Chapter 4
The FFT and Power Spectrum Estimation

Contents

Slide 1 The Discrete-Time Fourier Transform
Slide 2 Data Window Functions
Slide 3 Rectangular Window Function (cont. 1)
Slide 4 Rectangular Window Function (cont. 2)
Slide 5 Normalization for Spectrum Estimation
Slide 6 The Hamming Window Function
Slide 7 Other Window Functions
Slide 8 The DFT and IDFT
Slide 9 DFT Examples
Slide 10 The Inverse DFT (IDFT)
Slide 11 The Fast Fourier Transform (FFT)
Slide 11 Decimation in Time FFT Algorithm
Slide 12 Decimation in Time FFT (cont. 1)
Slide 13 Decimation in Time FFT (cont. 2)
Slide 14 Decimation in Time FFT (cont. 3)
Slide 15 Decimation in Time FFT (cont. 4)
Slide 16 Decimation in Time FFT (cont. 5)
Slide 17 Decimation in Time FFT (cont. 6)
Slide 17 Bit Reversed Input Ordering

Slide 18 C Decimation in Time FFT Program
Slide 19 C FFT Program (cont. 1)
Slide 20 C FFT Program (cont. 2)
Slide 21 C FFT Program (cont. 3)
Slide 22 C FFT Program (cont. 4)
Slide 23 C FFT Program (cont. 5)
Slide 24 C FFT Program (cont. 6)
Slide 25 C FFT Program (cont. 7)
Slide 26 Estimating Power Spectra by FFT's
Slide 26 The Periodogram and Sample Autocorrelation Function
Slide 27 Justification for Using the Periodogram
Slide 28 Averaging Periodograms
Slide 29 Efficient Method for Computing the Sum of the Periodograms of Two Real Sequences

Slide 30 **Experiment 4.1 The FFT**
Slide 31 FFT Experiments (cont. 1)
Slide 32 FFT Experiments (cont. 2)
Slide 33 FFT Experiments (cont. 3)

Slide 34 **Experiment 4.2 Power Spectrum Estimation**
Slide 35 Making a Spectrum Analyzer
Slide 35 Ping-Pong Buffers

Slide 36	Ping-Pong Buffers (cont.)
Slide 37	Spectrum Estimation (cont. 1)
Slide 38	Spectrum Estimation (cont. 2)
Slide 39	Spectrum Estimation (cont. 3)
Slide 40	Another Method of Averaging Periodograms
Slide 41	Testing Your Spectrum Analyzer
Slide 42	Initial Testing (cont.)
Slide 43	Displaying the Spectrum Using CCS
Slide 44	Displaying the Spectrum (cont.)
Slide 45	Testing with External Inputs
Slide 46	Testing with External Inputs (cont. 1)
Slide 47	Testing with External Inputs (cont.)
Slide 48	Testing with an Exponential Averager

Chapter 4
The FFT and Power Spectrum Estimation

The Discrete-Time Fourier Transform

The discrete-time signal $x[n] = x(nT)$ is obtained by sampling the continuous-time $x(t)$ with period T or sampling frequency $\omega_s = 2\pi/T$.

The *discrete-time Fourier transform* of $x[n]$ is

$$X(\omega) = \sum_{n=-\infty}^{\infty} x[n] e^{-j\omega nT} = X(z)\big|_{z=e^{j\omega T}} \tag{1}$$

Notice that $X(\omega)$ has period ω_s.

The discrete-time signal can be determined from its discrete-time Fourier transform by the inversion integral

$$x[n] = \frac{1}{\omega_s} \int_{-\omega_s/2}^{\omega_s/2} X(\omega) e^{j\omega nT} d\omega \tag{2}$$

$=$ a “sum” of sinusoids, $e^{j\omega nT}$, scaled by $X(\omega)$.

Data Window Functions

The observed data sequence must be limited to a finite duration to compute the transform summation in practice.

The Rectangular Window Function

The most obvious approach is to simply truncate the summation to a finite range, for example, $0 \le n \le N-1$. Let the N-point rectangular data window function be

$$h_1[n] = \begin{cases} 1 & \text{for } \; n = 0, 1, \ldots, N-1 \\ 0 & \text{elsewhere} \end{cases} \tag{3}$$

Then the truncated sequence is $y[n] = x[n]h_1[n]$. Let $H_1(\omega)$, $X(\omega)$, and $Y(\omega)$ be the discrete-time Fourier transforms of $h_1[n]$, $x[n]$, and $y[n]$. Then

$$Y(\omega) = \frac{1}{\omega_s} \int_{-\omega_s/2}^{\omega_s/2} X(\lambda) H_1(\omega - \lambda)\, d\lambda \tag{4}$$

Which is a frequency domain convolution.

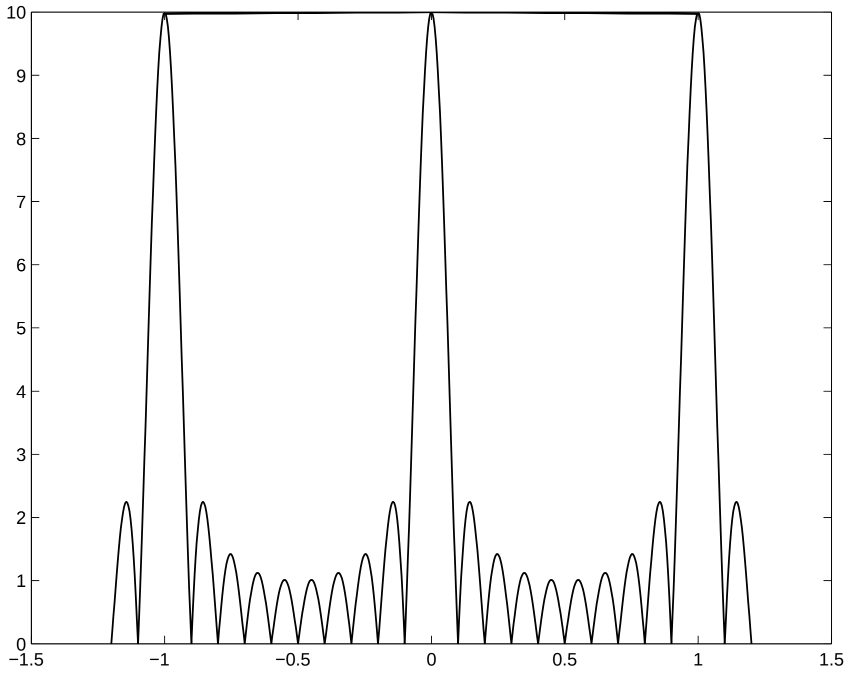
Rectangular Window Function (cont. 1)

