FLORIDA INTERNATIONAL UNIVERSITY

Department of Electrical and Computer Engineering

Digital Image Processing

Implementation of the 2 Dimension Fast Fourier Transform (FFT) of an Image

Pablo Gomez, Ph.D.

Miami, October 12, 2002

OBJECTIVE

The purpose of this project is to develop and implement an algorithm to obtain the 2D FFT (2 Dimension Fast Fourier Transform) of an image. The implementation is done in C++ language.

MATHEMATICAL BACKGROUND

Let's first review the 1D FFT (1 Dimension Fast Fourier Transform). The Discrete Fourier Transform is defined by:

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

Let $W_N = e^{-j2\pi/N}$ then $W_N^{ux} = e^{-j2\pi ux/N}$

And let M=N/2, we have

$$F(u) = \frac{1}{2M}\left[\sum_{x=0}^{N/2-1} f(2x)W_N^{u2x} + \sum_{x=0}^{N/2-1} f(2x+1)W_N^{u(2x+1)}\right]$$

$$F(u) = \frac{1}{2M}\left[\sum_{x=0}^{N/2-1} f(2x)W_N^{u2x} + \sum_{x=0}^{N/2-1} f(2x+1)W_N^{u2x}W_N^{u}\right] \qquad (1.0)$$

Let

$$F_{even}(u) = F_e(u) = \frac{1}{M}\sum_{x=0}^{M-1} f(2x)W_M^{u2x}, u = 0,1,..,M-1$$ and

$$F_{odd}(u) = F_o(u) = \frac{1}{M}\sum_{x=0}^{M-1} f(2x+1)W_M^{u2x}, u = 0,1,..,M-1$$

then Equation 1.0 can be expressed as

$$F(u) = \frac{1}{2}\left[F_e(u) + F_o(u)W_{2M}^{u}\right], u = 0,1,..,M-1 \qquad (2.0)$$

the above equation gives us half of the FFT. Lets use the following equalities

$W_{N/2}^{u+N/2} = W_{N/2}^{u}$ and $W_N^{u+N/2} = -W_N^{u}$

To obtain the other half of the FFT,

$$F(u+M)=\frac{1}{2}\left[\frac{1}{M}\sum_{x=0}^{M-1}f(2x)W_M^{(u+M)x}+\frac{1}{M}\sum_{x=0}^{M-1}f(2x+1)W_M^{(u+M)x}W_{2M}^{u+M}\right]$$

since $W_M^{u+M}=W_M^u$ and $W_{2M}^{u+M}=-W_{2M}^u$ we obtain

$$F(u+M)=\frac{1}{2}\left[F_e(u)-F_o(u)W_N^u\right] \qquad (3.0)$$

To visually understand the FFT algorithm, lets assume N=16, that is M=N/2=8. Equation 1.0 becomes,

$$F(u)=\frac{1}{2*8}\left[\sum_{x=0}^{8-1}f(2x)W_M^{ux}+\sum_{x=0}^{8-1}f(2x+1)W_M^{ux}W_N^u\right]$$

Figure 2 shows the detailed 1D Radix-2 Decimated in Time FFT. The transformation takes place in *$log_2(N)$* stages. Each stage processes N/2 butterflies, which is the name of the transformation structure in Figure 1 as follows

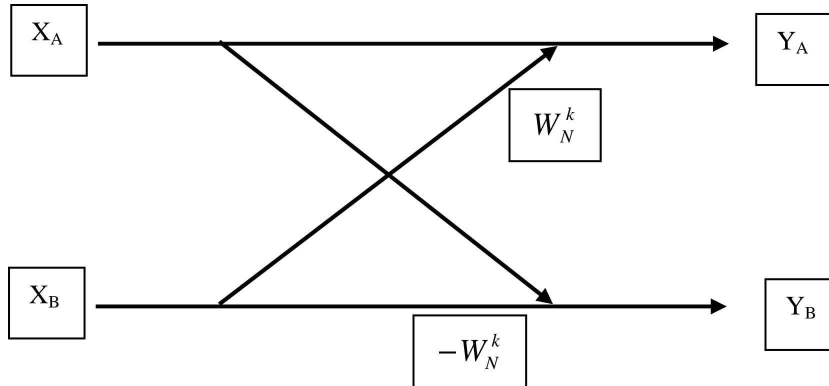

Figure 1. An FFT Butterfly.

