A modified split-radix FFT with fewer arithmetic operations

Steven G. Johnson* and Matteo Frigo

Abstract— **Recent results by Van Buskirk *et al.* have broken the record set by Yavne in 1968 for the lowest exact count of real additions and multiplications to compute a power-of-two discrete Fourier transform (DFT). Here, we present a simple recursive modification of the split-radix algorithm that computes the DFT with asymptotically about 6% fewer operations than Yavne, matching the count achieved by Van Buskirk's program-generation framework. We also discuss the application of our algorithm to real-data and real-symmetric (discrete cosine) transforms, where we are again able to achieve lower arithmetic counts than previously published algorithms.**

Index Terms— **FFT, DCT, split radix, arithmetic complexity**

I. Introduction

ALL known fast Fourier transform (FFT) algorithms compute the discrete Fourier transform (DFT) of size N in $\Theta(N \log N)$ operations,[1] so any improvement in them appears to rely on reducing the exact number or cost of these operations rather than their asymptotic functional form. For many years, the time to perform an FFT was dominated by real-number arithmetic, and so considerable effort was devoted towards proving and achieving lower bounds on the exact count of arithmetic operations (real additions and multiplications), herein called "flops" (floating-point operations), required for a DFT of a given size [2]. Although the performance of FFTs on recent computer hardware is determined by many factors besides pure arithmetic counts [3], there still remains an intriguing unsolved mathematical question: what is the smallest number of flops required to compute a DFT of a given size N, in particular for the important case of $N = 2^m$? In 1968, Yavne [4] presented what became known as the "split-radix" FFT algorithm [5]–[7] for $N = 2^m$, and achieved a record flop count of $4N \lg N - 6N + 8$ for $N > 1$ (where lg denotes $\log_2$), an improvement by 20% over the classic "radix-2" algorithm presented by Cooley and Tukey (flops $\sim 5N \lg N$) [8]. Here, we present a modified version of the split-radix FFT that (without sacrificing numerical accuracy) lowers the flop count by a further $\sim 5.6\%$ ($\frac{1}{18}$) to:

$$\frac{34}{9} N \lg N - \frac{124}{27} N - 2 \lg N - \frac{2}{9} (-1)^{\lg N} \lg N + \frac{16}{27} (-1)^{\lg N} + 8 \quad (1)$$

* S. G. Johnson (stevenj@math.mit.edu) is with the Massachusetts Institute of Technology, 77 Mass. Ave. Rm. 2-388, Cambridge, MA 02139; tel. 617-253-4073, fax 617-253-8911. He was supported in part by the Materials Research Science and Engineering Center program of the National Science Foundation under award DMR-9400334.

M. Frigo (athena@fftw.org) is with the IBM Austin Research Laboratory, 11501 Burnet Rd., Austin, TX 78758; tel. 512-838-8173, fax 512-838-4036.

[1]We employ the standard notation of Ω and Θ for asymptotic lower and tight bounds, respectively [1].

TABLE I
Flop counts (real additions + multiplications) of standard complex-data split radix and our new algorithm

N	Yavne split radix	New algorithm
64	1160	1152
128	2824	2792
256	6664	6552
512	15368	15048
1024	34824	33968
2048	77832	75688
4096	172040	166856
8192	376840	364680
16384	819208	791264

for $N > 1$, where the savings (starting at $N = 64$) are due to simplifications of complex multiplications. See also Table I. More specifically, throughout most of this paper we assume that complex multiplications are implemented with the usual 4 real multiplications and 2 real additions (as opposed to the 3 mults + 3 adds variant [9]), and in this case the savings are purely in the number of real multiplications.

The first demonstration of this improved count was in a 2004 Usenet post by Van Buskirk [10], who had managed to save 8 operations over Yavne by hand optimization for $N = 64$, using an unusual algorithm based on decomposing the DFT into its real and imaginary and even-symmetry and odd-symmetry components (essentially, type-I discrete cosine and sine transforms). These initial gains came by rescaling the size-8 sub-transforms and absorbing the scale factor elsewhere in the computation (related savings occur in the type-II discrete cosine transform of size 8, where one can save six multiplications by rescaling the outputs [11] as discussed in Sec. VIII). Van Buskirk *et al.* later developed an automatic code-generation implementation of his approach that achieves Eq. (1) given an arbitrary fixed $N = 2^m$ [12], [13]. Meanwhile, following his initial posting, we developed a way to explicitly achieve the same savings recursively in a more conventional split-radix algorithm. Our split-radix approach involves a recursive rescaling of the trigonometric constants ("twiddle factors" [14]) in sub-transforms of the DFT decomposition (while the final FFT result is still the correct, unscaled value), relying on four mutually recursive stages.

A few rigorous bounds on the DFT's arithmetic complexity have been proven in the literature, but no tight lower bound on the flop count is known (and we make no claim that Eq. (1) is the lowest possible). Following work by Winograd [15], a realizable $\Theta(N)$ lower bound *is* known for the number

of irrational real multiplications for $N = 2^m$, given by $4N - 2\lg^2 N - 2\lg N - 4$ [2], [16], [17] (matching split radix as well as our algorithm up to $N = 16$), but is achieved only at the price of many more additions and thus has limited utility on CPUs with hardware multipliers. The DFT has been shown to require $\Omega(N \log N)$ complex-number additions for linear algorithms under the assumption of bounded multiplicative constants [18], or alternatively assuming a bound on a measure of the algorithm's "asynchronicity" [19]. Furthermore, the number Nm of complex-number additions obtained in Cooley-Tukey–related algorithms (such as split-radix) for $N = 2^m$ has been argued to be optimal [20] over a broad class of algorithms that do not exploit additive identities in the roots of unity.[2] Our algorithm does not change this number of complex additions.

In the following, we first review the known variant of the split-radix FFT that is the starting point for our modifications, then describe our modified algorithm, analyze its arithmetic costs (both theoretically and with two sample implementations instrumented to count the operations) as well as its numerical accuracy, describe its application to real-input and real-symmetric (discrete cosine) transforms where one also finds arithmetic gains over the literature, and conclude with some remarks about practical realizations and further directions.

