2.161 Signal Processing: Continuous and Discrete
Fall 2008

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF MECHANICAL ENGINEERING
2.161 Signal Processing - Continuous and Discrete

The Fast Fourier Transform [1]

Introduction:

Although the DFT was known for many decades, its utility was severely limited because of the computational burden. The calculation of the DFT of an input sequence of an N length sequence $\{f_n\}$

$$F_m = \sum_{n=0}^{N-1} f_n e^{-j\frac{2\pi mn}{N}}, \qquad m = 0, \ldots, N-1 \tag{1}$$

requires N complex multiplications to compute each on the N values, F_m, for a total of N^2 multiplications. Early digital computers had neither fixed-point nor floating-point hardware multipliers, and multiplication was performed by binary shift-and-add software algorithms. Multiplication was therefore a computationally "expensive" and time consuming operation, rendering machine computation of the DFT impractical for common usage.

In 1965 Cooley and Tukey [1] of IBM labs published a ground-breaking paper that presented a computational algorithm for the DFT that required just a small fraction of the complex multiplications in Eq. (1). The *Fast Fourier Transform* (FFT) has revolutionized digital signal processing by allowing practical fast frequency domain implementation of processing algorithms. (It is interesting to note in retrospect that the techniques used in the FFT can be found all the way back to Gauss, although the significance was not recognized until Cooley and Tukey).

There are many variations on the FFT algorithm. This handout examines just one of them: the *Radix-2 FFT with decimation in time*, which is probably the most commonly used FFT. The term *Radix-2* refers to the limitation that the sample length N must be an integer power of 2, while *decimation in time* means that the sequence $\{f_n\}$ must be re-ordered before applying the algorithm.

The Radix-2 FFT with Decimation in Time:

We start by writing the DFT of Eq. (1) as

$$F_m = \sum_{n=0}^{N-1} f_n e^{-j\frac{2\pi mn}{N}} = \sum_{n=0}^{N-1} f_n W_N^{mn}, \qquad m = 0, \ldots, N-1 \tag{2}$$

where $W_N = e^{-j2\pi/N}$. We also note the following periodic and symmetry properties of W_N

- $W_N^{k(N-n)} = W_N^{-kn} = \overline{W_N^{kn}}$ (complex conjugate symmetry),
- $W_N^{kn} = W_N^{k(n+N)} = W_N^{n(k+N)}$ (periodicity in n and k),
- $W_N^n = -W_N^{n-N/2}$ for $n \geq N/2$.

The FFT recognizes that these properties render many of the N^2 complex multiplications in Eq. (1) redundant.

We start by assuming that the input sequence length N is even. We then ask whether any computational efficiency might be gained from splitting the calculation of $\{F_m\}$ into two sub-computations, each of length $N/2$, involving the even samples, $\{f_{2n}\}$, and odd samples $\{f_{2n+1}\}$ for

[1] D. Rowell October 4, 2007

$n = 0 \ldots N/2 - 1$:

$$\begin{aligned} F_m &= \sum_{n=0}^{P-1} f_{2n} W_N^{2mn} + \sum_{n=0}^{P-1} f_{2n+1} W_N^{m(2n+1)} \\ &= \sum_{n=0}^{P-1} f_{2n} W_N^{2mn} + W_N^m \sum_{n=0}^{P-1} f_{2n+1} W_N^{2mn} \\ &= \sum_{n=0}^{P-1} f_{2n} W_P^{mn} + W_N^m \sum_{n=0}^{P-1} f_{2n+1} W_P^{mn} \\ &= A_m + W_N^m B_m, \qquad m = 0, \ldots, N-1. \end{aligned} \tag{3}$$

where $P = N/2$, $\{A_m\}$ is a DFT of length $N/2$, based on the even sample points, and similarly $\{B_m\}$ is a DFT of length $N/2$ based on the odd sample points of $\{f_n\}$. We also note from the properties of the DFT that both $\{A_m\}$ and $\{B_m\}$ are periodic with period $N/2$, that is

$$A_{m+N/2} = A_m, \qquad \text{and} \qquad B_{m+N/2} = B_m$$

so that

$$F_m = A_m + W_N^m B_m \qquad \text{for } m = 0 \ldots (N/2 - 1), \text{ and} \tag{4}$$

$$F_m = A_{m-N/2} - W_N^{m-N/2} B_{m-N/2} \qquad \text{for } m = N/2 \ldots (N-1) \tag{5}$$