The butterfly calculates the following 2 expressions:

$$Y_A=X_A+W_N^kX_B \qquad (4.0)$$

and

$$Y_B=X_A-W_N^kX_B \qquad (5.0)$$

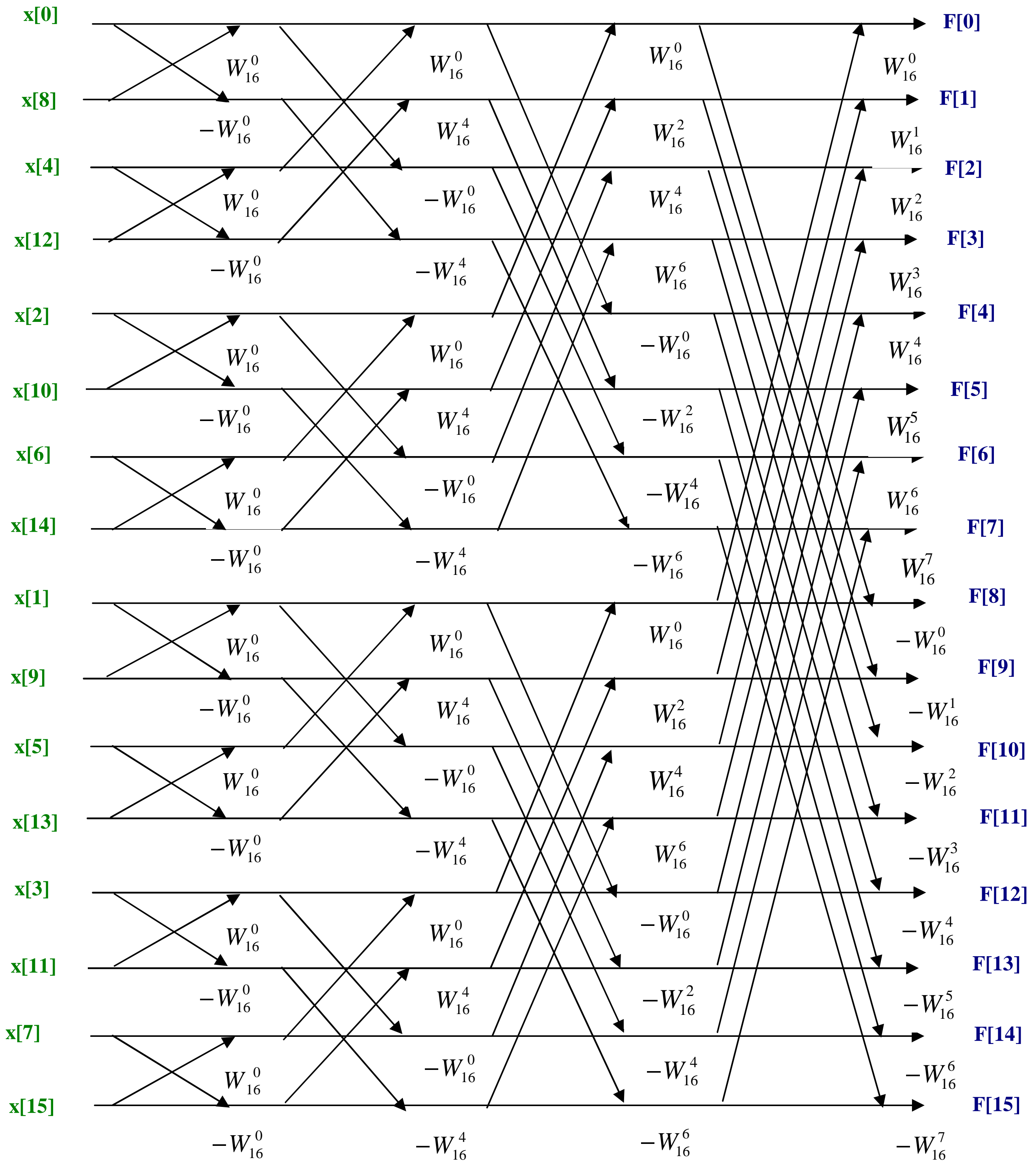

Figure 2. Radix-2 Decimated in Time FFT.

Using Euler's formula

$$e^{j\theta} = \cos(\theta) + j\sin(\theta)$$

Equations 4.0 and 5.0 can be expressed as

$$Y_A = X_A + e^{-j2\pi k/N} X_B$$
$$Y_B = X_A - e^{-j2\pi k/N} X_B$$

$$Y_A = X_A + (\cos(-2\pi k/N) + j\sin(-2\pi k/N))X_B$$
$$Y_B = X_A - (\cos(-2\pi k/N) + j\sin(-2\pi k/N))X_B$$

$$Y_A = X_A + (\cos(2\pi k/N) - j\sin(2\pi k/N))X_B \qquad (6.0)$$
$$Y_B = X_A - (\cos(2\pi k/N) - j\sin(2\pi k/N))X_B \qquad (7.0)$$

Where X_A, X_B, Y_A, Y_B are complex variables. The algorithm implements equations 6.0 and 7.0 by separating all the variables into their real and imaginary parts.