II. Conjugate-pair split-radix FFT

The starting point for our improved algorithm is not the standard split-radix algorithm, but rather a variant called the "conjugate-pair" FFT that was itself initially proposed to reduce the number of flops [21], but its operation count was later proved identical to that of ordinary split radix [22]–[24]. This variant was rediscovered in unpublished work by Bernstein [25], who argued that it reduces the number of twiddle-factor loads. We use it for a related reason: because the conjugate-pair FFT exposes redundancies in the twiddle factors, it enables rescalings and simplifications of twiddle pairs that we do not know how to extract from the usual split-radix formulation. To derive the algorithm, recall that the DFT is defined by:

$$y_k = \sum_{n=0}^{N-1} \omega_N^{nk} x_n, \tag{2}$$

where $k = 0 \ldots N-1$ and ω_N is the primitive root of unity $\exp(-2\pi i/N)$. Then, for N divisible by 4, we perform a decimation-in-time decomposition of x_n into three smaller DFTs, of x_{2n_2} (the even elements), x_{4n_4+1}, and x_{4n_4-1} (where $x_{-1} = x_{N-1}$)—this last sub-sequence would be x_{4n_4+3} in standard split radix, but here is shifted cyclically by -4.[3] We obtain:

$$\begin{aligned} y_k &= \sum_{n_2=0}^{N/2-1} \omega_{N/2}^{n_2 k} x_{2n_2} \\ &\quad + \omega_N^k \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k} x_{4n_4+1} \\ &\quad + \omega_N^{-k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k} x_{4n_4-1}, \end{aligned} \tag{3}$$

where the ω_N^k and ω_N^{-k} are the conjugate pair of twiddle factors (whereas ordinary split radix would have ω_N^k and ω_N^{3k}). (In this paper, we will use the term "twiddle factor" to refer to all data-independent trigonometric constants that appear in an FFT.) These summations are DFTs of size $N/2$ and $N/4$, and the ω_N^k for $k \geq N/4$ are related to $k = 0 \ldots N/4-1$ via trivial multiplications by i and -1. Thus, we obtain Algorithm 1, in which the results of the three sub-transforms are denoted by u_k, z_k, and z'_k.

Algorithm 1 Standard conjugate-pair split-radix FFT of length N (divisible by 4). (Special-case optimizations for $k = 0$ and $k = N/8$ are not shown.)

function $y_{k=0..N-1} \leftarrow \mathbf{splitfft}_N(x_n)$:
  $u_{k_2=0\ldots N/2-1} \leftarrow \mathbf{splitfft}_{N/2}(x_{2n_2})$
  $z_{k_4=0\ldots N/4-1} \leftarrow \mathbf{splitfft}_{N/4}(x_{4n_4+1})$
  $z'_{k_4=0\ldots N/4-1} \leftarrow \mathbf{splitfft}_{N/4}(x_{4n_4-1})$
  for $k = 0$ to $N/4-1$ **do**
    $y_k \leftarrow u_k + \left(\omega_N^k z_k + \omega_N^{-k} z'_k\right)$
    $y_{k+N/2} \leftarrow u_k - \left(\omega_N^k z_k + \omega_N^{-k} z'_k\right)$
    $y_{k+N/4} \leftarrow u_{k+N/4} - i\left(\omega_N^k z_k - \omega_N^{-k} z'_k\right)$
    $y_{k+3N/4} \leftarrow u_{k+N/4} + i\left(\omega_N^k z_k - \omega_N^{-k} z'_k\right)$
  end for

For clarity, Algorithm 1 omits special-case optimizations for $k = 0$ in the loop (where ω_N^k is unity and $\omega_N^k z_k$ requires no flops) and for $k = N/8$ (where $\omega_N^k = (1-i)/\sqrt{2}$ and requires only 2 real multiplications instead of 4 for $\omega_N^k z_k$). It also omits the base cases of the recursion: $N = 1$ is just a copy $y_0 = x_0$, and $N = 2$ is an addition $y_0 = x_0 + x_1$ and a subtraction $y_1 = x_0 - x_1$. With these optimizations and base cases, the standard assumptions that multiplications by ± 1 and $\pm i$ are free,[4] and extracting common sub-expressions such as $\omega_N^k z_k \pm \omega_N^{-k} z'_k$, the flop count of Yavne is obtained. More specifically, the number of real additions $\alpha(N)$ and multiplications $\mu(N)$ (for 4/2 mult/add complex multiplies) is [26]:

$$\alpha(N) = \frac{8}{3} N \lg N - \frac{16}{9} N - \frac{2}{9}(-1)^{\lg N} + 2 \tag{4}$$

$$\mu(N) = \frac{4}{3} N \lg N - \frac{38}{9} N + \frac{2}{9}(-1)^{\lg N} + 6 \tag{5}$$

Traditionally, the recursion is "flattened" into an iterative algorithm that performs all the FFTs of a given size at once [27], may work in-place, can exploit $\omega_N^{N/4-k} = -i\omega_N^{-k}$ to halve the number of twiddle factors (see Sec. VI), *etc.*,

[2] Although this was originally framed as a bound on the number of multiplications [20], because multiplications by unity were included it was later construed as a statement about the number of additions [2].

[3] Past formulations of the conjugate-pair FFT sent $n_4 \rightarrow -n_4$ and used an inverse DFT for this sub-transform, but they are essentially equivalent to our expression; the difference is a matter of convenience only.

[4] In the algorithms of this paper, all negations can be eliminated by turning additions into subtractions or vice-versa.

but none of this affects the flop count. Although in this paper we consider only decimation-in-time (DIT) decompositions, a dual decimation-in-frequency (DIF) algorithm can always be obtained by network transposition[5] (reversing the flow graph of the computation) with identical operation counts.

III. New FFT: Rescaling the twiddles

The key to reducing the number of operations is the observation that, in Algorithm 1, both z_k and z'_k (the k-th outputs of the size-$N/4$ sub-transforms) are multiplied by a twiddle factor ω_N^k or ω_N^{-k} before they are used to find y_k. This means that we can rescale the size-$N/4$ sub-transforms by *any* factor $1/s_{N/4,k}$ desired, and absorb the scale factor into $\omega_N^k s_{N/4,k}$ at no cost. So, we merely need to find a rescaling that will save some operations in the sub-transforms. (As is conventional in counting FFT operations, we assume that all data-independent constants like $\omega_N^k s_{N/4,k}$ are precomputed and are therefore not included in the flops.) Moreover, we rely on the fact that z_k and z'_k have conjugate twiddle factors in the conjugate-pair algorithm, so that a single rescaling below will simplify both twiddle factors to save operations—this is not true for the ω_N^k and ω_N^{3k} factors in the usual split radix. Below, we begin with an outline of the general ideas, and then analyze the precise algorithm in Sec. IV.

Consider a sub-transform of a given size N that we wish to rescale by some $1/s_{N,k}$ for each output y_k. Suppose we take $s_{N,k} = s_{N,k+N/4} = \cos(2\pi k/N)$ for $k \le N/8$. In this case, y_k from Algorithm 1 becomes $y_k \leftarrow u_k/s_{N,k} + (t_{N,k} z_k + t^*_{N,k} z'_k)$, where

$$t_{N,k} = 1 - i\tan(2\pi k/N) = \omega_N^k / \cos(2\pi k/N). \qquad (6)$$

Multiplying $\omega_N^k z_k$ requires 4 real multiplications and 2 real additions (6 flops) for general k, but multiplying $t_{N,k} z_k$ requires only 2 real multiplications and 2 real additions (4 flops). (A similar rescaling was proposed [29] to increase the number of fused multiply-add operations, and an analogous rescaling also relates the Givens and "fast Givens" QR algorithms [30].) Thus, we have saved 4 real multiplications in computing $t_{N,k} z_k \pm t^*_{N,k} z'_k$, but spent 2 real multiplications in $u_k/s_{N,k}$ and another 2 for $u_{k+N/4}/s_{N,k}$, for what may seem to be no net change. However, instead of computing $u_k/s_{N,k}$ directly, we can instead push the $1/s_{N,k}$ scale factor "down" into the recursive computation of u_k. In this way, it turns out that we can save most of these "extra" multiplications by combining them with twiddle factors inside the $N/2$ transform. Indeed, we shall see that we need to push $1/s_{N,k}$ down through *two* levels of recursion in order to gain all of the possible savings.

