

Type-II/III DCT/DST algorithms with reduced number of arithmetic operations

Xuancheng Shao and Steven G. Johnson*

Department of Mathematics, Massachusetts Institute of Technology, Cambridge MA 02139

Abstract

We present algorithms for the discrete cosine transform (DCT) and discrete sine transform (DST), of types II and III, that achieve a lower count of real multiplications and additions than previously published algorithms, without sacrificing numerical accuracy. Asymptotically, the operation count is reduced from $\sim 2N \log_2 N$ to $\sim \frac{17}{9} N \log_2 N$ for a power-of-two transform size N . Furthermore, we show that a further N multiplications may be saved by a certain rescaling of the inputs or outputs, generalizing a well-known technique for $N = 8$ by Arai et al. These results are derived by considering the DCT to be a special case of a DFT of length $4N$, with certain symmetries, and then pruning redundant operations from a recent improved fast Fourier transform algorithm (based on a recursive rescaling of the conjugate-pair split radix algorithm). The improved algorithms for DCT-III, DST-II, and DST-III follow immediately from the improved count for the DCT-II.

Key words: discrete cosine transform; fast Fourier transform; arithmetic complexity

1. Introduction

In this paper, we describe recursive algorithms for the type-II and type-III discrete cosine and sine transforms (DCT-II and DCT-III, and also DST-II and DST-III), of power-of-two sizes, that require fewer total real additions and multiplications than previously published algorithms (with an asymptotic reduction of about 6%), without sacrificing numerical accuracy. Our DCT and DST algorithms are based on a recently published fast Fourier transform (FFT) algorithm, which reduced the operation count for the discrete Fourier transform (DFT) compared to the previous-best split-radix algorithm [1]. The gains in this new FFT algorithm, and consequently in the new DCTs and DSTs, stem from a recursive rescaling of the internal multiplicative factors within an algorithm called a “conjugate-pair” split-radix FFT [2–5] so as to simplify some of the multiplications. In order to derive a DCT algorithm from this FFT, we simply consider the DCT-II to be a special case of a DFT with real input of a certain even symmetry, and discard the redundant operations [1, 6–10]. The DCT-III, DST-II, and DST-III have identical opera-

tion counts to the DCT-II of the same size, since the algorithms are related by simple transpositions, permutations, and sign flips [11–13].

Since 1968, the lowest total count of real additions and multiplications, herein called “flops” (floating-point operations), for the DFT of a power-of-two size $N = 2^m$ was achieved by the split-radix algorithm, with $4N \log_2 N - 6N + 8$ flops for $N > 1$ [6, 8, 14–16]. This count was recently surpassed, as described in Ref. [1], at first in unpublished results by Van Buskirk *et al.* for $N = 64$ and later generalized to any power-of-two N , leading to a flop count asymptotically proportional to $\frac{34}{9} N \log_2 N$. Similarly, the lowest-known flop count for the DCT-II of size $N = 2^m > 1$ was previously $2N \log_2 N - N + 2$ for a unitary normalization (with the additive constant depending on normalization) [6, 7, 12, 13, 17–25], and could be achieved by starting with the split-radix FFT and discarding redundant operations [6, 7]. (Many DCT algorithms with an unreported or larger flop count have also been described [26–37].) Based on the new FFT algorithm, the flop counts for the various DCT types were reduced using a code generator [10, 38] that automatically pruned the redundant operations from an FFT with a given symmetry, but neither an explicit algorithm nor a general formula for the flop count were presented [1]. In this paper, we use the same starting point to “manually” derive a DCT-II algorithm by pruning redun-

* Corresponding author. Address: 77 Massachusetts Ave. Rm. 2-388, Cambridge, MA 02139. Email: stevenj@math.mit.edu. Tel: 617-253-4073. Fax: 617-253-8911.

N	previous DCT-II	New algorithm
16	114	112
32	290	284
64	706	686
128	1666	1614
256	3842	3708
512	8706	8384
1024	19458	18698
2048	43010	41266
4096	94210	90264

Table 1
Flop counts (real adds + mults) of previous best DCT-II and our new algorithm

dant operations from a real-even FFT, and give the general formula for the new flop count (for $N = 2^m > 1$):

$$\frac{17}{9}N \log_2 N - \frac{17}{27}N - \frac{1}{9}(-1)^{\log_2 N} \log_2 N + \frac{7}{54}(-1)^{\log_2 N} + \frac{3}{2}. \quad (1)$$

The first savings over the previous record occur for $N = 16$, and are summarized in Table 1 for several N . We also consider the effect of normalization on this flop count: the above count was for a unitary transform, but slightly different counts are obtained by other choices. Moreover, we show that a further N multiplications can be saved by individually rescaling *every* output of the DCT-II (or input of the DCT-III). In doing so, we generalize a result by Arai *et al.*, who showed that eight multiplications could be saved by scaling the outputs of a DCT-II of size $N = 8$ [39], a result commonly applied to JPEG compression [40].