DETAILED ALGORITHM.

To better understand the actual implemented FFT algorithm from the programmatic point of view, the FFT process shown in Figure 2 was extended to identify the characteristics of every step on each of the stages and how every stage/butterfly is actually processed.

Figure 3 shows additional arrows/curves: red curves indicate blocks and number of consecutive butterflies on each stage; purple vertical arrows indicate the "jump" that the program needs to make to point to the next block of butterflies; when the "jump" goes beyond N, all blocks in the current stage have been processed; the FFT is progressively calculated from left to right: the first set of values is obtained by storing the input vector in bit-reverse sequence. This process is indicated in Table 2.

Table 1 summarizes the behavior of the FFT algorithm at each stage. The bottom row shows the general formula to obtain each value and control the flow of the program.

Stage	Consecutive Butterflies	Length of Jump	Twiddle Factors
i=0	1	2	k=0 +/-W^0
i=1	2	4	k=0 +/-W^0 k=1 +/-W^4
i=2	4	8	k=0 +/-W^0 k=1 +/-W^2 k=2 +/-W^4 k=3 +/-W^6
i=3	8	16	k=0 +/-W^0 k=1 +/-W^1 k=2 +/-W^2 k=3 +/-W^3 k=4 +/-W^4 k=5 +/-W^5 k=6 +/-W^6 k=7 +/-W^7
i	2^i	$2*2^i$	Exponent = $k*N/(2^{(i+1)})$

Table 1. FFT Algorithm Characteristics.

The FFT algorithm requires $\log_2(N)$ stages.