Equations (4) and (5) show that a DFT of length N may be synthesized by combining two shorter DFTs from an even/odd decomposition of the original data set. For example, if $N = 8$, F_3 and F_7 are simply related:

$$\begin{aligned} F_3 &= A_3 + W_8^3 B_3 \\ F_7 &= A_7 + W_8^7 B_7 = A_3 - W_8^3 B_3 \end{aligned}$$

so that $N/2$ multiplications are required to combine the two sets. Each of the two shorter DFTs requires $(N/2)^2$ complex multiplications, therfore the total required is $N^2/2 + N/2 < N^2$, for $N > 2$, indicating a computational saving.

A modified discrete form of Mason's signal-flow graph is commonly used to display the algorithmic structure of Eq. (3). Figure 1 shows the signal-flow graph - consisting of a network of nodes connected by line segments. The algorithm works from left to right, and each right-hand node is assigned a value that is the weighted sum of the connected left-hand nodes, where the indicated weight n is the exponent of W_N. If no weight is indicated, it is assumed to be unity (or equivalent to W_N^0). Thus the output of the step shown in Fig. 1 is $c = a + W_N^n b$.

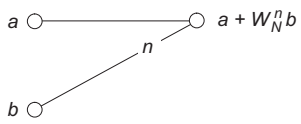

Figure 1: Signal-flow graph notation.

With this notation the combining of the two length $N/2$ DFTs is illustrated in Fig. 2 for N=8. Each right-hand node is one of the F_m, formed by the appropriate combination of A_m and B_m as developed in Eqs. (3), (4) and (5).

If N is divisible by 4, the process may be repeated, and each length $N/2$ DFT may be formed by decimating the two $N/2$ sequences into even and odd components, forming the length $N/4$ DFTs,

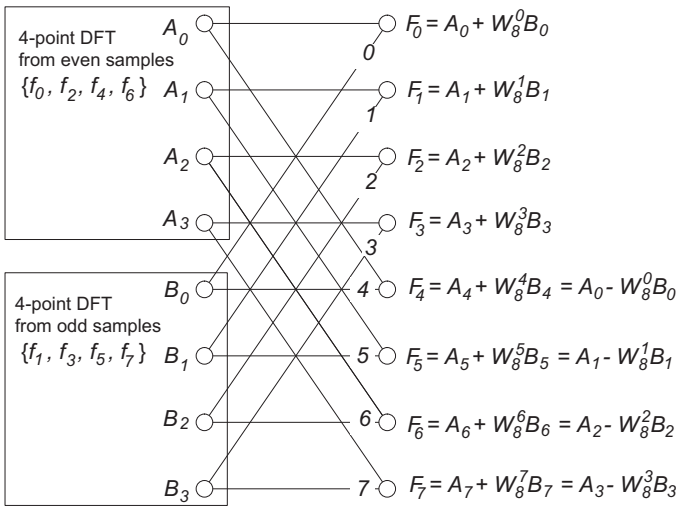

Figure 2: Signal-flow graph representation of combining two length-4 DFT sequences, derived from even and odd sample sets, into a length-8 sequence as defined in Eq. (3).

and combining these back into a length $N/2$ DFT, as is shown for $N = 8$ in Fig. 3. Notice that all weights in the figure are expressed by convention as exponents of W_8. In general, if the length of the data sequence is an integer power of 2, that is $N = 2^q$ for integer q, the DFT sequence $\{F_m\}$ may be formed by adding additional columns to the left and halving the length of the DFT at each step, until the length is two. For example if $N = 256 = 2^8$ a total of seven column operations would be required.

The final step is to evaluate the N/2 length-2 DFTs. Each one may be written

$$\begin{aligned} F_0 &= f_0 + W_2^0 f_1 = f_0 + f_1 \\ F_1 &= f_0 + W_2^1 f_1 = f_0 - f_1, \end{aligned}$$

which is simply the sum and difference of the two sample points. No complex multiplications are necessary. The 2-point DFT is shown in signal-flow graph form in Fig. 4, and is known as the *FFT butterfly*.