The discrete-time Fourier transform of the rectangular window is

$$H_1(\omega) = \sum_{n=0}^{N-1} e^{-j\omega nT} = e^{-j\omega(N-1)T/2}\, H_0(\omega) \qquad (5)$$

where $$H_0(\omega) = \frac{\sin(\omega NT/2)}{\sin(\omega T/2)} \qquad (6)$$

This transform is called the *spectral window.*



Normalized Frequency ω/ω_s

Figure 1: Spectral Window for Rectangular Data Window, $|H_1(\omega)|$, for $N = 10$

Rectangular Window Function (cont. 2)

- $|H_1(\omega)|$ has a peak magnitude of N at integer multiples of ω_s and is 0 at frequencies $k\omega_s/N$ that are not multiples of ω_s.
- The main lobe at the origin has width $2\omega_s/N$.
- The transform of the truncated sum is a smoothed version of the true spectrum, $X(\omega)$, obtained by convolving $X(\omega)$ with $H_1(\omega)$.
- The value at frequency ω is predominantly an average of values in the vicinity of ω weighted by $H_1(\omega - \lambda)$ over its main lobe which extends from $\lambda = \omega - (\omega_s/N)$ to $\lambda = \omega + (\omega_s/N)$.
- The maximum sidelobe magnitude of $H_1(\omega)$ is down only about 13 dB from the main lobe peak. So $X(\omega)$ estimated by the truncated summation can be significantly distorted by large values of $X(\lambda)$ away from ω "leaking through" the spectral window.

Normalization for Spectrum Estimation

Spectral leakage can be reduced by using a data window with smaller sidelobes in its transform. For unbiased power spectral density estimates, a data window $h[n]$ should be normalized so that

$$\frac{1}{N}\sum_{n=0}^{N-1} h^2[n] = 1 \tag{7}$$

The Hanning Window

The Hanning spectral window is

$$\begin{aligned} H_2(\omega) &= c_2 e^{-j\omega(N-1)T/2}\left[0.5H_0(\omega) + 0.25H_0\left(\omega - \frac{\omega_s}{N}\right)\right. \\ &\qquad \left. + 0.25H_0\left(\omega + \frac{\omega_s}{N}\right)\right] \end{aligned} \tag{8}$$

with the corresponding data window

$$h_2[n] = \begin{cases} c_2 0.5\left\{1+\cos\left[\left(n-\frac{N-1}{2}\right)\frac{2\pi}{N}\right]\right\} & \text{for } n = 0,\ldots,N-1 \\ 0 & \text{elsewhere} \end{cases} \tag{9}$$

where $c_2 = (3/8)^{-1/2}$.

- The maximum sidelobe amplitude is down by 37.5 dB from the main lobe peak for the Hanning window.
- However, the mainlobe has width $4\omega_s/N$ which is double the width of the main lobe for the rectangular window.
- There is a trade-off between the main lobe width and peak side lobe amplitude.

The Hamming Window Function

The Hamming spectral window is

$$\begin{aligned} H_3(\omega) &= c_3 e^{-j\omega(N-1)T/2}\left[0.54 H_0(\omega) + 0.23 H_0\left(\omega - \frac{\omega_s}{N}\right)\right. \\ &\quad \left. + 0.23 H_0\left(\omega + \frac{\omega_s}{N}\right)\right] \qquad (10) \end{aligned}$$

with the corresponding data window

$$h_3[n] = \begin{cases} c_3\left\{0.54 + 0.46\cos\left[\left(n - \frac{N-1}{2}\right)\frac{2\pi}{N}\right]\right\} & \text{for } n = 0, \ldots, N-1 \\ 0 & \text{elsewhere} \end{cases} \qquad (11)$$

where $c_3 = (0.3974)^{-1/2}$.

It is almost the same as the Hanning window. Its spectral sidelobes are down by at least 40 dB.

Other Window Functions

See DSP books for the Blackman and Kaiser windows.

- The Blackman spectral window is formed by adding in shifts of $H_1(\omega)$ to the right and left by $2\omega_s/N$ as well as the shifts of ω_s/N for the Hamming and Hanning windows. The width of the main lobe is $6\omega_s/N$ but the peak sidelobes are down by 80 dB.
- The Kaiser window approximates the prolate spheroidal waveforms and has a parameter that can be varied to trade-off the main lobe width and peak sidelobe level. Excellent designs can be achieved.
- It seems to be a "law of nature" that the main lobe width must be increased to reduce sidelobe levels.

The Discrete Fourier Transform and its Inverse

Let $x_0, x_1, \ldots, x_{N-1}$ be an N-point sequence and let

$$x[n] = \begin{cases} x_n & \text{for } n = 0, \ldots, N-1 \\ 0 & \text{elsewhere} \end{cases}$$

Let $X(\omega)$ be the discrete-time Fourier transform of $x[n]$. The *discrete Fourier transform* (DFT) of x_n is defined to be the N-point sequence

$$\begin{aligned} X_k &= X(k\omega_s/N) = \sum_{n=0}^{N-1} x[n] e^{-jnTk\omega_s/N} \\ &= \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}nk};\ k = 0, \ldots, N-1 \quad (12) \end{aligned}$$

The DFT is simply the set of N samples of $X(\omega)$ taken at frequencies spaced by ω_s/N in the Nyquist band. Notice that if k is allowed to take values outside the set $\{0, \ldots, N-1\}$, the value computed by (12) repeats with period N.