Moreover, we perform the rescaling recursively, so that the sub-transforms z_k and z'_k are themselves rescaled by $1/s_{N/4,k}$ for the same savings, and the product of the sub-transform scale factors is combined with $\cos(2\pi k/N)$ and pushed up to the top-level transform. The resulting scale factor $s_{N,k}$ is given by the following recurrence, where we let $k_4 = k \mod N/4$:

[5] Network transposition is equivalent to matrix transposition [28] and preserves both the DFT (a symmetric matrix) and the flop count (for equal numbers of inputs and outputs), but changes DIT into DIF and vice versa.

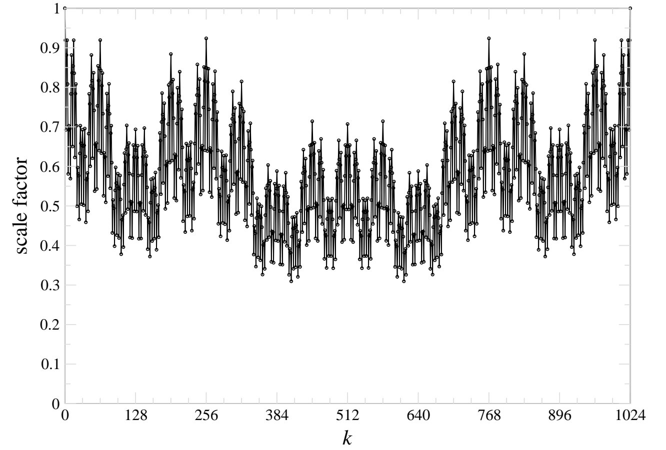


Fig. 1. Scale factor $s_{N,k}$, from Eq. (7), vs. one period of k for $N = 2^{12} = 4096$.

$$s_{N=2^m,k} = \begin{cases} 1 & \text{for } N \le 4 \\ s_{N/4,k_4}\cos(2\pi k_4/N) & \text{for } k_4 \le N/8 \\ s_{N/4,k_4}\sin(2\pi k_4/N) & \text{otherwise} \end{cases}, \qquad (7)$$

which has an interesting fractal pattern plotted in Fig. 1. This definition has the properties: $s_{N,0} = 1$, $s_{N,k+N/4} = s_{N,k}$, and $s_{N,N/4-k} = s_{N,k}$ (a symmetry whose importance appears in subsequent sections). We can now generally define:

$$t_{N,k} = \omega_N^k s_{N/4,k}/s_{N,k}, \qquad (8)$$

where $s_{N/4,k}/s_{N,k}$ is either sec or csc and thus $t_{N,k}$ is *always* of the form $\pm 1 \pm i\tan$ or $\pm\cot \pm i$. This last property is critical because it means that we obtain $t_{N,k} z_k \pm t^*_{N,k} z'_k$ in all of the scaled transforms and multiplication by $t_{N,k}$ requires at most 4 flops as above.

Rather than elaborate further at this point, we now simply present the algorithm, which consists of four mutually-recursive split-radix–like functions listed in Algorithms 2–3, and analyze it in the next section. As in the previous section, we omit for clarity the special-case optimizations for $k = 0$ and $k = N/8$ in the loops, as well as the trivial base cases for $N = 1$ and $N = 2$.

IV. Operation counts

Algorithms 2 and 3 manifestly have the same number of real additions as Algorithm 1 (for 4/2 mult/add complex multiplies), since they only differ by real multiplicative scale factors. So, all that remains is to count the number $M(N)$ of real multiplications *saved* compared to Algorithm 1, and this will give the number of flops saved over Yavne. We must also count the numbers $M_S(N)$, $M_{S2}(N)$, and $M_{S4}(N)$ of real multiplications saved (or spent, if negative) in our three rescaled sub-transforms. In $\mathbf{newfft}_N(x)$ itself, the number of multiplications is clearly the same as in $\mathbf{splitfft}_N(x)$, since all scale factors are absorbed into the twiddle factors—note that $s_{N/4,0} = 1$ so the $k = 0$ special case is not worsened either—and thus the savings come purely in the sub-transforms:

$$M(N) = M(N/2) + 2M_S(N/4). \qquad (9)$$

Algorithm 2 New FFT algorithm of length N (divisible by 4). The sub-transforms $\mathbf{newfftS}_{N/4}(x)$ are rescaled by $s_{N/4,k}$ to save multiplications. The sub-sub-transforms of size $N/8$, in turn, use two additional recursive subroutines from Algorithm 3 (four recursive functions in all, which differ in their rescalings).

function $y_{k=0..N-1} \leftarrow \mathbf{newfft}_N(x_n)$:
  {computes DFT}
  $u_{k_2=0...N/2-1} \leftarrow \mathbf{newfft}_{N/2}(x_{2n_2})$
  $z_{k_4=0...N/4-1} \leftarrow \mathbf{newfftS}_{N/4}(x_{4n_4+1})$
  $z'_{k_4=0...N/4-1} \leftarrow \mathbf{newfftS}_{N/4}(x_{4n_4-1})$
  for $k = 0$ to $N/4-1$ **do**
    $y_k \leftarrow u_k + \left(\omega_N^k s_{N/4,k} z_k + \omega_N^{-k} s_{N/4,k} z'_k\right)$
    $y_{k+N/2} \leftarrow u_k - \left(\omega_N^k s_{N/4,k} z_k + \omega_N^{-k} s_{N/4,k} z'_k\right)$
    $y_{k+N/4} \leftarrow u_{k+N/4}$
      $- i\left(\omega_N^k s_{N/4,k} z_k - \omega_N^{-k} s_{N/4,k} z'_k\right)$
    $y_{k+3N/4} \leftarrow u_{k+N/4}$
      $+ i\left(\omega_N^k s_{N/4,k} z_k - \omega_N^{-k} s_{N/4,k} z'_k\right)$
  end for

function $y_{k=0..N-1} \leftarrow \mathbf{newfftS}_N(x_n)$:
  {computes DFT / $s_{N,k}$}
  $u_{k_2=0...N/2-1} \leftarrow \mathbf{newfftS2}_{N/2}(x_{2n_2})$
  $z_{k_4=0...N/4-1} \leftarrow \mathbf{newfftS}_{N/4}(x_{4n_4+1})$
  $z'_{k_4=0...N/4-1} \leftarrow \mathbf{newfftS}_{N/4}(x_{4n_4-1})$
  for $k = 0$ to $N/4-1$ **do**
    $y_k \leftarrow u_k + \left(t_{N,k} z_k + t^*_{N,k} z'_k\right)$
    $y_{k+N/2} \leftarrow u_k - \left(t_{N,k} z_k + t^*_{N,k} z'_k\right)$
    $y_{k+N/4} \leftarrow u_{k+N/4} - i\left(t_{N,k} z_k - t^*_{N,k} z'_k\right)$
    $y_{k+3N/4} \leftarrow u_{k+N/4} + i\left(t_{N,k} z_k - t^*_{N,k} z'_k\right)$
  end for

