Decimation-in-time (DIT) Radix-2 FFT

Version 1.7: Sep 15, 2006 7:46 am GMT-5

Douglas L. Jones

Abstract

The radix-2 algorithms are the simplest FFT algorithms. The decimation-in-time (DIT) radix-2 FFT recursively partitions a DFT into two half-length DFTs of the even-indexed and odd-indexed time samples. The outputs of these shorter FFTs are reused to compute many outputs, thus greatly reducing the total computational cost.

The radix-2 decimation-in-time and decimation-in-frequency[1] fast Fourier transforms (FFTs) are the simplest FFT algorithms[2]. Like all FFTs, they gain their speed by reusing the results of smaller, intermediate computations to compute multiple DFT frequency outputs.

1 Decimation in time

The radix-2 decimation-in-time algorithm rearranges the discrete Fourier transform (DFT) equation[3] into two parts: a sum over the even-numbered discrete-time indices $n = [0, 2, 4, \ldots, N-2]$ and a sum over the odd-numbered indices $n = [1, 3, 5, \ldots, N-1]$ as in (1):

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} \left(x(n)\, e^{-\left(i\frac{2\pi nk}{N}\right)} \right) \\ &= \sum_{n=0}^{\frac{N}{2}-1} \left(x(2n)\, e^{-\left(i\frac{2\pi(2n)k}{N}\right)} \right) + \sum_{n=0}^{\frac{N}{2}-1} \left(x(2n+1)\, e^{-\left(i\frac{2\pi(2n+1)k}{N}\right)} \right) \\ &= \sum_{n=0}^{\frac{N}{2}-1} \left(x(2n)\, e^{-\left(i\frac{2\pi nk}{\frac{N}{2}}\right)} \right) + e^{-\left(i\frac{2\pi k}{N}\right)} \sum_{n=0}^{\frac{N}{2}-1} \left(x(2n+1)\, e^{-\left(i\frac{2\pi nk}{\frac{N}{2}}\right)} \right) \\ &= \mathrm{DFT}_{\frac{N}{2}}\left[[x(0), x(2), \ldots, x(N-2)] \right] + W_N^k \mathrm{DFT}_{\frac{N}{2}}\left[[x(1), x(3), \ldots, x(N-1)] \right] \end{aligned} \tag{1}$$

The mathematical simplifications in (1) reveal that all DFT frequency outputs $X(k)$ can be computed as the sum of the outputs of two length-$\frac{N}{2}$ DFTs, of the even-indexed and odd-indexed discrete-time samples, respectively, where the odd-indexed short DFT is multiplied by a so-called **twiddle factor** term $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$. This is called a **decimation in time** because the time samples are rearranged in alternating groups, and a **radix-2** algorithm because there are two groups. Figure 1 graphically illustrates this form of the DFT computation, where for convenience the frequency outputs of the length-$\frac{N}{2}$ DFT of the even-indexed time samples are denoted $G(k)$ and those of the odd-indexed samples as $H(k)$. Because of the periodicity with $\frac{N}{2}$ frequency samples of these length-$\frac{N}{2}$ DFTs, $G(k)$ and $H(k)$ can be used to compute *two* of the length-N DFT frequencies, namely $X(k)$ and $X\left(k+\frac{N}{2}\right)$, but with a different twiddle factor. This reuse of these short-length DFT outputs gives the FFT its computational savings.

*http://creativecommons.org/licenses/by/1.0

[1] "Decimation-in-Frequency (DIF) Radix-2 FFT" <http://cnx.org/content/m12018/latest/>

[2] "Overview of Fast Fourier Transform (FFT) Algorithms" <http://cnx.org/content/m12026/latest/>