DFT Examples

Complex Exponential Sinusoid

$$x_n = e^{j(\ell \frac{\omega_s}{N})nT} = e^{j\frac{2\pi}{N}n\ell} \tag{13}$$

For $k = 0, \ldots, N-1$

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} e^{j\frac{2\pi}{N}n\ell} e^{-j\frac{2\pi}{N}nk} = \sum_{n=0}^{N-1} e^{j\frac{2\pi}{N}n(\ell-k)} \\ &= \frac{1-e^{j2\pi(\ell-k)}}{1-e^{j\frac{2\pi}{N}(\ell-k)}} = N\delta[k-\ell] \end{aligned} \tag{14}$$

Cosine Wave

$$\begin{aligned} x_n &= \cos\frac{2\pi}{N}n\ell = 0.5e^{j\frac{2\pi}{N}n\ell} + 0.5e^{-j\frac{2\pi}{N}n\ell} \\ &= 0.5e^{j\frac{2\pi}{N}n\ell} + 0.5e^{j\frac{2\pi}{N}n(N-\ell)} \end{aligned} \tag{15}$$

$$X_k = 0.5N\delta[k-\ell] + 0.5N\delta[k-(N-\ell)] \tag{16}$$

The Inverse Discrete Fourier Transform (IDFT)

The original N-point sequence can be determined by using the inverse discrete Fourier transform (IDFT) formula

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{j\frac{2\pi}{N}nk} \quad \text{for} \quad n = 0, 1, \ldots, N-1 \tag{17}$$

Computational Requirements

Direct computation of a DFT value for a single k using (12) requires $N-1$ complex additions and N complex multiplications ignoring the fact that for some k the exponentials are 1 or -1. Thus, direct computation of all N points requires $N(N-1)$ complex additions and N^2 complex multiplications.

The next slide shows how the computation can be reduced to be proportional to $N\log_2 N$ by cleverly breaking the DFT sum down into $\log_2 N$ stages.

The Fast Fourier Transform

The computational complexity can be reduced to the order of $N \log_2 N$ by algorithms known as *fast Fourier transforms* (FFT's) that compute the DFT indirectly. For example, with $N = 1024$ the FFT reduces the computational requirements by a factor of

$$\frac{N^2}{N \log_2 N} = 102.4$$

The improvement increases with N.

Decimation in Time FFT Algorithm

One FFT algorithm is called the *decimation-in-time* algorithm. A brief derivation is presented below for reference. To simplify the notation, let $W_N = e^{-j2\pi/N}$ so (12) becomes

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} \tag{18}$$

This algorithm assumes that N is a power of 2.

Decimation in Time FFT (cont. 1)

Splitting the sum into a sum over even n and one over odd n gives

$$\begin{aligned} X_k &= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_N^{(2n+1)k} \\ & \qquad \text{for} \;\; k = 0, 1, \ldots, N-1 \end{aligned} \tag{19}$$

Let the even numbered points be the $N/2$ point sequence

$$a_n = x_{2n} \;\; \text{for} \;\; n = 0, 1, \ldots, \frac{N}{2} - 1 \tag{20}$$

and the odd numbered points be the $N/2$ point sequence

$$b_n = x_{2n+1} \;\; \text{for} \;\; n = 0, 1, \ldots, \frac{N}{2} - 1 \tag{21}$$

Also observe that $W_N^2 = W_{N/2}$. Thus, (19) can be written as

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} a_n W_{N/2}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} b_n W_{N/2}^{nk}; \;\; k = 0, \ldots, N-1 \tag{22}$$

Decimation in Time FFT (cont. 2)

Let A_k and B_k be the $N/2$-point DFT's of a_n and b_n so that these DFT's have period $N/2$ and

$$X_k = A_k + W_N^k B_k \quad \text{for} \quad k = 0, 1, \ldots, N-1 \quad (23)$$

The next step results in the key equations for the decimation-in-time FFT. First observe that $W_N^{N/2} = -1$. Then, the previous equation can be separated into the two equations

$$X_k = A_k + W_N^k B_k \quad \text{for} \quad k = 0, 1, \ldots, \frac{N}{2} - 1 \quad (24)$$

$$X_{k+\frac{N}{2}} = A_k - W_N^k B_k \quad \text{for} \quad k = 0, 1, \ldots, \frac{N}{2} - 1 \quad (25)$$

Equations (24) and (25) show how to compute an N-point DFT by combining a pair of $N/2$-point DFT's. A flowgraph for this pair of equations is shown in Figure 2. This computation is called an *FFT butterfly.* A complete flowgraph for this first step with $N = 8$ is shown in Figure 3.

Decimation in Time FFT (cont. 3)