In $\mathbf{newfftS}_N(x)$, as discussed above, the substitution of $t_{N,k}$ for ω_N^k means that 2 real multiplications are saved from each twiddle factor, or 4 multiplications per iteration k of the loop. This saves N multiplications, except that we have to take into account the $k = 0$ and $k = N/8$ special cases. For $k = N/8$, $t_{N,k} = 1 - i$, again saving two multiplications (by $1/\sqrt{2}$) per twiddle factor. Since $t_{N,0} = 1$, however, the $k = 0$ special case is unchanged (no multiplies), so we only save $N - 4$ multiplies overall. Thus,

$$M_S(N) = M_{S2}(N/2) + 2M_S(N/4) + N - 4. \qquad (10)$$

At first glance, the $\mathbf{newfftS2}_N(x)$ routine may seem to have the same number of multiplications as $\mathbf{splitfft}_N(x)$, since the 2 multiplications saved in each $t_{N,k}$ (as above) are exactly offset by the $s_{N,k}/s_{2N,k}$ multiplications. (Note that we do not fold the $s_{N,k}/s_{2N,k}$ into the $t_{N,k}$ because we also have to scale by $s_{N,k}/s_{2N,k+N/4}$ and would thus destroy the $t_{N,k}z_k$ common sub-expression.) However, we spend 2 extra multiplications in the $k = 0$ special case, which ordinarily requires no multiplies, since $s_{N,0}/s_{2N,N/4} = 1/s_{2N,N/4} \neq \pm 1$ (for $N > 2$) appears in $y_{N/4}$ and $y_{3N/4}$. Thus,

$$M_{S2}(N) = M_{S4}(N/2) + 2M_S(N/4) - 2. \qquad (11)$$

Algorithm 3 Rescaled FFT subroutines called recursively from Algorithm 2. The loops in these routines have *more* multiplications than in Algorithm 1, but this is offset by savings from $\mathbf{newfftS}_{N/4}(x)$ in Algorithm 2.

function $y_{k=0..N-1} \leftarrow \mathbf{newfftS2}_N(x_n)$:
  {computes DFT / $s_{2N,k}$}
  $u_{k_2=0...N/2-1} \leftarrow \mathbf{newfftS4}_{N/2}(x_{2n_2})$
  $z_{k_4=0...N/4-1} \leftarrow \mathbf{newfftS}_{N/4}(x_{4n_4+1})$
  $z'_{k_4=0...N/4-1} \leftarrow \mathbf{newfftS}_{N/4}(x_{4n_4-1})$
  for $k = 0$ to $N/4-1$ **do**
    $y_k \leftarrow u_k + \left(t_{N,k} z_k + t^*_{N,k} z'_k\right) \cdot (s_{N,k}/s_{2N,k})$
    $y_{k+N/2} \leftarrow u_k - \left(t_{N,k} z_k + t^*_{N,k} z'_k\right) \cdot (s_{N,k}/s_{2N,k})$
    $y_{k+N/4} \leftarrow u_{k+N/4}$
      $- i\left(t_{N,k} z_k - t^*_{N,k} z'_k\right) \cdot (s_{N,k}/s_{2N,k+N/4})$
    $y_{k+3N/4} \leftarrow u_{k+N/4}$
      $+ i\left(t_{N,k} z_k - t^*_{N,k} z'_k\right) \cdot (s_{N,k}/s_{2N,k+N/4})$
  end for

function $y_{k=0..N-1} \leftarrow \mathbf{newfftS4}_N(x_n)$:
  {computes DFT / $s_{4N,k}$}
  $u_{k_2=0...N/2-1} \leftarrow \mathbf{newfftS2}_{N/2}(x_{2n_2})$
  $z_{k_4=0...N/4-1} \leftarrow \mathbf{newfftS}_{N/4}(x_{4n_4+1})$
  $z'_{k_4=0...N/4-1} \leftarrow \mathbf{newfftS}_{N/4}(x_{4n_4-1})$
  for $k = 0$ to $N/4-1$ **do**
    $y_k \leftarrow \left[u_k + \left(t_{N,k} z_k + t^*_{N,k} z'_k\right)\right] \cdot (s_{N,k}/s_{4N,k})$
    $y_{k+N/2} \leftarrow \left[u_k - \left(t_{N,k} z_k + t^*_{N,k} z'_k\right)\right]$
      $\cdot (s_{N,k}/s_{4N,k+N/2})$
    $y_{k+N/4} \leftarrow \left[u_{k+N/4} - i\left(t_{N,k} z_k - t^*_{N,k} z'_k\right)\right]$
      $\cdot (s_{N,k}/s_{4N,k+N/4})$
    $y_{k+3N/4} \leftarrow \left[u_{k+N/4} + i\left(t_{N,k} z_k - t^*_{N,k} z'_k\right)\right]$
      $\cdot (s_{N,k}/s_{4N,k+3N/4})$
  end for

Finally, the routine $\mathbf{newfftS4}_N(x)$ involves $\Theta(N)$ more multiplications than ordinary split-radix, although we have endeavored to minimize this by proper groupings of the operands. We save 4 real multiplications per loop iteration k because of the $t_{N,k}$ replacing ω_N^k. However, because each output has a distinct scale factor ($s_{4N,k} \neq s_{4N,k+N/2} \neq s_{4N,k+N/4} \neq s_{4N,k+3N/4}$), we spend 8 real multiplications per iteration, for a net increase of 4 multiplies per iteration k. For the $k = 0$ iteration, however, the $t_{N,0} = 1$ gains us nothing, while $s_{4N,0} = 1$ does not cost us, so we spend 6 net multiplies instead of 4, and therefore:

$$M_{S4}(N) = M_{S2}(N/2) + 2M_S(N/4) - N - 2. \qquad (12)$$

Above, we omitted the base cases of the recurrences, *i.e.* the $N = 1$ or 2 that we handle directly as in Sec. II (without recursion). There, we find $M(N \le 2) = M_S(N \le 2) = M_{S2}(N \le 2) = M_{S4}(1) = 0$ (where the scale factors are unity), and $M_{S4}(2) = -2$. Finally, solving these recurrences

by standard generating-function methods [1] (for $N > 1$):

$$M(N) = \frac{2}{9}N\lg N - \frac{38}{27}N + 2\lg N + \frac{2}{9}(-1)^{\lg N}\lg N - \frac{16}{27}(-1)^{\lg N}. \quad (13)$$

Subtracting Eq. (13) from the flop count of Yavne, we obtain Eq. (1) and Table I. Separate counts of real adds/mults are obtained by subtracting (13) from (5).

In the above discussion, one immediate question that arises is: why stop at four routines? Why not take the scale factors in $\mathbf{newfftS4}_N(x)$ and push them down into yet another recursive routine? The reason is, unlike in $\mathbf{newfftS2}_N(x)$, we lack sufficient symmetry: because the scale factors are different for y_k and $y_{k+N/2}$, no single scale factor for u_k will save us that multiplication, nor can we apply the same scale factor to the $t_{N,k}z_k \pm t^*_{N,k}z'_k$ common sub-expressions. Thus, independent of any practical concerns about algorithm size, we currently see no arithmetic benefit to using more subroutines.

