create filter
for (unsigned u = 0; u < N; ++u)
{
for (unsigned v = 0; v < N; ++v)
{
unsigned index = u * N + v;
float u2 = u;
float v2 = v;
D = sqrtf(powf((float)u2-(float)N/2, 2) + powf((float)v2-(float)N/2, 2));
B[index] = 1 / (1+powf(D/Do, 2*n));
out[index].real = data[index].real * B[index];
out[index].imag = data[index].imag * B[index];
}
}
return out;
}

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

unsigned sizeX, sizeY, size; //image width, height, size
unsigned char *image; //image array
cplx *spectrum, *output; //struct image array
float *magnitude, *magnitudeFiltered, *B, power, powerFiltered;
char *pgmType;
char *nameRead = argv[1];
unsigned degree = atoi(argv[2]);
unsigned radius = atoi(argv[3]);
```

```
//Read image and normalize the values
Read_PGM(nameRead, image, pgmType, sizeX, sizeY);
size = sizeX*sizeY;
//Convert unsigned array to cplx array
spectrum = Unsigned_2_Cplx(image, sizeX * sizeY);

//call 2D fft
fft_Four2((float*)spectrum, sizeX, sizeY, false);
//Print spectrum array
//printf("Spectrum:\n");
//Print_2D_Array(spectrum, 25, 25);

//Calculate magnitude array
magnitude = Mag(spectrum, size);

//Calculate the total power before the filter
power = Total_Power(magnitude, size);

//Print magnitude array
//printf("Magnitude:\n");
//Print_2D_Array(magnitude, 25, 25);

//Normalize magnitude array
Norm(magnitude, size);

//Save magnitude as image
Write_PGM("Magnitude.pgm", Float_2_Unsigned(magnitude,size), pgmType, sizeX, sizeY);

//Apply Butterworth filter
output = BWF(degree, radius, B, spectrum, sizeX);

//Print Butterworth filter
//printf("Butterworth Filter:\n");
//Print_2D_Array(B, 25, 25);

//Normalize Butterworth filter
Norm(B, 1.0, size);

//Save Butterworth filter as image
Write_PGM("Butterworth.pgm", Float_2_Unsigned(B, size), pgmType, sizeX, sizeY);

//Calculate magnitude of filtered image
magnitudeFiltered = Mag(output, size);
```

```
//Calculate total power
powerFiltered = Total_Power(magnitudeFiltered, size);

//Print total power after filtering and print the percentage compared to the original
printf("\nTotal Power Before Filter = %f\n", power);
printf("\nTotal Power After Filter = %f\n", powerFiltered);
printf("\nRemaining Power: %f%%\n\n", powerFiltered/power*100);

//Normalize filtered magnitude
Norm(magnitudeFiltered, size);

//Save filtered magnitude
Write_PGM("Magnitude_Filtered.pgm", Float_2_Unsigned(magnitudeFiltered, size), pgmType, sizeX, sizeY);

//call inverse 2D fft
fft_Four2((float*) output, sizeY, sizeX, true);

//convert cplx array to unsigned array
//Write the image data to an output file
image = Cplx_2_Unsigned(output, size);
Write_PGM("Output.pgm", image, pgmType, sizeX, sizeY);

return 0;
}
```