[3] "DFT Definition and Properties" <http://cnx.org/content/m12019/latest/>

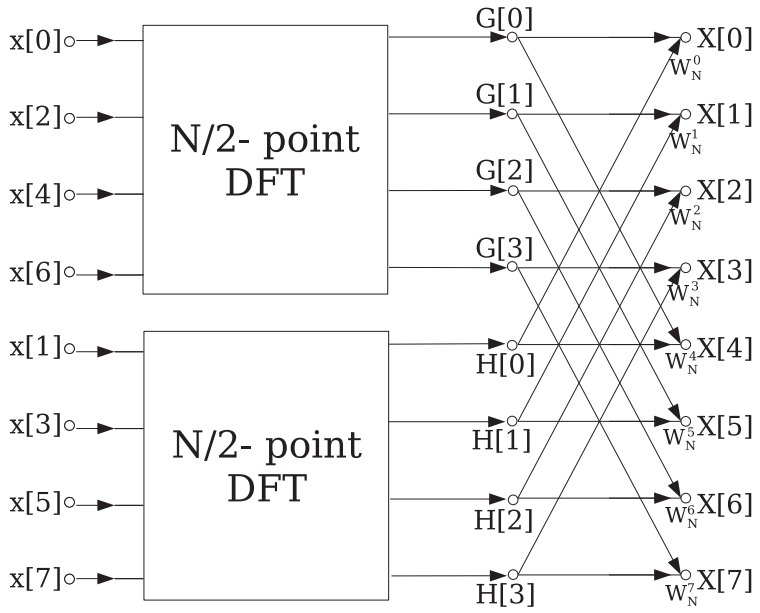

Figure 1: Decimation in time of a length-N DFT into two length-$\frac{N}{2}$ DFTs followed by a combining stage.

Whereas direct computation of all N DFT frequencies according to the DFT equation[4] would require N^2 complex multiplies and $N^2 - N$ complex additions (for complex-valued data), by reusing the results of the two short-length DFTs as illustrated in Figure 1, the computational cost is now

New Operation Counts

- $2\left(\frac{N}{2}\right)^2 + N = \frac{N^2}{2} + N$ complex multiplies
- $2\frac{N}{2}\left(\frac{N}{2} - 1\right) + N = \frac{N^2}{2}$ complex additions

This simple reorganization and reuse has reduced the total computation by almost a factor of two over direct DFT[5] computation!

2 Additional Simplification

A basic **butterfly** operation is shown in Figure 2, which requires only $\frac{N}{2}$**twiddle-factor** multiplies per **stage**. It is worthwhile to note that, after merging the twiddle factors to a single term on the lower branch, the remaining butterfly is actually a length-2 DFT! The theory of multi-dimensional index maps[6] shows that this must be the case, and that FFTs of any factorable length may consist of successive stages of shorter-length FFTs with twiddle-factor multiplications in between.

[4] "DFT Definition and Properties" <http://cnx.org/content/m12019/latest/>
[5] "DFT Definition and Properties" <http://cnx.org/content/m12019/latest/>
[6] "Multidimensional Index Maps" <http://cnx.org/content/m12025/latest/>

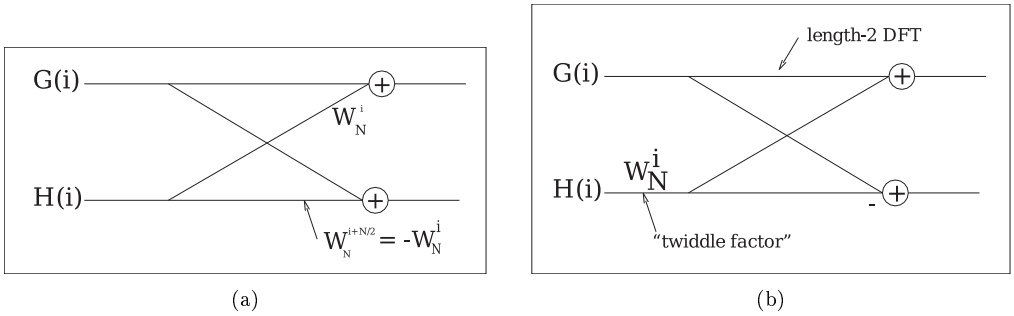

Figure 2: Radix-2 DIT butterfly simplification: both operations produce the same outputs

3 Radix-2 decimation-in-time FFT

The same radix-2 decimation in time can be applied recursively to the two length $\frac{N}{2}$DFT[7]s to save computation. When successively applied until the shorter and shorter DFTs reach length-2, the result is the radix-2 DIT FFT algorithm (Figure 3).