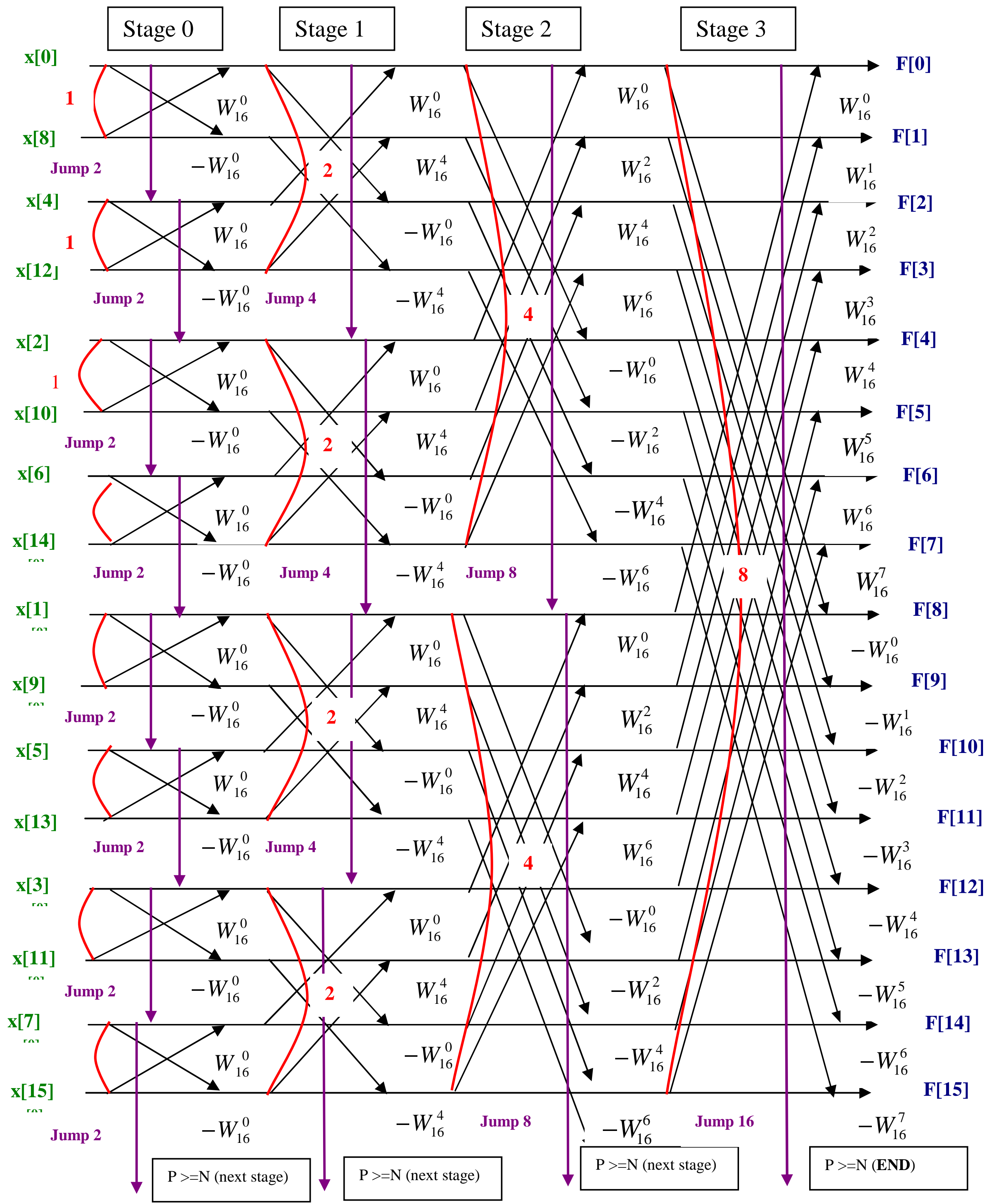

Figure 3. 1D Radix-2 16-Point Decimated in Time FFT.

The following table shows the relationship between the indices of the input vector and the output (FFT) vector. As it can be seen, the indices of the input result in reversing the bits of the sequence of the indices of the output vector. For this reason it is said that the input vector is sequenced in bit reverse order. This is the first step of the implemented FFT function.

X[I]	**X[I] Index bits**	**F[I]**	**F[I] Index bits**
X[0]	0000	F(0)	0000
X[8]	1000	F(1)	0001
X[4]	0100	F(2)	0010
X[12]	1100	F(3)	0011
X[2]	0010	F(4)	0100
X[10]	1010	F(5)	0101
X[6]	0110	F(6)	0110
X[14]	1110	F(7)	0111
X[1]	0001	F(8)	1000
X[9]	1001	F(9)	1001
X[5]	0101	F(10)	1010
X[13]	1101	F(11)	1011
X[3]	0011	F(12)	1100
X[11]	1011	F(13)	1101
X[7]	0111	F(14)	1110
X[15]	1111	F(15)	1111

Table 2. Bit Reverse Sequence.

Lets now proceed to define the DFT in 2 dimensions:

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

for *u=0,1,…,N-1 and v=0,1,…N-1*

assuming that the image *f(x,y)* is an *NxN* matrix.

The above equation can be also expressed as

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

$$F(u,v)=\sum_{x=0}^{N-1}F(x,v)e^{-j2\pi ux/N}$$

where

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

for each value of x, and for values of v=0,1,…,N-1, this equation is a complete 1-D Fourier Transform. In other words, F(x,v) is the Fourier Transform along one row of f(x,y). By varying x from 0 to N-1, we compute the Fourier Transform of all rows of the image matrix. The next step is to compute the 1-D Fourier Transform of the previously obtained matrix, column by column. In summary, Figure 4 shows the separability principle of the 2D DFT.

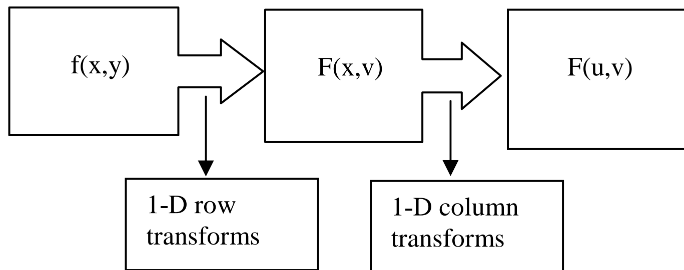

Figure 4. 2D Fourier Transform by separation of planes.

For the purpose of displaying the results, we are interested in the Power Spectrum of the FFT, defined by:

$$P(u,v)=\sqrt{R^2(u,v)+I^2(u,v)}$$

where *R* and *I* are the real and imaginary parts of the FFT, *F(u,v),* respectively.

To be able to display the power spectrum, we need to normalize its values so they fall in the range 0 through 255. Also, to enhance the resulting image, the following logarithmic function will be used:

$$PS(u,v)=\frac{255}{\log_{10}(256)}\log_{10}\left(1+\frac{255P(u,v)}{Max(P(u,v))}\right)$$

To center the FFT, we need to translate the input image, *f(x,y)*, by multiplying each element by $(-1)^{(x+y)}$. The resulting equation is:

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

SIMULATION

The algorithm was first implemented using MATLAB. Script *riFFT.m* was written to obtain the 1D FFT of a vector and compare the results with MATLAB function *fft()*. The following is the MATLAB script for the *riFFT.m* script:

```
%******************************************
% Radix-2 Decimated in Time FFT.
%     FR = Real[FFT(f)]
%     FI = Imaginary[FFT(f)]
%******************************************
% Size of FFT
N = 16;
% Input Signal:
f = randn(N,1);

% Obtain indexes in bitreverse mode:
br = zeros(N,1);
for i=1:N
   v=i-1;
   for bit=1:log2(N)
      z     = bitget(v,log2(N)+1-bit);
      br(i) = bitset(br(i),bit,z);
   end
end
br = br + 1; % Due to MATLAB indexing mode (1...N)

% Initial Stage: FR(i) = f(br(i))
FR = zeros(N,1);
FI = zeros(N,1);
for i=1:N
   FR(i) = f(br(i));
end




% There are log2(N) Stages:
for i=0:log2(N)-1
   % Length of jump to next butterfly set:
   jump = 2*2^i;
   p = 0;
   while p < N
```

```
        u = p;
        % Consecutive Butterflies (size of set = 2^i)
        for k=0:2^i-1
            % Butterfly inputs, Xa and Xb are F(u) and F(u + 2^i)
            RXa = FR(u+1);
            IXa = FI(u+1);
            RXb = FR(u+1 + 2^i);
            IXb = FI(u+1 + 2^i);
            % Exponent of twiddle factor W:
            w  = k*N/(2^(i+1));
            % Obtain Butterfly outputs, Ya and Yb:
            RYa = RXa + RXb * cos(2*pi*w/N) + IXb * sin(2*pi*w/N);
            IYa = IXa - RXb * sin(2*pi*w/N) + IXb * cos(2*pi*w/N);
            RYb = RXa - RXb * cos(2*pi*w/N) - IXb * sin(2*pi*w/N);
            IYb = IXa + RXb * sin(2*pi*w/N) - IXb * cos(2*pi*w/N);
            % Update F(u)
            FR(u+1)  = RYa;
            FI(u+1)  = IYa;
            FR(u+1 + 2^i) = RYb;
            FI(u+1 + 2^i) = IYb;
            % next Butterfly:
            u = u + 1;
        end
        % Next set of Butterflies:
        p = p + jump;
    end
    % Next Stage:
end
```

THE SOLUTION

Once the solution was simulated using MATLAB, the algorithms were implemented in Microsoft Visual C++ 6.0. The following diagram shows the components of the automated solution to this problem.

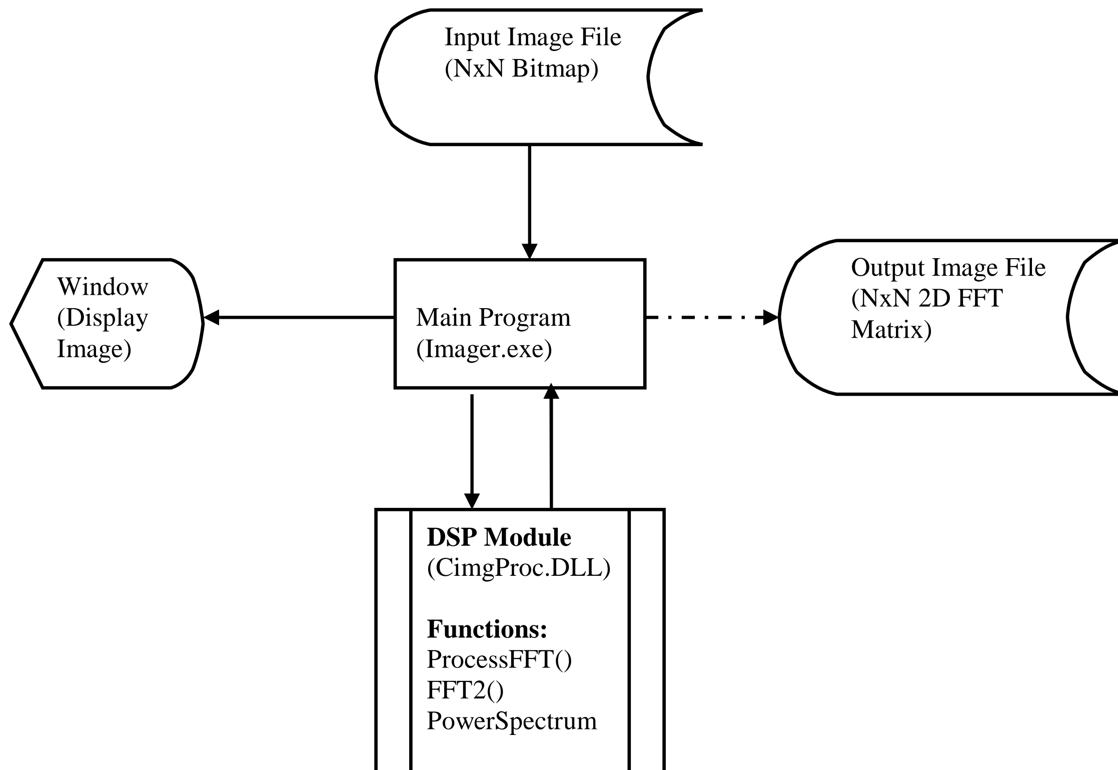

Figure 5. Program Components.

