

Modified Cooley-Tukey algorithms based on a generalized DFT framework

Steven G. Johnson

Created October 1, 2003; last updated October 16, 2008

1 Introduction

The purpose of this note is to outline “new” FFT algorithms based on a “generalized” discrete Fourier transform (**GDFT**) framework. These algorithms are in some sense a generalization of Cooley-Tukey but differ in that they can break up a given twiddle factor from Cooley-Tukey into multiple twiddle factors, subsequently recombined in a way that changes the twiddle access pattern. In another sense, they are equivalent to standard Cooley-Tukey, but use DIT-based twiddle factors with a DIF access pattern, or vice versa (*proof left as an exercise for the reader*). Although we were originally inspired by a polynomial-factorization framework similar to that of Bruun’s algorithm, we present them here in a more traditional algebraic formulation that permits easier implementation and comparison to standard Cooley-Tukey.

The basic transform that we will decompose is not the ordinary DFT, but the GDFT, defined by:

$$y_k = \sum_{j=0}^{n-1} \omega_n^{(j+a)(k+b)} x_j$$

for some arbitrary complex numbers a and b , where $\omega_n \equiv e^{-2\pi i/n}$ and $a = b = 0$ gives the ordinary DFT. Of course, the GDFT can be trivially computed via the DFT by pre-multiplying the input by the b factors and post-multiplying the output by the a factors, but we will not do that here.

2 Modified Cooley-Tukey

To compute the GDFT in the case of composite $n = n_1 n_2$, we use the Cooley-Tukey re-indexing $j = j_1 + j_2 n_1$ and $k = k_1 n_2 + k_2$. This yields the nested sum:

$$y_{k_1 n_2 + k_2} = \omega_n^{(k_1 n_2 + k_2)a} \sum_{j_1=0}^{n_1-1} \omega_{n_1}^{j_1 k_1} \omega_n^{j_1 k_2} \left(\sum_{j_2=0}^{n_2-1} \omega_{n_2}^{j_2 k_2} \omega_n^{(j_1 + j_2 n_1)b} x_{j_1 + j_2 n_1} \right).$$

The ordinary Cooley-Tukey algorithm would recurse in more or less the order indicated by the parenthesization: multiply x_j by the b factors (if any) first, then do the size- n_2 DFTs, then multiply by the $j_1 k_2$ **twiddle factors**, then do the size- n_1 DFTs, then

multiply by the a factors (if any). For a small **radix** r , $n_1 = r$ gives a decimation-in-time (**DIT**) algorithm and $n_2 = r$ gives a decimation-in-frequency (**DIF**) algorithm.

However, this decomposition can be performed in another way by making the a and b factors a part of the recursion, splitting and recombining them with the ordinary twiddle factors. That is, we write the nested sum above as a nested pair of GDFTs:

$$y_{k_1 n_2 + k_2} = \sum_{j_1=0}^{n_1-1} \omega_{n_1}^{(j_1+a_1)(k_1+b_1)} \left(\sum_{j_2=0}^{n_2-1} \omega_{n_2}^{(j_2+a_2)(k_2+b_2)} x_{j_1+j_2 n_1} \right),$$

where there are two possible choices of a_1 , b_1 , a_2 , and b_2 . One possible choice is:

$$\begin{aligned} a_1 &= a, \\ b_1 &= (k_2 + b)/n_2, \\ a_2 &= a/n_1, \\ b_2 &= b. \end{aligned}$$

Note that b_1 is a function of k_2 . Alternatively, one can use the same $a_1 = a$ and $b_2 = b$ as above, but choose:

$$\begin{aligned} b_1 &= b/n_2, \\ a_2 &= (j_1 + a)/n_1, \end{aligned}$$

where now a_2 is a function of j_1 .

In both cases, a key difference from ordinary Cooley-Tukey algorithm can be achieved from the fact that, in the nested transforms, the original a and b both appear *twice*. Since the $j_1 k_2$ twiddle factor is absorbed in to the new b_1 or a_2 , this means that it will be *split* into two pieces at the next stage of the recursion, instead of being multiplied in one step as in standard Cooley-Tukey. Of course, this assumes that there *is* a next stage of the recursion. That is, to be different from ordinary Cooley-Tukey, the first alternative requires that the size- n_1 transform (which contains the twiddle via b_1) be subdivided further, corresponding to a DIF algorithm; conversely, the second alternative requires that the size- n_2 transform (which contains the twiddle via a_2) be subdivided further, corresponding to a DIT algorithm. (Employing the first/second alternative with DIT/DIF yields the usual DIT/DIF Cooley-Tukey algorithm.)

2.1 Modified DIT Cooley-Tukey

In the modified DIT formulation from above, applied to compute the ordinary DFT, b is always zero. We get simply

$$y_{k_1 n_2 + k_2} = \sum_{j_1=0}^{n_1-1} \omega_{n_1}^{(j_1+a_1)k_1} \left(\sum_{j_2=0}^{n_2-1} \omega_{n_2}^{(j_2+a_2(j_1))k_2} x_{j_1+j_2 n_1} \right),$$

where $a_1 = a$ and $a_2(j_1) = (j_1 + a)/n_1$. If the recursion is done in a depth-first fashion, this has the same memory access pattern as the usual DIT Cooley-Tukey algorithm, but with a striking difference in the twiddle factors: the twiddle factor multiplied by the

output (instead of the input) of the outer DFT is dependent *only* on k_1 and is independent of k_2 . (This may have cache benefits, since one performs n_2 butterflies with the same set of n_1 twiddle factors.) Moreover, the twiddle factor for the topmost level of the recursion is trivial (unity), while conversely the leaf nodes of the recursion *do* have twiddle factors (since they are GDFTs and not DFTs). In terms of FFTW's codelets, this would use nontwiddle codelets for the topmost level and *DIF* twiddle codelets for subsequent levels, but with a *DIT* access pattern.

3 Application to Real-data FFTs

One of the more intriguing possibilities of our modified Cooley-Tukey structure is the development of new real-data FFT algorithms. Currently, real-data FFT algorithms based on pruning the redundant computations from the complex-data algorithm, such as Sorensen's or FFTW's, have an important limitation: real-input (hermitian-output) algorithms must be *DIT*, and real-output (hermitian-input) algorithms must be *DIF*. Thus, for example, an out-of-place real-output FFTW must unfortunately destroy its input array (without additional buffer space or bit-reversal passes, or in FFTW3 additional passes to re-express via DHTs). Another consequence is that certain efficient possibilities for in-place algorithms are precluded; in particular, for complex data with a size of the form pq^2 , one can combine a pair of size- q *DIT* and *DIF* steps, combined with a $q \times q$ transpose, to recursively reduce the problem to a smaller in-place transform. The modified Cooley-Tukey algorithm should lift these restrictions. For example, for a real-output transform, one can use the naturally prunable *DIF* computations but in a *DIT* computational pattern that need not destroy the input array. Also, for a real-input transform, one can develop a *DIT*-based-pruning algorithm with *DIF* structure and *DIF*-like reordering requirements, combined with the usual *DIT* algorithm, for in-place computations.