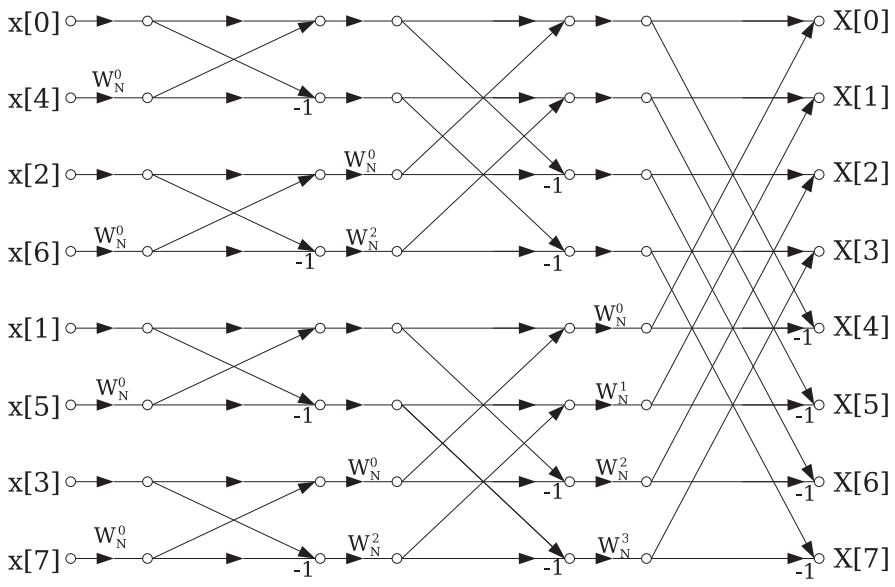

Figure 3: Radix-2 Decimation-in-Time FFT algorithm for a length-8 signal

[7] "DFT Definition and Properties" <http://cnx.org/content/m12019/latest/>

The full radix-2 decimation-in-time decomposition illustrated in Figure 3 using the simplified butterflies (Figure 2) involves $M = \log_2 N$ stages, each with $\frac{N}{2}$ butterflies per stage. Each butterfly requires 1 complex multiply and 2 adds per butterfly. The total cost of the algorithm is thus

Computational cost of radix-2 DIT FFT

- $\frac{N}{2}\log_2 N$ complex multiplies
- $N\log_2 N$ complex adds

This is a remarkable savings over direct computation of the DFT. For example, a length-1024 DFT would require 1048576 complex multiplications and 1047552 complex additions with direct computation, but only 5120 complex multiplications and 10240 complex additions using the radix-2 FFT, a savings by a factor of 100 or more. The relative savings increase with longer FFT lengths, and are less for shorter lengths.

Modest additional reductions in computation can be achieved by noting that certain twiddle factors, namely Using special butterflies for W_N^0, $W_N^{\frac{N}{2}}$, $W_N^{\frac{N}{4}}$, $W_N^{\frac{N}{8}}$, $W_N^{\frac{3N}{8}}$, require no multiplications, or fewer real multiplies than other ones. By implementing special butterflies for these twiddle factors as discussed in FFT algorithm and programming tricks, the computational cost of the radix-2 decimation-in-time FFT can be reduced to

- $2N\log_2 N - 7N + 12$ real multiplies
- $3N\log_2 N - 3N + 4$ real additions

NOTE: In a decimation-in-time radix-2 FFT as illustrated in Figure 3, the input is in **bit-reversed** order (hence "decimation-in-time"). That is, if the time-sample index n is written as a binary number, the order is that binary number reversed. The bit-reversal process is illustrated for a length-$N = 8$ example below.