The input to the main program is a bitmap image file, 256x256, 3 color planes. The main program passes the image pixel information to the DSP Module (DLL file), which performs the computation of the ***Normalized Power Spectrum of the FFT*** of the 3 color planes of the input image and returns the results back to the main program. The main program displays the image on a window, and optionally saves it to disk.

The main program, *Imager.exe*, was provided along with its documentation, so the actual work done in this project was to code the ***ProcessFFT()***, ***FFT2()*** and ***PowerSpectrum()*** functions contained in the *CimgProc.cpp* source file. Other support functions were developed to perform the bit reverse sequence vector: *getbit(), setbit(), resetbit()* and *log2()*. This file was compiled and built as a DLL as expected by the main program.

IMPLEMENTATION

The C++ implementation is a set of 3 main functions, ***ProcessFFT()***, ***FFT2()*** and ***PowerSpectrum()*** and 4 auxiliary functions, ***getbit(), setbit(), resetbit() and log2()***.

The main function, ***ProcessFFT(),*** retrieves the image string, obtains 3 color arrays corresponding to the RGB components, centers the arrays and converts them to double format, calls the ***FFT2()*** and ***PowerSpectrum()*** functions 3 times, once per each color plane, and the resulting matrices are converted back to a single string, as expected by the caller program.

This is the code of function ***ProcessFFT():***

```
//==========================================================
// Main Control function to perform the FFT of an the Image
//==========================================================
void ProcessFFT( LPSTR pszImage, long Y, long X, long Colors){
	ColorArray2D R, G, B;		// Color Planes (Input)
	DArray2D	dR, dG, dB;	// Color Planes in Double format
	DArray2D	FR, FI;		// FFT matrices (Real and Imaginary)
	int		N= X;		// Assume square matrices

	StringToArrays(pszImage, X, Y , Colors, R,G,B);
	// Center Arrays:
	redimArray(N,N,dR);
	redimArray(N,N,dG);
	redimArray(N,N,dB);
	for (int u=0;u<N;u++){
		for (int v=0;v<N;v++){
			dR[u][v] = pow(-1,u+v)*R[u][v];
			dG[u][v] = pow(-1,u+v)*G[u][v];
			dB[u][v] = pow(-1,u+v)*B[u][v];
		}
	}
	// obtain the 3 FFT planes:
	FFT2(dR,N, FR,FI);
	R = PowerSpectrum(FR,FI,N);
	FFT2(dG,N, FR,FI);
	G = PowerSpectrum(FR,FI,N);
	FFT2(dB,N, FR,FI);
	B = PowerSpectrum(FR,FI,N);

	// Back to string:
	ArraysToString(pszImage, X , Y, Colors, R,G,B);

}
```