In some FFT applications, such as convolution with a fixed kernel, it is acceptable to compute a scaled DFT instead of the DFT, since any output scaling can be absorbed elsewhere at no cost. In this case, one would call $\mathbf{newfftS}_N(x)$ directly and save $M_S(N)$ multiplications over Yavne, where:

$$M_S(N) = \frac{2}{9}N\lg N - \frac{20}{27}N + \frac{2}{9}(-1)^{\lg N}\lg N - \frac{7}{27}(-1)^{\lg N} + 1, \quad (14)$$

with savings starting at $M_S(16)$= 4.

To verify these counts, as well as the correctness, accuracy, and other properties of our algorithm, we created a "toy" (slow) implementation of Algorithms 1–3, instrumented to count the number of real operations. (We checked it for correctness via [31], and for accuracy in Sec. V.) This implementation was also instrumented to check for any simple multiplications by ± 1, $\pm i$, and $a \cdot (1 \pm i)$, as well as for equal scale factors, that might have been missed in the above analysis, but we did not discover any such obvious opportunities for further savings. We have also implemented our new algorithm in the symbolic code-generation framework of FFTW [32], which takes the abstract algorithm as input, performs symbolic simplifications, and outputs optimized C code for a given fixed size. The generator also outputs a flop count that again verified Eq. (1), and the simplifier did not find any trivial optimizations that we missed; this code was again checked for correctness, and its performance is discussed in the concluding section below.

Finally, we should note that if one compares instead to split-radix with the 3/3 mult/add complex multiplies often used in earlier papers (which trade off some real multiplications for additions without changing the total flops), then our algorithm has slightly more multiplications and fewer additions (still beating the total flops by the same amount, of course). The reason is that the factored form of the multiplications in Algorithm 3 cannot, as far as we can tell, exploit the 3/3 trick to trade off multiplies for adds. In any case, this tradeoff no longer appears to be beneficial on CPUs with hardware multipliers (and especially those with fused multiply-adders).

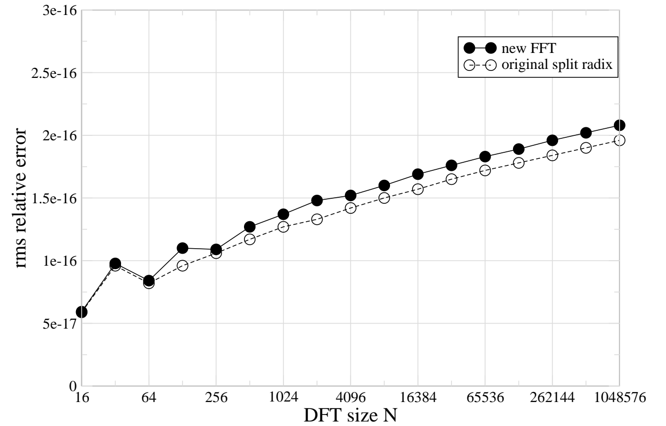


Fig. 2. Root-mean-square (L_2) relative error of our new FFT and the standard conjugate-pair split-radix FFT versus DFT size N, in 64-bit double precision.

V. Floating-point accuracy

In order to measure the accuracy of the new algorithm, we computed the L_2 (root-mean-square) relative error ($\sqrt{\sum|\Delta y_k|^2}/\sqrt{\sum|y_k|^2}$) of our "toy" implementation compared to the "exact" result (from an FFT implemented in arbitrary-precision arithmetic), for uniform pseudo-random inputs $x_n \in [-0.5, 0.5)$, in 64-bit double precision on a Pentium IV with Linux and gcc 3.3.5. The results, in Fig. 2, show that our new FFT has errors within 10% of the standard conjugate-pair split-radix algorithm, both growing roughly as $\sim \sqrt{\log N}$ [33]. At first glance, this may seem surprising, since our use of the tangent function, which is singular, or equivalently our division by a cosine in $1/s_{N,k}$, may appear to raise questions about the numerical accuracy of our algorithm. Although we have not performed a detailed numerical analysis, the reason for the similarity to standard split radix seems clear upon closer inspection: we *never add* scaled values with unscaled values, so that whenever standard split radix computes $a + b$ our new FFT merely computes $s \cdot (a + b)$ for some constant scale factor s. An alternative explanation might simply be that our scale factors are not very big, as described below, but we have checked this: changing Eq. (7) for $s_{N,k}$ to a less-symmetric form that always uses cos (and thus grows very small for $s_{N,N/4-1}$, *e.g.* reaching 10^{-25} for $N = 2^{20}$), the error varies by less than 10% from Fig. 2.

Another concern, nevertheless, might be simply that the scaling factor will grow so large/small as to induce over/underflow. This is not the case: the $1/s_{N,k}$ from Eq. (7) grows so much more slowly than the DFT values themselves (which grow as $\sim \sqrt{N}$ for random inputs) that over/underflow should not be significantly worsened by the new FFT algorithm. In particular, we explicitly avoided the cosine zero (at $k = N/4$) by the symmetric form of (7), so that its cosine (or sine) factor is always $\geq 1/\sqrt{2}$; thus, the loose bound $N^{-1/4} < s_{N,k} \leq 1$ follows. In fact, the smallest $k(m)$ where $s_{2^m,k(m)}$ is minimum apparently follows the integer sequence A007910 [34], which approaches $k(m) \to 2^m/10$, and thus $\min(s_{N,k}) \sim N^{\log_4 \cos(\pi/5)} \approx N^{-1/6.54}$ asymptotically. For example, with $N = 2^{20}$, the minimum scale factor is only $s_{2^{20},104858} \approx 0.133$.

It is instructive to contrast the present algorithm with the "real-factor" FFT algorithm that was once proposed to reduce the number of multiplications, but which proved numerically ill-behaved and was later surpassed by split radix [2], [35]. In that algorithm, one obtained an equation of the form $y_k = u_k - i\csc(2\pi k/N)c_k/2$ ($k \neq 0, N/2$), where u_k is the transform of x_{2n} (the even elements) and c_k is the transform of $x_{2n+1} - x_{2n-1}$ (the difference of adjacent odd elements). This reduces the number of real multiplications (matching standard split radix, albeit with more additions), but is numerically ill-behaved because of the singular csc function—unlike in our algorithm, c_k was *not* scaled by any sin function that would cancel the csc singularity, and thus the addition with the unscaled u_k exacerbates roundoff.