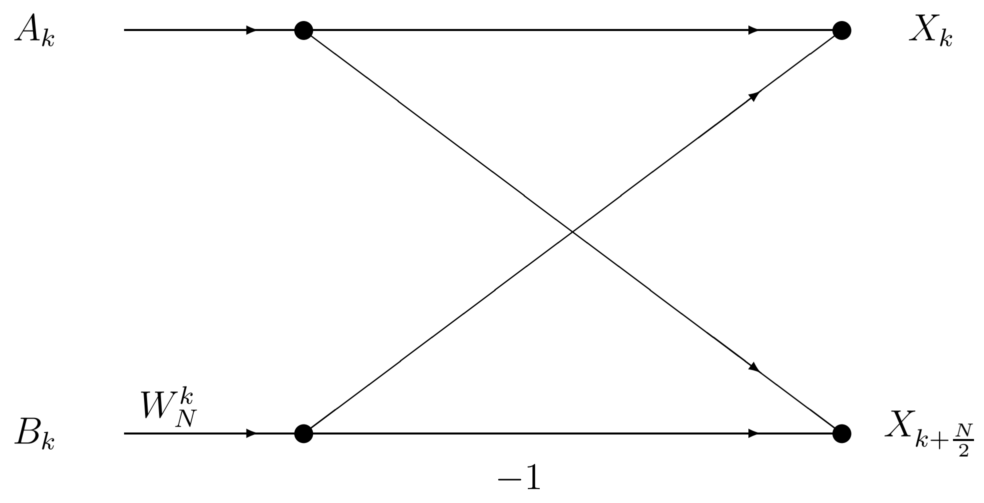


Figure 2: Flowgraph for an FFT Butterfly

If the $N/2$-point DFT's, A_k and B_k, are known, $N/2$ complex multiplications are required to compute $B_k W_N^k$, $N/2$ complex additions to compute $X_k = A_k + W_N^k B_k$, and $N/2$ complex subtractions to compute $X_{k+\frac{N}{2}} = A_k - W_N^k B_k$ for $k = 0, 1, \ldots, \frac{N}{2} - 1$. Addition and subtraction can be considered to be essentially the same. Thus, the entire N-point DFT can be computed with $N/2$ complex multiplications and N complex additions from the pair of $N/2$-point DFT's.

Decimation in Time FFT (cont. 4)

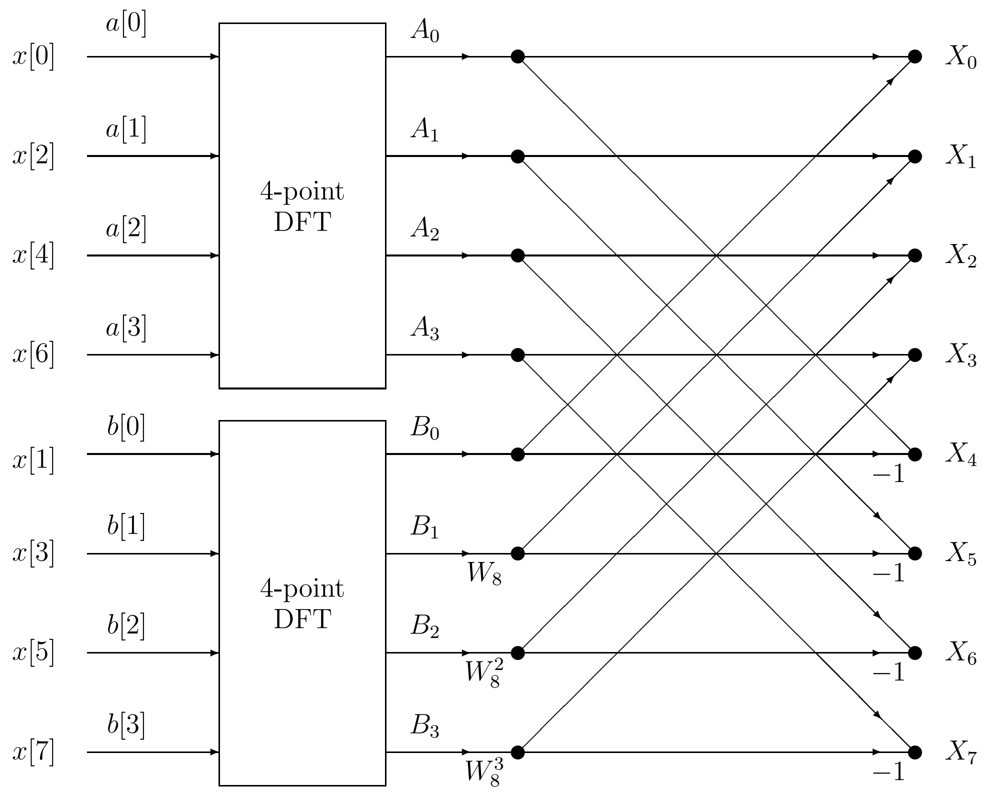


Figure 1: First Step in an 8-Point Decimation-in-Time FFT

Figure 3: First Step for 8-Point DIT FFT

The Second Step for the 8-Point DIT FFT

The same idea can be used to compute A_k and B_k from $N/4$-point DFT's. Computation of A_k would require $N/4$ complex multiplications and $N/2$ complex additions. Finding B_k would require the same amount of computation. Thus the total

Decimation in Time FFT (cont. 5)

amount of computation to compute both A_k and B_k is $N/2$ multiplications and N additions, which is the same as for computing X_k from A_k and B_k.

Total Amount of Computation

The reduction by 2 procedure can be repeated until one-point DFT's are reached. A one-point DFT of a point is just the point itself. This requires $\log_2(N)$ stages. Therefore, the entire amount of computation required to compute the N-point DFT is:

$$\tfrac{N}{2}\log_2(N) \text{ complex multiplications}$$

and

$$N\log_2(N) \text{ complex additions.}$$

A complete flowgraph for an 8-point decimation-in-time FFT is shown in Figure 4. Notice that the input points are arranged in a scrambled order while the output DFT is in its natural order.

Decimation in Time FFT (cont. 6)