The following is the code of function ***FFT2()*** which does the actual work of obtaining the Fast Fourier Transform of an image:

```
//===========================================================================
// Calculates the 2D FFT of an input NxN matrix
//===========================================================================
void FFT2 ( DArray2D f, int N, DArray2D &FR, DArray2D &FI){
//===========================================================================
    //---------------------------------------------------------------------------
    // INPUTS  = Color Plane NxN in double format/centered ==> f
    //           FFT Size                                   ==> N
    // OUTPUTS = 2D FFT Real Part Matrix NxN                ==> FR
    //           2D FFT Imaginary Part Matrix NxN           ==> FI
    //---------------------------------------------------------------------------

    int       i;          // FFT Stages counter and general purpose
    int       row;        // FFT matrix row
    int       col;        // FFT matrix column
    int       stages;     // Number of FFT stages (log2(N)
    double    RZ[256];    // Real Buffer Vector
    double    IZ[256];    // Imaginary Buffer Vector
    int       br[256];    // Bit reverse sequence vector
    double    RYa;        // Butterfly Output A - Real
    double    IYa;        // Butterfly Output A - Imaginary
    double    RXa;        // Butterfly Input  A - Real
    double    IXa;        // Butterfly Input  A - Imaginary
    double    RYb;        // Butterfly Output B - Real
    double    IYb;        // Butterfly Output B - Imaginary
    double    RXb;        // Butterfly Input  B - Real
    double    IXb;        // Butterfly Input  B - Imaginary

    int       jump;       // Length of jump to next set of butterflies
    int       w;          // Exponent of twiddle factor Wn,k
    int       p;          // pointer to next butterfly set
    int       k;          // counter of consecutive butterflies
    int       u;          // index of consecutive butterflies
    int       bit;        // bit number

    stages = log2(N);

    // obtain the bit reverse sequence:
    for (i=0;i<N;i++){
        br[i] = 0;
        for (bit=0;bit<stages;bit++){
            if (getbit(i,stages-1-bit) == 1)
            {
```

```
                        br[i] = setbit(br[i],bit);
                    }
                    else
                    {
                        br[i] = resetbit(br[i],bit);
                    }
            }
    }

    redimArray(N,N,FR);
    redimArray(N,N,FI);

// Process the 1D - FFT on each row:
for (row=0;row<N;row++){
    //load row data in bit reverse sequence:
    for (i=0;i<N;i++){
        FR[row][i] = f[row][br[i]];
        FI[row][i] = 0;
    }
    //calculate FFT of current row:
   for (i=0;i<stages;i++){
        jump = 2*(int)pow(2,i);
        p = 0;
       do {
           u = p;
           for (k=0;k<(int)pow(2,i);k++){
                    RXa = FR[row][u];
                    IXa = FI[row][u];
                    RXb = FR[row][u + (int)pow(2,i)];
                    IXb = FI[row][u + (int)pow(2,i)];
                    // Exponent of twiddle factor W:
                    w  = k*N/(int)pow(2,(i+1));
                    // Obtain Butterfly outputs, Ya and Yb:
                    RYa = RXa + RXb * cos(2*pi*w/N) + IXb * sin(2*pi*w/N);
                    IYa  = IXa   - RXb  * sin(2*pi*w/N) + IXb * cos(2*pi*w/N);
                    RYb = RXa - RXb  * cos(2*pi*w/N) -  IXb * sin(2*pi*w/N);
                    Iyb   = IXa  + RXb  * sin(2*pi*w/N) -  IXb * cos(2*pi*w/N);
                    // Update F(u)
                    FR[row][u]  = RYa;
                    FI[row][u]  = IYa;
                    FR[row][u + (int)pow(2,i)] = RYb;
                    FI[row][u + (int)pow(2,i)] = IYb;
                    // next Butterfly:
                    u = u + 1;
           }
           // Next set of Butterflies:
```

```
            p = p + jump;
        } while (p<N);
    }
    //next row
}

// Process the 1D - FFT on each column of previous results:
for (col=0;col<N;col++){
    //save current column in buffer vectors:
    for (i=0;i<N;i++){
        RZ[i] = FR[i][col];
        IZ[i] = FI[i][col];
    }
    //load column data in bit reverse sequence:
    for (i=0;i<N;i++){
        FR[i][col] = RZ[br[i]];
        FI[i][col] = IZ[br[i]];
    }
    //calculate FFT of current column:
    for (i=0;i<stages;i++){
        jump = 2*(int)pow(2,i);
        p = 0;
        do {
            u = p;
            for (k=0;k<(int)pow(2,i);k++){
                RXa = FR[u][col];
                IXa = FI[u][col];
                RXb = FR[u + (int)pow(2,i)][col];
                IXb = FI[u + (int)pow(2,i)][col];
                // Exponent of twiddle factor W:
                w  = k*N/(int)pow(2,(i+1));
                // Obtain Butterfly outputs, Ya and Yb:
                RYa = RXa + RXb * cos(2*pi*w/N) + IXb * sin(2*pi*w/N);
                IYa = IXa - RXb * sin(2*pi*w/N) + IXb * cos(2*pi*w/N);
                RYb = RXa - RXb * cos(2*pi*w/N) - IXb * sin(2*pi*w/N);
                IYb = IXa + RXb * sin(2*pi*w/N) - IXb * cos(2*pi*w/N);
                // Update F(u):
                FR[u][col]  = RYa;
                FI[u][col]  = IYa;
                FR[u + (int)pow(2,i)][col] = RYb;
                FI[u + (int)pow(2,i)][col] = IYb;
                // next Butterfly:
                u = u + 1;
            }
            // Next set of Butterflies:
            p = p + jump;
```

```
        } while (p<N);
    }
    //next column
}

} // End of Function
```