VI. Twiddle factors

In the standard conjugate-pair split-radix Algorithm 1, there is a redundancy in the twiddle factors between k and $N/4-k$: $\omega_N^{N/4-k} = -i\omega_N^{-k}$. This can be exploited to halve the number of twiddle factors that need to be computed (or loaded from a lookup table): ω_N^k is computed only for $k \leq N/8$, and for $N/8 < k < N/4$ it is found by the above identity via conjugation and multiplication by $-i$ (both of which are costless operations). This symmetry is preserved by the rescaling of our new algorithm, since $s_{N,N/4-k} = s_{N,k}$. Thus, for $N/8 < k < N/4$ we can share the (rescaled) twiddle factors with $k < N/8$. For example, in $\mathbf{newfftS}_N(x)$, we obtain $y_k = u_k - it^*_{N,k}z_k + it_{N,k}z'_k$ for $k > N/8$ (the operation count is unchanged, of course). The twiddle factors in $\mathbf{newfft}_N(x)$ are also redundant because $s_{N/4,N/4-k} = s_{N/4,k}$ (from the periodicity of $s_{N,k}$). For $\mathbf{newfftS2}_N(x)$, we have constants $s_{2N,k}$ and $s_{2N,k+N/4}$, and we use the fact that $s_{2N,N/4-k} = s_{2N,k+N/4}$ and $s_{2N,(N/4-k)+N/4} = s_{2N,k}$ so that the constants are shared between $k < N/8$ and $k > N/8$, albeit in reverse order. Similarly, for $\mathbf{newfftS4}_N(x)$, we have constants $s_{4N,k}$, $s_{4N,k+N/2}$, $s_{4N,k+N/4}$, and $s_{4N,k+3N/4}$; when $k \to N/4 - k$, these become $s_{4N,k+3N/4}$, $s_{4N,k+N/4}$, $s_{4N,k+N/2}$, and $s_{4N,k}$ respectively.

Despite these redundancies, our new FFT requires a larger number of distinct twiddle-factor–like constants to be computed or loaded than the standard conjugate-pair FFT algorithm, because of the differing scale factors in the four subroutines. It is difficult to make precise statements about the consequences of this fact, however, because the performance impact will depend on the implementation of the FFT, the layout of the pre-computed twiddle tables, the memory architecture, and the degree to which loads of twiddle factors can be overlapped with other operations in the CPU. Moreover, the access pattern is complex; for example, the $\mathbf{newfftS}_N(x)$ routine actually requires *fewer* twiddle constants than $\mathbf{splitfft}_N(x)$, since $t_{N,k}$ is only 1 nontrivial real constant vs. 2 for ω_N^k. Such practical concerns are discussed further in the concluding remarks.

A standard alternative to precomputed tables of twiddle constants is to generate them on the fly using an iterative recurrence relation of some sort (*e.g.* one crude method is $\omega_N^{k+1} = \omega_N^k \cdot \omega_N^1$), although this sacrifices substantial accuracy in the FFT unless sophisticated methods with $\Omega(\log N)$ storage are employed [36]. Because of the recursive nature of Eq. (7), however, it is not obvious to us how one might compute $s_{N,k+1}$ by a simple recurrence from $s_{N,k}$ or similar.

TABLE II
Flops of standard real-data split radix and our new algorithm

N	Real-data split radix	New algorithm
64	518	514
128	1286	1270
256	3078	3022
512	7174	7014
1024	16390	15962
2048	36870	35798
4096	81926	79334
8192	180230	174150
16384	393222	379250

VII. Real-data FFTs

For real inputs x_n, the outputs y_k obey the symmetry $y_{N-k} = y_k^*$ and one can save slightly more than a factor of two in flops when computing the DFT by eliminating the redundant calculations; practical implementations of this approach have been devised for many FFT algorithms, including a split-radix–based real-data FFT [37], [38] that achieved the best known flop count of $2N\lg N - 4N + 6$ for $N = 2^m$. The same elimination of redundancy applies to our algorithm, and thus we can lower the minimum flops required for the real-data FFT.

Because our algorithm only differs from standard split radix by scale factors that are purely real and symmetric, existing algorithms for the decimation-in-time split-radix FFT of real data [38] immediately apply: the number of additions is unchanged as for the complex algorithm, and the number of real multiplications is exactly half that of the complex algorithm. In particular, we save $M(N)/2$ multiplies compared to the previous algorithms. Thus, the flop count for a real-data FFT of length $N = 2^m$ is now:

$$\frac{17}{9}N\lg N - \frac{89}{27}N - \lg N - \frac{1}{9}(-1)^{\lg N}\lg N + \frac{8}{27}(-1)^{\lg N} + 6 \qquad (15)$$

for $N > 1$. To derive this more explicitly, note that each of the recursive sub-transforms in Algorithms 2–3 operates on real inputs x_n and thus has $y_{N-k} = y_k^*$ (a symmetry unaffected by the real scale factors, which satisfy $s_{\ell N,N-k} = s_{\ell N,k}$ for $\ell \leq 4$). Therefore, in the loop over k, the computation of $y_{k+N/2}$ and $y_{k+3N/4}$ is redundant and can be eliminated, saving half of $M(N)$ (in $M_S(N)$, *etc.*), except for $k = 0$ where $y_{N/2}$ is the real Nyquist element. For $k = 0$, we must compute both y_0 and $y_{N/2}$, but since these are *both* purely real we still save half of $M(N)$ (multiplying a real number by a real scale factor costs 1 multiply, *vs.* 2 multiplies for a complex number and a real scale factor). As for complex data, Eq. (15) yields savings over the standard split-radix method starting at $N = 64$, as summarized in Table II.

As mentioned above, we also implemented our complex-data algorithm in the code-generation program of FFTW, which performs symbolic-algebra simplifications that have

TABLE III

FLOPS REQUIRED FOR THE DISCRETE COSINE TRANSFORM (DCT) BY PREVIOUS ALGORITHMS AND BY OUR NEW ALGORITHM

N, DCT type	Previous best	New algorithm
16, DCT-II	112	110
32, DCT-II	288	282
64, DCT-II	704	684
128, DCT-II	1664	1612
8, DCT-IV	56	54
16, DCT-IV	144	140
32, DCT-IV	352	338
64, DCT-IV	832	800
128, DCT-IV	1920	1838
32, DCT-I	239	237
64, DCT-I	593	585
128, DCT-I	1427	1399

proved sufficiently powerful to automatically derive optimal-arithmetic real-data FFTs from the corresponding "optimal" complex-data algorithm—it merely imposes the appropriate input/output symmetries and prunes redundant outputs and computations [32]. Given our new FFT, we find that it can again automatically derive a real-data algorithm matching the predicted flop count of Eq. (15).

VIII. DISCRETE COSINE TRANSFORMS

Similarly, our new FFT algorithm can be specialized for DFTs of real-symmetric data, otherwise known as discrete cosine transforms (DCTs) of the various types [39] (and also discrete sine transforms for real-antisymmetric data). Moreover, since FFTW's generator can automatically derive algorithms for types I–IV of the DCT and DST [3], we have found that it can automatically realize arithmetic savings over the best-known DCT/DST implementations given our new FFT, as summarized in Table III.[6] Although here we exploit the generator and have not derived explicit general-N algorithms for the DCT flop count (except for type I), the same basic principles (expressing the DCT as a larger DFT with appropriate symmetries and pruning redundant computations from an FFT) have been applied to "manually" derive DCT algorithms in the past and we expect that doing so with the new algorithm will be straightforward. Below, we consider types II, III, IV, and I of the DCT.