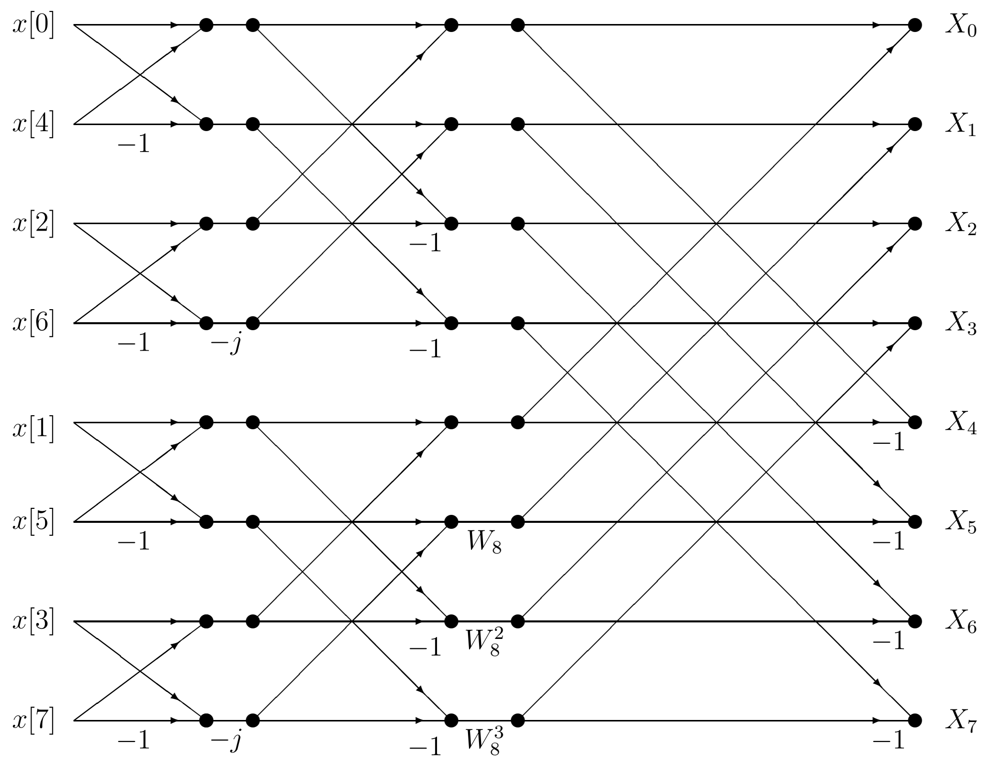


Figure 4: Complete DIT FFT Flowgraph for $N = 8$

Bit Reversed Input Ordering

It can be shown that the successive separation into even and odd numbered sequences puts the input sequence in bit-reversed order for any N that is a power of 2. The bit-reversed order is obtained by reversing the bits of the indexes for the original input array elements.

C Decimation in Time FFT Program

A C function for computing a complex, radix-2, decimation-in-time FFT is included below. You can find the sources in `c:\digfil\fft`. The program

- takes its complex input array in natural order and
- rearranges the input into bit reversed order.
- The output is in natural order.
- The computations are performed *in-place* with the output array written over the input array.
- The complex exponentials W^k are computed recursively.

The program could be made more efficient by precomputing and storing a cosine/sine table at angle increments needed for the largest N to be used and addressing the table appropriately for smaller N.

C FFT Program (cont. 1)

Header File Defining Complex Data Structure

```
/*  Header File complex.h    */
struct cmpx
  {
  float real;
  float imag;
  };
  typedef struct cmpx complex;
```

C Main Program to Test fft.c

```
/*************************/
/*  Program testfft.c   */
/*************************/
#include "complex.h"
/* Include all the DSK initialization headers here */
#include <math.h>
  #define M 10
  #define N 1024
  complex X[N]; /* Declare input array as external  */
extern void fft(complex *XX, int MM);

main()
  {
  int i;        /* loop index             */
  float pi = 3.141592653589;
```

C FFT Program (cont. 2)

```
/*****************************************/
/* Put DSK initialization code here.      */
/*****************************************/
/*   Initialize input array               */
/*Generate spectral lines at k = 5 and N-5 */
/ of height N/2.                          */

  for(i=0; i<N; i++)
    {
    (X[i]).real = cos(i*5*2.0*pi/N);
    (X[i]).imag = 0.0;
    }
/*-----------------------------------------*/
/* Perform FFT                              */

  fft(X,M);

/*  Display results on screen               */

  for(i=0; i<N; i++)
 printf("%4d%15.5f\t%15.5f\n",i,(X[i]).real,
        (X[i]).imag);
  }
```

C FFT Program (cont. 3)

C Function for Radix-2 Decimation-in-Time FFT

```
/*********************************************/
/*  Function fft(complex *X, int M)          */
/*                                           */
/*  This is an elementary, complex, radix 2, */
/*  decimation in time FFT.  The computations*/
/*  are performed "in place" and the output  */
/*   overwrites the input array.             */
/*********************************************/

#include "complex.h"
#include <math.h>

void fft(complex *X, int M)
    /* X is an array of N = 2**M complex points. */
{
complex temp1;/*temporary storage complex variable*/
complex W;  /* e**(-j 2 pi/ N)                     */
complex U;  /* Twiddle factor W**k                 */
int i,j,k;  /* loop indexes                        */
int id;     /* Index of lower point in butterfly  */
```

C FFT Program (cont. 4)

```
int N = 1 << M; /* Number of points for FFT       */
int N2 = N/2;
int L;       /* FFT stage                         */
int LE  /* Number of points in sub DFT at stage L,*/
        /* and offset to next DFT in stage        */
int LE1;    /* Number of butterflies in one DFT at*/
            /*  stage L. Also is offset to lower  */
            /*  point in butterfly at stage L     */
float pi = 3.1415926535897;

/*==============================================*/
/*  Rearrange input array in bit-reversed order   */
/*                                                */
/*    The index j is the bit reversed value of i. */
/*  Since 0 ->0 and N-1 ->N-1 under bit-reversal, */
/*  these two reversals are skipped.              */

  j = 0;
  for(i=1; i<(N-1); i++)
    {

/*++++++++++++++++++++++++++++++++++++++++++++++*/
/*  Increment bit-reversed counter for j by adding*/
/*  1 to msb and propagating carries from left to */
/*  right.                                        */
```

C FFT Program (cont. 5)

```
    k = N2; /* k is 1 in msb, 0 elsewhere          */
/*--------------------------------------------------*/
/*  Propagate carry from left to right              */

    while(k<=j) /* Propagate carry if bit is 1     */
      {
      j = j - k;/* Bit tested is 1, so clear it.  */
      k = k/2;  /* Set up 1 for next bit to right.*/
      }
    j = j+k;    /* Change 1st 0 from left to 1     */
/*--------------------------------------------------*/
/*  Swap samples at locations i and j if not        */
/*   previously swapped.                            */

    if(i<j) /* Test if previously swapped.          */
      {
      temp1.real = (X[j]).real;
      temp1.imag = (X[j]).imag;
      (X[j]).real = (X[i]).real;
      (X[j]).imag = (X[i]).imag;
      (X[i]).real = temp1.real;
      (X[i]).imag = temp1.imag;
      }
/*++++++++++++++++++++++++++++++++++++++++++++++++++*/
    }
```