The code for the ***PowerSpectrum()*** Function is as follows:

```
//==============================================================
ColorArray2D PowerSpectrum(DArray2D  FR, DArray2D FI, int N){
//==============================================================
//----------------------------------------------------------------------------------
// INPUTS = Real Part of FFT          ==> FR
//                Imaginary Part of FFT ==> FI
//                        FFT Size             ==> N
// OUTPUT = Normalized smoothed Power Spectrum ==> PS
//----------------------------------------------------------------------------------
        DArray2D     P;          // Power Spectrum - not normalized
        ColorArray2D PS;      // Power Spectrum - smoothed and normalized
        double MAX;
        int u,v;

        redimArray(N,N,P);
        redimArray(N,N,PS);
        // Obtain the Power Spectrum P = sqrt (FR^2 + FI^2);
        MAX=0;
        for (u=0;u<N;u++){
                for (v=0;v<N;v++){
                        P[u][v]= sqrt(pow(FR[u][v],2) + pow(FI[u][v],2));
                        if (P[u][v] > MAX) {MAX = P[u][v];}
                }
        }
        // Normalize the power spectrum and enhance to display as an
        // image. This is the result matrix passed back to the
        // caller function.
        for (u=0;u<N;u++){
                for (v=0;v<N;v++){
                        P[u][v] = (255/log10(256))*log10(1+255*P[u][v]/MAX);
                        PS[u][v] = (BYTE)P[u][v];
                }
        }

        return PS;
}
```

The code of the auxiliary functions, **log2(), getbit(), setbit()** and **resetbit()** is shown here:

```
//-----------------------------------------------------
int log2(int N){
        int n=1;
        int x=0;
        do {
                n <<= 1;
                x++;
        } while (n != N);
        return x;
}

//----------------------------------------------------
int getbit(int x, int bit){
        int mask;
        int r;
        mask = 1;
        mask <<= bit;
        r = x & mask;
        if (r==mask) {return 1;} else {return 0;}
}

//----------------------------------------------------
int setbit(int x, int bit){
        int mask;
        mask = 1;
        mask <<= bit;
        return x | mask;
}

//----------------------------------------------------
int resetbit(int x, int bit){
        int mask;
        mask = 1;
        mask <<= bit;
        mask = ~mask;
        return x & mask;
}
```

RESULTS.

The program was tested successfully with a very good performance. The following Figure shows two 128x128 images on the left side and their corresponding FFT on the right column:

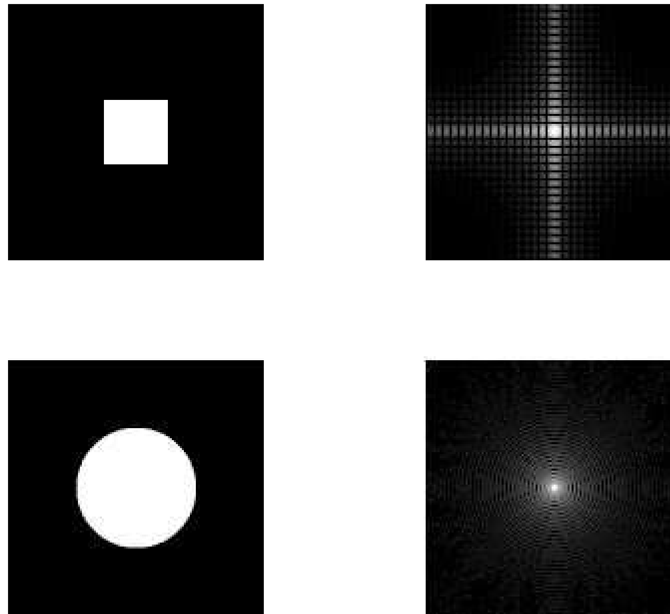

Figure 6. Experiment Results. Input Images (Left) and their FFT (Right)

The same images were tested using different sizes, from 16x16 through 256x256. The response time is summarized in Table 3, were the results of the previous Project, Direct DFT Calculation, are also shown.

Image Size	FFT processing time	DFT processing time
16x16	< 1 second	N/A
32x32	< 1 second	50 seconds
64x64	2 seconds	5 minutes
128x128	5 seconds	2 hours
256x256	22 seconds	> 15 hours

Table 3. FFT vs. Direct DFT calculation Processing Time

The difference in processing time between the direct implementation of the DFT and the FFT algorithm is of several orders of magnitude.

The implemented FFT algorithm can still be improved to obtain a better performance. Radix-4 FFT, a variation of the algorithm implemented here, yields even better performance. Another improvement that can be easily implemented is to store all the possible values of the twiddle factors (W_N^k) into a Look-Up Table. For instance, assuming that our images have a maximum of 256x256, we know in advance that we can divide the range 0-2π into 256 points and pre-calculate the sine/cosine values and store them in a look-up table. This way the program only needs to obtain the exponent, a modulo-N number, and use it as an index to obtain the corresponding entry in the table. By using this technique we avoid the multiple computations of sine/cosines at runtime.

CONCLUSION.

The design and implementation of the FFT algorithm was successfully accomplished. Very good results were achieved when computing the FFT of images of different sizes ranging from 16x16 through 256x256.

The enormous computation time difference when compared to a direct DFT algorithm yields to the conclusion that the FFT is a very powerful tool that has to be used to develop real life DSP applications.