A type-II DCT (often called simply "the" DCT) of length N is derived from a real-data DFT of length $4N$ with appropriate symmetries. Therefore, since our new algorithm begins to yield improvements starting at $N = 64$ for real/complex data, it yields an improved DCT-II starting at $N = 16$. Previously, a 16-point DCT-II with 112 flops was reported [11] for the (unnormalized) N-point DCT-II defined as:

$$y_k = x_0 + \sqrt{2}\sum_{n=1}^{N-1} x_n \cos\left[\frac{\pi(n+\frac{1}{2})k}{N}\right], \qquad (16)$$

whereas our generator now produces the same transform with only 110 flops. In general, $2N \lg N - N$ flops were required for this DCT-II [40], as can be derived from the standard split-radix approach [41] (and is also reproduced automatically by our generator starting from complex split radix), whereas the flop counts produced by our generator starting from our new FFT are given in Table III. The DCT-III (also called the "IDCT" since it inverts DCT-II) is simply the transpose of the DCT-II and its operation counts are identical.

It is also common to compute a DCT-II with scaled outputs, *e.g.* for the JPEG image-compression standard where an arbitrary scaling can be absorbed into a subsequent quantization step [42], and in this case the scaling can save 6 multiplications [11] over the 40 flops required for an unscaled 8-point DCT-II. Since our $\mathbf{newfftS}_N(x)$ attempts to be the optimal scaled FFT, we should be able to derive this scaled DCT-II by using it in the generator instead of $\mathbf{newfft}_N(x)$—indeed, we find that it does save exactly 6 multiplies over our unscaled result (after normalizing the DFT by an overall factor of $1/2$ due to the DCT symmetry). Moreover, we can now find the corresponding scaled transforms of larger sizes: *e.g.* 96 flops for a size-16 scaled DCT-II, and 252 flops for size 32, saving 14 and 30 flops, respectively, compared to the unscaled transform above.

For the DCT-IV, which is the basis of the modified discrete cosine transform (MDCT) [43], the corresponding symmetric DFT is of length $8N$, and thus the new algorithm yields savings starting at $N = 8$: the best (split-radix) methods for an 8-point DCT-IV require 56 flops (or $2N \lg N + N$ [41]) for the DCT-IV defined by

$$y_k = \sqrt{2}\sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi(n+\frac{1}{2})(k+\frac{1}{2})}{N}\right], \qquad (17)$$

whereas the new algorithm requires 54 flops for $N = 8$ (as derived by our generator), with other sizes shown in Table III.

Finally, a type-I DCT of length N (with $N+1$ data points) defined as

$$y_k = x_0 + (-1)^k x_N + 2\sum_{n=1}^{N-1} x_n \cos\left[\frac{\pi n k}{N}\right] \qquad (18)$$

is exactly equivalent to a DFT of length $2N$ where the input data are real-symmetric ($x_{2N-n} = x_n$), and the split-radix FFT adapted for this symmetry requires $2N \lg N - 3N + 2\lg N + 5$ flops [37].[7] Because the scale factors $s_{N,k}$ preserve this symmetry, one can employ exactly the same approach to save $M(2N)/4$ multiplications starting from our new FFT (proof is identical). Indeed, precisely these savings are derived by the FFTW generator for the first few N, as shown in Table III.

IX. CONCLUDING REMARKS

The longstanding arithmetic record of Yavne for the power-of-two DFT has been broken, but at least two important questions remain unanswered. First, can one do better still? Second, will the new algorithm result in practical improvements to actual computation times for the FFT?

[6] The precise multiplication count for a DCT generally depends upon the normalization convention that is chosen; here, we use the same normalizations as the references cited for comparison.

[7] Our count is slightly modified from that of Duhamel [37], who omitted all multiplications by 2 from the flops.

Since this algorithm represents a simple transformation applied to the existing split-radix FFT, a transformation that has obviously been insufficiently explored in the past four decades of FFT research, it may well be that further gains can be realized by applying similar ideas to other algorithms or by extending these transformations to greater generality. One avenue to explore is the *automatic* application of such ideas—is there a simple algebraic transformational rule that, when applied recursively in a symbolic FFT-generation program [32], [44], can derive automatically the same (or greater) arithmetic savings? (Note that both our own code generation and that of Van Buskirk currently require explicit knowledge of a rescaled FFT algorithm.) Moreover, a new fundamental question is to find the lowest-arithmetic *scaled* DFT—our current best answer is $\mathbf{newfftS}_N(x)$ and Eq. (14), but any improvement will also improve the unscaled DFT.

The question of practical impact is even harder to answer, because the question is not very well defined—the "fastest" algorithm depends upon what hardware is running it. For large N, however, it is likely that the split-radix algorithm here will have to be substantially modified in order to be competitive, since modern architectures tend to favor much larger radices combined with other tricks to placate the memory hierarchy [3]. (Unless similar savings can be realized directly for higher radices [45], this would mean "unrolling" or "blocking" the decomposition of N so that several subdivisions are performed at once.) On the other hand, for small N, which can form the computational "kernels" of general-N FFTs, we already use the original conjugate-pair split-radix algorithm in FFTW [32] and can immediately compare the performance of these kernels with ones generated from the new algorithm. We have not yet performed extensive benchmarking, however, and the results of our limited tests are somewhat difficult to assess. On a 2GHz Pentium-IV with gcc, the performance was indistinguishable for the DFT of size 64 or 128, but the new algorithm was up to 10% faster for the DCT-II and IV of small sizes—a performance difference greater than the change in arithmetic counts, leading us to suspect some fortuitous interaction with the code scheduling. Nevertheless, it is precisely because practical performance is so unpredictable that the availability of new algorithms, especially ones with reasonably regular structure amenable to implementation, opens up rich areas for future experimentation.

Acknowledgement

We are grateful to James Van Buskirk for helpful discussions, and for sending us his code when we were somewhat skeptical of his initial claim.

References

[1] D. E. Knuth, *Fundamental Algorithms*, 3rd ed., ser. The Art of Computer Programming. Addison-Wesley, 1997, vol. 1.

[2] P. Duhamel and M. Vetterli, "Fast Fourier transforms: a tutorial review and a state of the art," *Signal Processing*, vol. 19, pp. 259–299, Apr. 1990.

[3] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[4] R. Yavne, "An economical method for calculating the discrete Fourier transform," in *Proc. AFIPS*, vol. 33, 1968, pp. 115–125.

[5] P. Duhamel and H. Hollmann, "Split-radix FFT algorithm," *Electron. Lett.*, vol. 20, no. 1, pp. 14–16, 1984.

[6] M. Vetterli and H. J. Nussbaumer, "Simple FFT and DCT algorithms with reduced number of operations," *Signal Processing*, vol. 6, no. 4, pp. 267–278, 1984.

[7] J. B. Martens, "Recursive cyclotomic factorization—a new algorithm for calculating the discrete Fourier transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 32, no. 4, pp. 750–761, 1984.

[8] J. W. Cooley and J. W. Tukey, "An algorithm for the machine computation of the complex Fourier series," *Math. Computation*, vol. 19, pp. 297–301, Apr. 1965.

[9] D. E. Knuth, *Seminumerical Algorithms*, 3rd ed., ser. The Art of Computer Programming. Addison-Wesley, 1998, vol. 2, section 4.6.4, exercise 41.