The complete FFT algorithm for $N = 8$ is formed by adding a column of 2-point DFTs to the left of Fig. 3, as shown in Fig. 5. We note that if $N = 2^q$, there will be $q = \log_2(N)$ columns in the signal-flow graph, and after the sum and difference to form the 2-point DFTs there will be $\log_2(N) - 1$ column operations, each involving $N/2$ complex multiplications, giving a total of $N/2\,(\log_2(N) - 1) \leq N^2$. We will address the issue of computational savings in more detail later.

Input Bit-Reversal:

Notice that the algorithm of Fig. 5 requires that the input sequence $\{f_n\}$ be re-ordered in the left-hand column to accomplish the even-odd decomposition at each step. For this reason this form of the FFT is known as the *FFT with decimation in time through input bit reversal*. The term *input bit reversal* refers to a simple algorithm to determine the position k of the sample f_n in the re-ordered sequence:

1. Express the index n as a N-bit binary number.
2. Reverse the order of the binary digits (bits) in the binary number.
3. Translate the bit-reversed number back into decimal, to create the position in the sequence k.

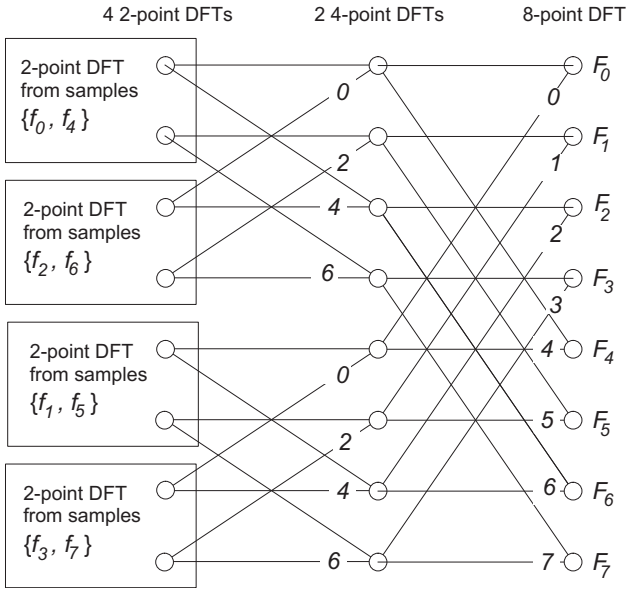

Figure 3: Two steps to combining four length-2 DFT sequences into a length-8 sequence.

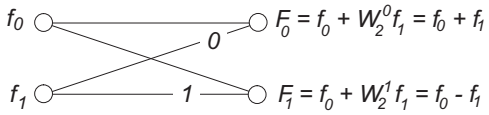

Figure 4: The 2-point DFT butterfly.

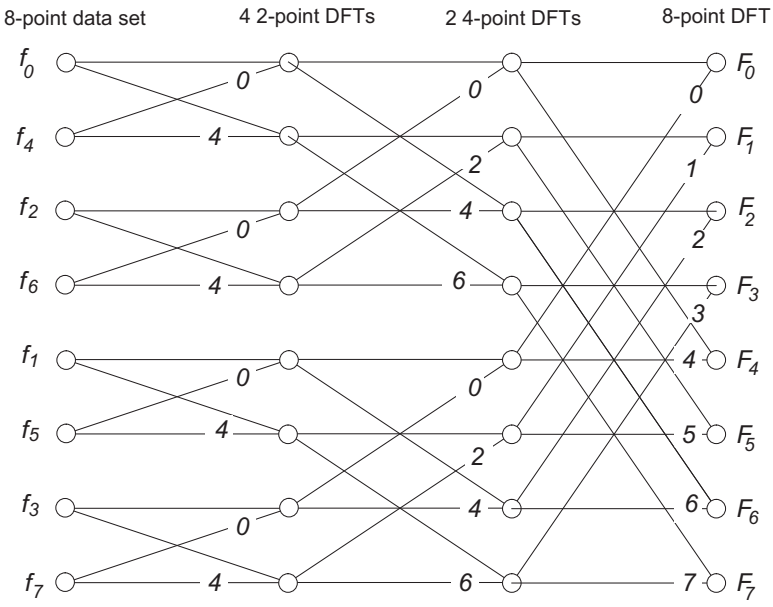

Figure 5: Complete radix-2 with decimation in time (input bit reversal) FFT algorithm for $N = 8$.