C FFT Program (cont. 6)

```
/*==================================================*/
/*  Do M stages of butterflies                       */

  for(L=1; L<= M; L++)
    {
    LE = 1 << L;/*  LE = 2**L = points in sub DFT */
    LE1 = LE/2; /* Number butterflies in sub-DFT  */
    U.real = 1.0;
    U.imag = 0.0; /* U = 1 + j 0                   */
    W.real = cos(pi/LE1);
    W.imag = - sin(pi/LE1); /* W= e**(-j 2 pi/LE) */
/*--------------------------------------------------*/
/*   Do butterflies for L-th stage                   */

/* Do the LE1 butterflies per sub DFT*/
    for(j=0; j<LE1; j++)
      {
/*..................................................*/
/*    Compute butterflies that use same W**k         */

      for(i=j; i<N; i += LE)
        {
      /*id is index of lower point in butterfly */
        id = i + LE1;
```

C FFT Program (cont. 7)

```
        temp1.real = (X[id]).real*U.real
                      - (X[id]).imag*U.imag;
        temp1.imag = (X[id]).imag*U.real
                      + (X[id]).real*U.imag;

        (X[id]).real = (X[i]).real - temp1.real;
        (X[id]).imag = (X[i]).imag - temp1.imag;

        (X[i]).real  = (X[i]).real + temp1.real;
        (X[i]).imag  = (X[i]).imag + temp1.imag;
        }
/*................................................*/
/*   Recursively compute W**k as W*W**(k-1) = W*U  */

      temp1.real = U.real*W.real - U.imag*W.imag;
      U.imag = U.real*W.imag + U.imag*W.real;
      U.real = temp1.real;
/*................................................*/
      }
/*------------------------------------------------*/
    }
  return;
  }
```

Using the FFT to Estimate a Power Spectrum

One method for estimating power spectral densities is based on using a function called the *periodogram.* The periodogram of an N-point sequence $y[n]$ is defined to be

$$I_N(\omega) = \frac{1}{N}|Y(\omega)|^2 \tag{26}$$

where

$$Y(\omega) = \sum_{n=0}^{N-1} y[n]e^{-j\omega nT} \tag{27}$$

is the discrete-time Fourier transform of $y[n]$. It can be shown that the inverse transform of the periodogram is the *sample autocorrelation function*

$$R(n) = \begin{cases} \dfrac{1}{N}\displaystyle\sum_{k=0}^{N-1} y[n+k]\bar{y}[k] & \text{for } \ |n| \leq N-1 \\ 0 & \text{elsewhere} \end{cases} \tag{28}$$

Justification for Using Periodogram

The variable, n, in the autocorrelation function is called the *lag*. For zero lag

$$R(0) = \frac{1}{N}\sum_{k=0}^{N-1} |y[k]|^2 = \frac{1}{\omega_s}\int_{-\omega_s/2}^{\omega_s/2} I_N(\omega)\, d\omega \quad (29)$$

is the average power in the sequence. This equation provides some justification for interpreting the periodogram as a function that shows how the power is distributed in the frequency domain.

Statistical Fluctuations of the Periodogram

It is natural to assume that as N increase, the periodogram becomes a better estimate of the power spectral density for a stationary random process. **This is not true.** Actually, the mean of the periodogram converges to the true spectral density but its variance remains large. As N increases, the periodogram tends to oscillate more and more rapidly.

Averaging Periodograms

A solution to this problem is to average the periodograms of different N-point sections of the observed data sequence. Let $x[n]$ be an observed data sequence with duration $M = LN$ and form the L windowed N-point data sections

$$y_k[n] = \begin{cases} h[n]x[n+kN] & \text{for } n = 0, \ldots, N-1 \\ 0 & \text{elsewhere} \end{cases} \tag{30}$$

for $k = 0, 1, \ldots, L-1$, where h[n] is a desired data window function. Designate the periodogram formed from the k-th widowed section by $I_{N,k}(\omega)$. Then the desired power spectral density estimator is

$$\hat{S}(\omega) = \frac{1}{L} \sum_{k=0}^{L-1} I_{N,k}(\omega) \tag{31}$$

When the data sections are statistically independent, averaging L sections reduces the variance by a factor of L. Additional gains can be achieved by overlapping the sections to some degree.

Computing the Sum of the Periodograms of Two Real Sequences as the FFT of One Complex Sequence

The periodograms can be computed at the uniformly spaced frequencies $\{k\omega_s/N; k = 0, 1, \ldots, N-1\}$ by using an N-point FFT. When the observed data sequence is real and the FFT program is designed to accept complex inputs, the computation time can be reduced by almost a factor of two by using the following identity. Let $a[n]$ and $b[n]$ be two real N-point sequences and form the complex sequence $c[n] = a[n] + jb[n]$. Then

$$|A_k|^2 + |B_k|^2 = \frac{|C_k|^2 + |C_{N-k}|^2}{2} \tag{32}$$

Thus, the sum of the periodograms of the two real sequences can be computed from the FFT of the single complex sequence.

Chapter 4, Experiment 1 FFT Experiments

You can use the C FFT function starting on Slide 4-21 for these experiments. The files `testfft.c`, `complex.h`, and `fft.c` can be found in `C:\DIGFIL\FFT`.