[10] J. Van Buskirk, `comp.dsp` Usenet posts, January 2004.

[11] Y. Arai, T. Agui, and M. Nakajima, "A fast DCT-SQ scheme for images," *Trans. IEICE*, vol. 71, no. 11, pp. 1095–1097, 1988.

[12] J. Van Buskirk, `http://home.comcast.net/~kmbtib/` and `http://www.cuttlefisharts.com/newfft/`, 2004.

[13] T. J. Lundy and J. Van Buskirk, "A new matrix approach to real FFTs and convolutions of length 2^k," *IEEE Trans. Signal Processing*, 2006, submitted for publication.

[14] W. M. Gentleman and G. Sande, "Fast Fourier transforms—for fun and profit," *Proc. AFIPS*, vol. 29, pp. 563–578, 1966.

[15] S. Winograd, "On computing the discrete Fourier transform," *Math. Computation*, vol. 32, no. 1, pp. 175–199, Jan. 1978.

[16] M. T. Heideman and C. S. Burrus, "On the number of multiplications necessary to compute a length-2^n DFT," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 34, no. 1, pp. 91–95, 1986.

[17] P. Duhamel, "Algorithms meeting the lower bounds on the multiplicative complexity of length-2^n DFTs and their connection with practical algorithms," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 38, no. 9, pp. 1504–1511, 1990.

[18] J. Morgenstern, "Note on a lower bound of the linear complexity of the fast Fourier transform," *J. ACM*, vol. 20, no. 2, pp. 305–306, 1973.

[19] V. Y. Pan, "The trade-off between the additive complexity and the asyncronicity of linear and bilinear algorithms," *Information Proc. Lett.*, vol. 22, pp. 11–14, 1986.

[20] C. H. Papadimitriou, "Optimality of the fast Fourier transform," *J. ACM*, vol. 26, no. 1, pp. 95–102, 1979.

[21] I. Kamar and Y. Elcherif, "Conjugate pair fast Fourier transform," *Electron. Lett.*, vol. 25, no. 5, pp. 324–325, 1989.

[22] R. A. Gopinath, "Comment: Conjugate pair fast Fourier transform," *Electron. Lett.*, vol. 25, no. 16, p. 1084, 1989.

[23] H.-S. Qian and Z.-J. Zhao, "Comment: Conjugate pair fast Fourier transform," *Electron. Lett.*, vol. 26, no. 8, pp. 541–542, 1990.

[24] A. M. Krot and H. B. Minervina, "Comment: Conjugate pair fast Fourier transform," *Electron. Lett.*, vol. 28, no. 12, pp. 1143–1144, 1992.

[25] D. J. Bernstein, `http://cr.yp.to/djbfft/faq.html`, 1997, see note on "exponent -1 split-radix".

[26] H. V. Sorensen, M. T. Heideman, and C. S. Burrus, "On computing the split-radix FFT," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 34, no. 1, pp. 152–156, Feb. 1986.

[27] C. van Loan, *Computational Frameworks for the Fast Fourier Transform*. Philadelphia: SIAM, 1992.

[28] R. E. Crochiere and A. V. Oppenheim, "Analysis of linear digital networks," *Proc. IEEE*, vol. 63, no. 4, pp. 581–595, 1975.

[29] E. Linzer and E. Feig, "Modified FFTs for fused multiply-add architectures," *Math. Comp.*, vol. 60, no. 201, pp. 347–361, 1993.

[30] G. H. Golub and C. F. van Loan, *Matrix Computations*. Johns Hopkins Univ. Press, 1989.

[31] F. Ergün, "Testing multivariate linear functions: Overcoming the generator bottleneck," in *Proc. Twenty-Seventh Ann. ACM Symp. Theory of Computing*, Las Vegas, Nevada, June 1995, pp. 407–416.

[32] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, vol. 34, no. 5. Atlanta, Georgia: ACM, May 1999, pp. 169–180.

[33] J. C. Schatzman, "Accuracy of the discrete Fourier transform and the fast Fourier transform," *SIAM J. Scientific Computing*, vol. 17, no. 5, pp. 1150–1166, 1996.

[34] N. J. A. Sloane, "The on-line encyclopedia of integer sequences," `http://www.research.att.com/~njas/sequences/`.

[35] C. M. Rader and N. M. Brenner, "A new principle for fast Fourier transformation," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 24, pp. 264–265, 1976.

[36] M. Tasche and H. Zeuner, “Improved roundoff error analysis for precomputed twiddle factors,” *J. Comput. Anal. Appl.*, vol. 4, no. 1, pp. 1–18, 2002.

[37] P. Duhamel, “Implementation of “split-radix” FFT algorithms for complex, real, and real-symmetric data,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 34, no. 2, pp. 285–295, 1986.

[38] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus, “Real-valued fast Fourier transform algorithms,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 35, no. 6, pp. 849–863, June 1987.

[39] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications*. Boston, MA: Academic Press, 1990.

[40] G. Plonka and M. Tasche, “Fast and numerically stable algorithms for discrete cosine transforms,” *Linear Algebra Appl.*, vol. 394, pp. 309–345, 2005.

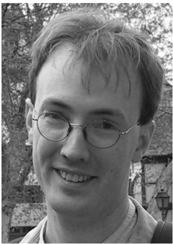
[41] ——, “Split-radix algorithms for discrete trigonometric transforms,” `http://citeseer.ist.psu.edu/600848.html`.

[42] W. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*. New York: Van Nostrand Reinhold, 1993.

[43] H. S. Malvar, *Signal Processing with Lapped Transforms*. Norwood, MA: Artech House, 1992.

[44] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “SPIRAL: Code generation for DSP transforms,” *Proc. IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[45] S. Bouguezel, M. O. Ahmad, and M. N. S. Swamy, “A new radix-2/8 FFT algorithm for length-$q \times 2^m$ DFTs,” *IEEE Trans. Circuits Syst. I*, vol. 51, no. 9, pp. 1723–1732, 2004.



Steven G. Johnson joined the faculty of Applied Mathematics at the Massachusetts Institute of Technology (MIT) in 2004. He received his Ph. D. in 2001 from the Dept. of Physics at MIT, where he also received undergraduate degrees in computer science and mathematics. His recent work, besides FFTs, has focused on the theory of photonic crystals: electromagnetism in nano-structured media. This has ranged from general research in semi-analytical and numerical methods for electromagnetism, to the design of integrated optical devices, to the development of optical fibers that guide light within an air core to circumvent limits of solid materials. His Ph. D. thesis was published as a book, *Photonic Crystals: The Road from Theory to Practice*, in 2002.

Joint recipient, with Matteo Frigo, of the 1999 *J. H. Wilkinson Prize for Numerical Software*, in recognition of their work on FFTW.

Matteo Frigo received his Ph. D. in 1999 from the Dept. of Electrical Engineering and Computer Science at MIT, and is currently with the IBM Austin Research Laboratory. Besides FFTW, his research interests include the theory and implementation of Cilk (a multithreaded system for parallel programming), cache-oblivious algorithms, and software radios. In addition, he has implemented a gas analyzer that is used for clinical tests on lungs.