For example, the re-ordered position of f_{37} in a data sequence of length $N = 256 = 2^8$ is found from

$$37_{10} = 00100101_2 \quad \overset{\text{bit reversal}}{\longrightarrow} \quad 10100100_2 = 164_{10}$$

so that f_{37} would be positioned at $k = 164$ in the decimated input sequence. Table 1 shows the complete re-ordering process for $N = 8$.

Input position n:	0	1	2	3	4	5	6	7
	$(000)_2$	$(001)_2$	$(010)_2$	$(011)_2$	$(100)_2$	$(101)_2$	$(110)_2$	$(111)_2$
Bit reversal	↓	↓	↓	↓	↓	↓	↓	↓
	$(000)_2$	$(100)_2$	$(010)_2$	$(110)_2$	$(001)_2$	$(101)_2$	$(011)_2$	$(111)_2$
Modified position k:	0	4	2	6	1	5	3	7

Table 1: Determination of input bit-reversed sequence order for $N = 8$.

A sample FFT routine with input bit-reversal, written as a MATLAB script `fftx()`, is presented in the Appendix. This is not intended as a substitute for MATLAB's built-in FFT functions, and is intended only as a tutorial example.

As noted in the introduction, the algorithmic structure presented here is just one of several FFT structures. Some of these take the time domain data in the correct order and produce the DFT values in bit-reversed order. Other algorithms are based on frequency decomposition, and matrix factoring operations.

The Inverse Fast Fourier Transform (IFFT):
The inverse FFT is defined as

$$f_n = \frac{1}{N} \sum_{m=0}^{N-1} F_m e^{j\frac{2\pi mn}{N}}, \qquad n = 0, \ldots, N-1 \tag{6}$$

While the IFFT can be implemented in the same manner as the FFT described above, it is possible to use a forward FFT routine to compute the IFFT as follows: Since the conjugate of a product is the product of the conjugates, if we take the complex conjugate of both sides we have

$$\overline{f}_n = \frac{1}{N} \sum_{m=0}^{N-1} \overline{F}_m e^{-j\frac{2\pi mn}{N}}.$$

The right-hand side is recognized as the DFT of $\overline{F}_m$ and can be computed using a forward FFT, such as described above. The complete IDFT may therefore be computed by conjugating the output, that is

$$f_n = \frac{1}{N} \overline{\left[\sum_{m=0}^{N-1} \overline{F}_m e^{-j\frac{2\pi mn}{N}} \right]}, \qquad n = 0, \ldots, N-1 \tag{7}$$

The steps are:

1. Conjugate the data set $\{F_m\}$.
2. Compute the forward FFT.
3. Conjugate the result and divide by N.

A tutorial MATLAB inverse FFT routine, `ifftx()`, is presented in the Appendix.

Computational Savings of the FFT:
As expressed above the computational requirements (in terms of complex multiplications) is $\mathrm{M}_{\mathrm{FFT}} = (N/2)\log_2(N)$ if the initial 2-point DFTs are implemented with exponentials. The number of complex multiplications for the direct DFT computation is $\mathrm{M}_{\mathrm{DFT}} = N^2$) We can therefore define a speed improvement factor $\mathrm{M}_{\mathrm{FFT}}/\mathrm{M}_{\mathrm{DFT}}$ as is shown in Table 2.

It should be realized,however, that in practice these dramatic improvements shown in Table 2 will not be realized because of the significant improvement in arithmetic processing speeds, particularly with regard to multiplications.