To test and extend your understanding of FFT's, perform the following tasks:

1. Let the sampling rate be 16 kHz and the sequence length be $N = 1024$ points. Theoretically find the DFT of the sequence

   $$x_n = \sin(2\pi \times 2000 \times n/16000) \text{ for } n = 0, \ldots, 1023$$

2. Generate a program for the TMS320C6713 to compute the FFT of the sequence x_n defined above by doing the following:
   (a) First copy the linker command file, `dsk6713_RTDX.cmd` from `C:\c6713dsk`

FFT Experiments (cont. 1)

to your project directory and increase `-stack` from 0x1000 to 0x2000 so the stack does not overflow. Be sure to use this modified command file in your project.

(b) Fill an N-point complex array with the real and imaginary parts of the sequence x_n defined above.

(c) Compute the FFT of x_n by calling the function `fft()`.

(d) Fill a separate N-point real array with the squared complex magnitudes of the FFT values.

Note: Make your complex 1024-point array `X[N]` an external (global) array. This causes `X` to be stored in a fixed memory location. If `X` is a local variable, the TI compiler generates code that copies the entire array onto the stack when `fft(X,M)` is called. This can cause the stack to overflow if it is not made large enough in the linker command file and it also takes a

FFT Experiments (cont. 2)

significant amount of time. In addition, when you make your spectrum analyzer your interrupt service routine declared with the "interrupt" keyword cannot have arguments and you must pass `X` to it as an external variable.

3. Use Code Composer Studio to:
   (a) Check your answer by displaying the complex FFT array in a Code Composer Studio watch window. Alternatively, you can use the C function `printf()` to write the FFT values to the CCS display window.
   (b) Use the C function `fprintf()` to write the array to a PC disk file. Compare the disk file with the theoretical result to further check your program.
   (c) Plot the squared magnitude array using the CCS graphing capability.

FFT Experiments (cont. 3)

(d) Find the time required to compute the FFT by using the CCS profiling capability.

4. Repeat steps 1, 2, and 3 except multiply the input sequence by a Hamming window. When checking your results be sure to examine the FFT in the vicinity of 2000 and 14000 Hz.

5. Change the input sequence to:

$$x_n = \sin\left[2\pi\left(2000 + 0.5\,\frac{16000}{1024}\right)\frac{n}{16000}\right]$$

Repeat steps 1 through 4 for this new signal. Explain the results.

Chapter 4, Experiment 2
Power Spectrum Estimation

Now you will make an elementary spectrum analyzer. The DSK will be used to collect blocks of $N = 1024$ samples taken at a 16 kHz rate. The DSP will compute and average the periodograms. The results will be displayed on the PC by using Code Composer Studio's graphing capabilities.

Suggestions on How to Structure the Spectrum Estimation Program

The power spectral density estimates will be based on periodograms of 1024-point blocks of input samples taken at a 16 kHz rate. The technique described on Slide 4-29 to compute the sum of pairs of periodograms should be used to efficiently utilize the 1024-point FFT. The following list suggests a method of data collection for your program and the tasks it should perform.

Making a Spectrum Analyzer

1. Initialize the DSK as usual.
2. Set up a 1024-word array that contains the floating-point samples of the Hamming window.
3. Set up an external 513-word array of floats, `spectrum[]`, for the spectrum estimates at frequencies from 0 to 8000 Hz ($k = 0, \ldots, 512$). (Values from 8000 to 16000 Hz are the mirror images of the ones from 0 to 8000 Hz.)
4. **Ping-Pong Buffers:** Use the technique of *ping-pong buffers.* Set up two external complex floating-point arrays named `ping[]` and `pong[]` each of size 1024 complex words. (See the header file `complex.h`.) One array will be used to collect new samples from the ADC using RRDY interrupts from the McBSP1 receiver while an FFT and periodogram averaging are being performed on the other.

Ping-Pong Buffers (cont.)

The samples should be read, converted to floating-point words, and stored in the `ping` or `pong` array in an interrupt service routine. The first 1024 samples should be stored in the real part of the array and the next 1024 samples in the imaginary part of the array.

5. While one array is being filled with new samples through interrupts:

   (1) **Hamming Windowing:** Multiply the real and imaginary parts of the previously filled array by the Hamming window and leave the results in the same array.

   (2) Perform a complex 1024-point FFT on the windowed array.

   (3) Use equation (32) on Slide 4-29 to compute the sum of the squared magnitudes of the FFT's of the real and imaginary parts of the array. Remember that when x_n is real,

Experiment 4.2 Spectrum Estimation (cont. 1)

$X_k = \bar{X}_{N-k}$ so that the second half of the FFT is totally redundant. Thus, it is only necessary to compute the sum of the squared magnitudes for $n = 0, \ldots, 512$. As each value is computed, add it to the corresponding element of `spectrum[]`.

(4) Once the FFT and additions to `spectrum[]` have been completed, wait for the array collecting new samples to be filled. You'll have to devise a way that the main function can determine when arrays are filled. Then switch arrays, allow the array just processed to begin collecting new samples, and begin processing the array that was newly filled.

Spectrum Estimation (cont. 2)

(5) Continue to accumulate FFT squared magnitudes in `spectrum[]` until $L = 8$ have been added and then divide the elements by 8×1024 or whatever is appropriate depending on any previous normalizations. You can experiment later with using larger or smaller values for L.

(6) Add a dummy line to your main function after averaging $L = 8$ periodograms, that is, when one spectrum estimate has been completed, and before going back to begin a new spectrum estimate, as a place for a Code Composer break point for graph updating. For example, the line might be

```
dummy = ping[0];
```

(7) Your program should then loop back, clear `spectrum[]`, and compute a new averaged periodogram, etc.

Spectrum Estimation (cont. 3)

6. Compile your program using -o3 optimization.
   Important Note: Using the "volatile" Declaration to Stop the Optimizer from Breaking Things
   Consider the C source code lines:

```
int insample;
insample = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
```

   The optimizer looks at `MCBSP_read()` and sees a function with a constant argument. It thinks the returned value of this function will never change. It does not know the serial port DRR contents can be different each time the function is executed. Therefore, it creates code to set "`insample`" just once and never do it again. The declaration "volatile" informs the optimizer that "`insample`" can actually change and should be updated every time it is encountered. So, to make sure the optimizer does the correct thing, use the code:

```
    volatile int insample;
    insample = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
```

WARNING: Remember to increase `-stack` to 0x1000 in the linker command file. Otherwise the stack might overflow and overwrite some variables causing strange answers.