Example 1: N=8

In-order index	In-order index in binary	Bit-reversed binary	Bit-reversed index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

It is important to note that, if the input signal data are placed in bit-reversed order before beginning the FFT computations, the outputs of each butterfly throughout the computation can be placed in the same memory locations from which the inputs were fetched, resulting in an **in-place algorithm** that requires no extra memory to perform the FFT. Most FFT implementations are in-place, and overwrite the input data with the intermediate values and finally the output.

4 Example FFT Code

The following function, written in the C programming language, implements a radix-2 decimation-in-time FFT. It is designed for computing the DFT of complex-valued inputs to produce complex-valued outputs,

with the real and imaginary parts of each number stored in separate double-precision floating-point arrays. It is an in-place algorithm, so the intermediate and final output values are stored in the same array as the input data, which is overwritten. After initializations, the program first bit-reverses the discrete-time samples, as is typical with a decimation-in-time algorithm (but see alternate FFT structures[8] for DIT algorithms with other input orders), then computes the FFT in stages according to the above description.

Ihis FFT program (p. 5) uses a standard three-loop structure for the main FFT computation. The outer loop steps through the stages (each column in Figure 3); the middle loop steps through "**flights**" (butterflies with the same twiddle factor from each short-length DFT at each stage), and the inner loop steps through the individual butterflies. This ordering minimizes the number of fetches or computations of the twiddle-factor values. Since the bit-reverse of a bit-reversed index is the original index, bit-reversal can be performed fairly simply by swapping pairs of data.

> NOTE: While of $O(N\log N)$ complexity and thus much faster than a direct DFT, this simple program is optimized for clarity, not for speed. A speed-optimized program making use of additional efficient FFT algorithm and programming tricks[9] will compute a DFT several times faster on most machines.

```
/*************************************************************/
/* fft.c                                                     */
/* (c) Douglas L. Jones                                      */
/* University of Illinois at Urbana-Champaign                */
/* January 19, 1992                                          */
/*                                                           */
/*   fft: in-place radix-2 DIT DFT of a complex input        */
/*                                                           */
/*   input:                                                  */
/* n: length of FFT: must be a power of two                  */
/* m: n = 2**m                                               */
/*   input/output                                            */
/* x: double array of length n with real part of data        */
/* y: double array of length n with imag part of data        */
/*                                                           */
/*   Permission to copy and use this program is granted      */
/*   under a Creative Commons "Attribution" license          */
/*   http://creativecommons.org/licenses/by/1.0/             */
/*************************************************************/
fft(n,m,x,y)
int n,m;
double x[],y[];
{
int i,j,k,n1,n2;
double c,s,e,a,t1,t2;


j = 0; /* bit-reverse */
n2 = n/2;
for (i=1; i < n - 1; i++)
```

[8] "Alternate FFT Structures" <http://cnx.org/content/m12012/latest/>
[9] "Efficient FFT Algorithm and Programming Tricks" <http://cnx.org/content/m12021/latest/>

```
{
  n1 = n2;
  while ( j >= n1 )
   {
    j = j - n1;
    n1 = n1/2;
   }
  j = j + n1;

  if (i < j)
   {
    t1 = x[i];
    x[i] = x[j];
    x[j] = t1;
    t1 = y[i];
    y[i] = y[j];
    y[j] = t1;
   }
}


n1 = 0; /* FFT */
n2 = 1;

for (i=0; i < m; i++)
{
  n1 = n2;
  n2 = n2 + n2;
  e = -6.283185307179586/n2;
  a = 0.0;

  for (j=0; j < n1; j++)
   {
    c = cos(a);
    s = sin(a);
    a = a + e;

    for (k=j; k < n; k=k+n2)
     {
      t1 = c*x[k+n1] - s*y[k+n1];
      t2 = s*x[k+n1] + c*y[k+n1];
      x[k+n1] = x[k] - t1;
      y[k+n1] = y[k] - t2;
      x[k] = x[k] + t1;
      y[k] = y[k] + t2;
     }
   }
}

return;
}
```