N	$\mathrm{M}_{\mathrm{DFT}}$	$\mathrm{M}_{\mathrm{FFT}}$	$\mathrm{M}_{\mathrm{FFT}}/\mathrm{M}_{\mathrm{DFT}}$
4	16	4	0.25
8	64	12	0.188
16	256	32	0.125
32	1,024	80	0.0781
64	4,096	192	0.0469
128	16,384	448	0.0273
256	65,536	1024	0.0156
512	262,144	2,304	0.00879
1024	1,048,576	5,120	0.00488
2048	4,194,304	11,264	0.00268
4096	16,777,216	24,576	0.00146

Table 2: Computational speed improvement from the use of the FFT

.

Efficient Use of the FFT:
The computational efficiency of an FFT function may be increased by several methods, depending on the application:

- If the algorithm is to be applied to a fixed length data set repetitively, the table of complex exponentials W_N^n (sines and cosines) may be pre-computed and stored.
- If it is known that $\{f_n\}$ is real, then $\{F_m\}$ is conjugate symmetric about its mid-point, so that only $N/2$ values have to be computed.
- It is possible to compute the FFT of a two data sequences of length N in one length N FFT by creating a complex sequence $\{f_n\}$ with one data set in the real part and the other in the imaginary part.

  Let $\{c_n\}$ and $\{d_n\}$ be two real data sets of length N, and form them into a single complex sequence

  $$f_n = c_n + jd_n \qquad 0 \le n \le N-1.$$

  Then since the DFT is a linear operation $F_m = C_m + jD_m$. The two sequences can be expressed in terms of f_n as follows:

  $$\begin{aligned} c_n &= \frac{1}{2}\left(f_n + \overline{f}_n\right) \\ d_n &= \frac{1}{2j}\left(f_n - \overline{f}_n\right) \end{aligned}$$

  and therefore

  $$\begin{aligned} C_m &= \frac{1}{2}\left(\mathrm{DFT}\left\{f_n\right\} + \mathrm{DFT}\left\{\overline{f}_n\right\}\right) \\ D_m &= \frac{1}{2j}\left(\mathrm{DFT}\left\{f_n\right\} - \mathrm{DFT}\left\{\overline{f}_n\right\}\right) \end{aligned}$$

but the DFT of $\left\{\overline{f}_n\right\} = \{F_{N-m}\}$ so that the two DFTs may be recovered from $\{F_m\}$:

$$\begin{aligned} C_m &= \frac{1}{2}\left(F_m + \overline{F}_{N-m}\right) \\ D_m &= \frac{1}{2j}\left(F_m - \overline{F}_{N-m}\right) \end{aligned}$$

Because this method requires real data, it will not generally be applicable to computing an inverse FFT.

- Similarly, it is possible to compute the DFT of a length $2N$ data sequence by splitting into two length N sequences and placing each in the real and imaginary parts of a length N complex sequence.

  Perform an even/odd decomposition into two N-point sequences:

$$\begin{aligned} c_n &= f_{2n} \qquad \text{even samples,} \\ d_n &= f_{2n+1} \qquad \text{odd samples.} \end{aligned}$$

  Form the complex N-point sequence $\{f_n\} = \{c_n\} + j\{d_n\}$, and use the method described above, so that

$$\begin{aligned} C_m &= \frac{1}{2}\left(F_m + \overline{F}_{N-m}\right) \\ D_m &= \frac{1}{2j}\left(F_m - \overline{F}_{N-m}\right). \end{aligned}$$

  Finally, combine the two N-point DFTs into a single $2N$-point DFT using Eqs. (3),

$$\begin{aligned} F_m &= C_m + W_{2N}^k D_m \qquad m = 0, 1, \ldots, N-1 \\ F_{m+N} &= C_m - W_{2N}^k D_m \qquad m = 0, 1, \ldots, N-1 \end{aligned}$$

  Again, this method will generally not be applicable to inverse FFT calculations because the F_m are usually complex.

MATLAB FFT Routines:

MATLAB contains several DFT routines based on the FFT, including multidimensional FFTs. These routines are not restricted to radix-2 lengths (and will even handle prime lengths) by adjusting the internal algorithm. In particular, the following four routines are useful for one dimensional transforms:

`fft(x)` is the discrete Fourier transform (DFT) of vector X. For matrices, the FFT operation is applied to each column. For N-D arrays, the FFT operation operates on the first non-singleton dimension.
`f(x,n)` is the N-point FFT, padded with zeros if x has less than N points and truncated if it has more.
`fft(X,[],DIM)` or `fft(X,N,DIM)` applies the FFT operation across the dimension DIM.