Another Method of Averaging Periodograms

Periodograms can be averaged by using a one-pole IIR lowpass filter. A filter of this type has the transfer function

$$H(z) = (1-\alpha)/(1-\alpha z^{-1}) \text{ where } 0 < \alpha < 1$$

The closer α is to 1 the more lowpass the filter is, and the slower its output changes. The impulse response of the filter is

$$h(n) = (1-\alpha)\alpha^n u(n)$$

and it is sometimes called an *exponential averager*.

Let the current averaged spectrum estimate at DFT slot k be $S_k(n)$ and the current periodogram at slot k be $I_k(n)$. Then the exponential averager output is computed by the formula:

$$S_k(n) = (1-\alpha)I_k(n) + \alpha S_k(n-1) \tag{33}$$

This computation can be performed each time the sum of a new pair of periodograms is computed.

Testing Your Spectrum Analyzer

A. Initial Testing Using a Known Synthesized Input

As a first test of your spectrum analyzer, replace the input samples in your interrupt service routine by samples of a sine wave generated in your program at one of the FFT bin frequencies . For example, for bin $k = 100$ your synthesized input could be

$$10000.0 * \cos(n \times 100 \times 2\pi/1024)$$

Initial Testing (cont.)

The scale factor 10000.0 is to model the dynamic range of the integer samples arriving from the ADC. In your program, the angle inside the cosine function should be generated recursively by adding the constant $100 \times 2\pi/1000$ to the old angle each time the interrupt routine is entered. Also, limit the size of the angle by subtracting 2π when it exceeds 2π as you did in Chapter 2. Perform the following two exercises:

1. Temporarily replace the Hamming window by a rectangular window, that is, all 1's, and fill the imaginary parts of the sample arrays with all 0's. Prove that the FFT values for $n = 100$ and $1024 - 100 = 924$ should be $10000 \times 1024/2$ and should be 0 for all other k. Determine the theoretical values your spectrum analyzer should produce. Check that your analyzer is giving the theoretical answer and correct your program if it isn't. Try cosines at a few other bin frequencies.

2. Re-enable the Hamming window. Theoretically determine what the analyzer output should be. Check that your analyzer is giving the correct results.

Displaying the Spectrum Using Code Composer Studio

To display the spectrum estimates using Code Composer Studio's break point and graphing capabilities:

1. Load your program by clicking on the green bug.

2. Scroll down in the source code window to the line "`dummy = ping[0];`" and click on it to put the cursor there.

3. Right click on this dummy line and from the top menu item click on "Breakpoint."

Displaying the Spectrum Using CCS (cont.)

4. The "Breakpoints" window will appear. Right click on the line with your breakpoint, select "Breakpoint Properties ..." and set the "Action" to "Refresh All Windows."

5. Click on "Tools." Select "Graph" and then "Single Time".

6. On the graph properties menu enter:
   Acquisition Buffer Size: 513
   DSP Data Type: 32 bit floating point
   Start Address: spectrum
   Data Plot Style: Bar
   Display Data Size: 513
   Time Display Unit: sample

7. Click "OK" and the graph should appear.

8. Click on the "resume" icon. The program should run and the graph should get updated with each new spectrum estimate.

Testing with External Inputs

Now attach the signal generator output to the DSK line input and an oscilloscope. Set your program to use the actual input samples rather than the synthesized cosine wave. Perform the following experiments:

1. Set the signal generator to generate a 2 kHz sinewave and observe the output of your spectrum analyzer using CCS as described above. Compute which FFT bin, that is, value of k, corresponds to 2 kHz and check that your spectrum analyzer display is correct.
2. Derive a formula for the Fourier coefficients of a non-symmetric square-wave with one period given by the following equation.

   $$x(t) = \begin{cases} A/2 & \text{for } |t| < \tau/2 \\ -A/2 & \text{for } \tau/2 \leq |t| \leq T_0/2 \end{cases}$$

   The duty cycle for the square-wave is defined to be τ/T_0.

Testing with External Inputs (cont. 1)

(a) Set the signal generator to generate a 200 Hz square-wave with a 50% duty cycle and compare the measured and theoretical spectra.

(b) Set the duty cycle to 20% and compare the measured and theoretical spectra.

3. Test your spectrum analyzer with amplitude modulated signals. These have the form

$$A_c[1 + m(t)] \cos 2\pi f_c t$$

See Chapter 5 for a detailed discussion of AM. The waveform $m(t)$ is called the modulating signal and f_c the carrier frequency. The function $|A_c[1 + m(t)]|$ is called the signal envelope.

In particular, let $f_c = 4$ kHz and do the following:

Testing with External Inputs (cont.)

(a) Let $m(t) = A_m \cos 2\pi 500t$ and derive the theoretical spectrum for $x(t)$. A_m is called the modulation index for the AM signal. Set the function generator to generate a signal of this type with a modulation index of 100%. Compare the theoretical and measured spectra.

(b) Repeat the previous item but change $m(t)$ to a 200 Hz square-wave with a 20% duty cycle.

4. Experiment with the FM signals of the signal generator. FM is discussed in detail in Chapter 8. In particular, see Equation 8.12 of Section 8.1.2 for the spectrum. It is much more complex than the AM spectrum. Change the modulation index and observe how the carrier component can disappear and how the spectrum spreads out as the modulation index is increased.

Testing with an Exponential Averager

5. If time permits and for extra credit, modify your spectrum estimation program to use exponential periodogram averaging instead of the arithmetic average of L pairs of periodograms. Experiment with different values of α and see how your display responds when the input signal is changed. For example, you could change the carrier frequency of an AM or FM signal, or the modulation index of an FM signal.