`ifft(x)` is the inverse discrete Fourier transform of X. `ifft(X,N)` is the N-point inverse transform. `ifft(X,[],DIM)` or `ifft(X,N,DIM)` is the inverse discrete Fourier transform of X across the dimension DIM.

`fftshift(X)` is useful for visualizing the Fourier transform with the zero-frequency component in the middle of the spectrum. The routine shifts the zero-frequency component to center of

spectrum. For vectors, `fftshift(X)` swaps the left and right halves of X. For matrices, `fftshift(X)` swaps the first and third quadrants and the second and fourth quadrants. For N-dimensional arrays, `fftshiftT(X)` swaps "half-spaces" of X along each dimension.
`fftshift(X,DIM)` applies the `fftshift` operation along the dimension DIM.

`ifftshift(X)` undoes the effects of `fftshift`. For vectors, `ifftshift(X)` swaps the left and right halves of X. For matrices, `ifftshift(X)` swaps the first and third quadrants and the second and fourth quadrants. For N-dimensional arrays, `ifftshift(X)` swaps "half-spaces" of X along each dimension.
`ifftshift(X,DIM)` applies the `ifftshift` operation along the dimension DIM.

See MATLaB's `help` and `doc` facilities for more information and descriptions of multidimensional transforms.

References:

[**1**] Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Computation of the Complex Fourier Series,"Mathematics of Computation, Vol. 19, April 1965, pp. 297-301.

Appendix: MATLAB Demonstration FFT Functions:

```
%--------------------------------------------------------------
%   *** 2.161 Signal Processing - Continuous and Discrete  ***
%
%   fftx -  Tutorial FFT routine to demonstrate the Radix-2 FFT with
%              decimation in time.
%
%   Fout = fftx(f, k)
%   Arguments:
%      f  - time domain data set ()real or complex
%      k  - size of the data set k = log_2(N)
%
%   Author:   D. Rowell
%   Revision: 1.0 10-1-2007
%
%--------------------------------------------------------------
function Fout = fftx(f,ksize)
N = 2^ksize;
% Move the input data to the output array
Fout=f;
%
% Perform the "bit-reversed" re-ordering of the input data
%
MR = 0;
for M = 1:N-1
    L = N/2;
    while MR + L > N-1
       L = L/2;
    end
    MR = mod(MR,L) + L;
    if MR >= M
        temp       = Fout(M+1);   %swap the data points
        Fout(M+1)  = Fout(MR+1);
        Fout(MR+1) = temp;
    end
end
%
% Now perform the column operations
%
L = 1;
while L < N
    ISTEP = 2*L;
    for K = 1:L
        W = exp(-i*pi*(K-1)/L);
         for P = K:ISTEP:N
             Q = P + L;       % P & Q are the two points to be updated
             WBm     = W*Fout(Q);
             Fout(Q) = Fout(P) - WBm;  % Q identifies the "odd" block.
             Fout(P) = Fout(P) + WBm;  % P identifies "even" block,
         end
    end
         L = ISTEP;
end
```

```
%--------------------------------------------------------------------
%     *** 2.161 Signal Processing - Continuous and Discrete  ***
%
%   ifftx -  Tutorial inverse FFT routine to demonstrate the Radix-2
%            inverseFFT using a forward FFT routine.
%
%   fout = ifftx(F, k)
%   Arguments:
%      F  - frequency domain data set
%      k  - size of the data set k = log_2(N)
%
%   Author:   D. Rowell
%   Revision: 1.0 10-1-2007
%
%--------------------------------------------------------------------
function fout = ifftx(F,ksize)
N = 2^ksize;
%
% Conjugate the input data
fout=conj(F);
% Take the forward FFT, conjugate and divide by N
fout =demofft(fout,ksize);
fout=conj(fout)/N;
```

```
% Form a 16-point data seqience
t =[0:.375:15*.375];
f = exp(-t).*sin(t);
% Compute the FFT and then the IFFT
F = fftx(f,4);
f1 = real(ifftx(F,4));
% Plot the results
plot(t, f, t, f1, 'o');
```