In the following sections, we first review how a DCT-II may be expressed as a special case of a DFT, and how the previous optimum flop count can be achieved by pruning redundant operations from a conjugate-pair split-radix FFT. Then, we briefly review the new FFT algorithm presented in [1], and derive the new DCT-II algorithm. We follow by considering the effect of normalization and scaling. Finally, we consider the extension of this algorithm to algorithms for the DCT-III, DST-II, and DST-III. We close with some concluding remarks about future directions. We emphasize that no *proven* tight lower bound on the DCT-II flop count is currently known, and we make no claim that eq. (1) is the lowest possible.

2. DCT-II from DFT

Various forms of discrete cosine transform have been defined, corresponding to different boundary conditions on the transform [41]. Perhaps the most common form is the type-II DCT, used in image compression [40] and many other applications. The DCT-II is typically defined as a real, orthogonal (unitary), linear transformation by the formula:

$$C_k^{\text{II}} = \sqrt{\frac{2 - \delta_{k,0}}{N}} \sum_{n=0}^{N-1} x_n \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right], \quad (2)$$

for N inputs x_n and N outputs C_k^{II} , where $\delta_{k,0}$ is the Kronecker delta ($= 1$ for $k = 0$ and $= 0$ otherwise). However, we wish to emphasize in this paper that the DCT-II (and, indeed, all types of DCT) can be viewed as special cases of the discrete Fourier transform (DFT) with real inputs of a certain symmetry. This (well known) viewpoint is fruitful because it means that any FFT algorithm for the DFT leads immediately to a corresponding fast algorithm for the DCT-II simply by discarding the redundant operations [1, 6–10].

The discrete Fourier transform of size N is defined by

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{nk}, \quad (3)$$

where $\omega_N = e^{-\frac{2\pi i}{N}}$ is an N th primitive root of unity. In order to relate this to the DCT-II, it is convenient to choose a different normalization for the latter transform:

$$C_k = 2 \sum_{n=0}^{N-1} x_n \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right]. \quad (4)$$

This normalization is not unitary, but it is more directly related to the DFT and therefore more convenient for the development of algorithms. Of course, any fast algorithm for C_k trivially yields a fast algorithm for C_k^{II} , although the exact count of required multiplications depends on the normalization, an issue we discuss in more detail in section 6.

In order to derive C_k from the DFT formula, one can use the identity $2 \cos(\pi \ell / N) = \omega_{4N}^{2\ell} + \omega_{4N}^{4N-2\ell}$ to write:

$$\begin{aligned} C_k &= 2 \sum_{n=0}^{N-1} x_n \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right] \\ &= \sum_{n=0}^{N-1} x_n \omega_{4N}^{(2n+1)k} + \sum_{n=0}^{N-1} x_n \omega_{4N}^{(4N-2n-1)k} \\ &= \sum_{n=0}^{4N-1} \tilde{x}_n \omega_{4N}^{nk}, \end{aligned} \quad (5)$$

where \tilde{x}_n is a real-even sequence of length $4N$, defined as follows for $0 \leq n < N$:

$$\tilde{x}_{2n} = \tilde{x}_{4N-2n-2} = 0, \quad (6)$$

$$\tilde{x}_{2n+1} = \tilde{x}_{4N-(2n+1)} = x_n. \quad (7)$$

Thus, the DCT-II of size N is precisely a DFT of size $4N$, of real-even inputs, where the even-indexed inputs are zero.

This is illustrated by an example, for $N = 8$, in figure 1. The eight inputs of the DCT are shown as open dots, which are interleaved with zeros (black dots) and extended to an even (squares) periodic (gray dots) sequence of length $4N = 32$ corresponding to the DFT. (The type-II DCT is distinguished from the other DCT types by the fact that it is even about both the left and right boundaries of the original

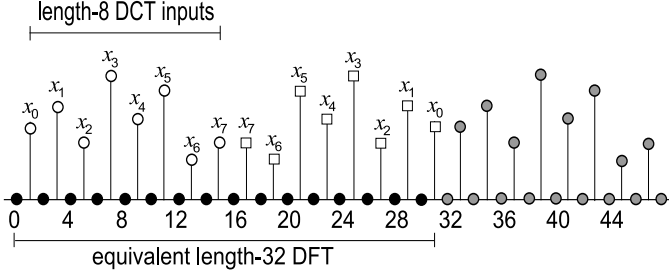


Figure 1. A DCT-II of size $N = 8$ (open dots, x_0, \dots, x_7) is equivalent to a size- $4N$ DFT via interleaving with zeros (black dots) and extending in an even (squares) periodic (gray) sequence.

data, and the symmetry points fall halfway in between pairs of the original data points.) We will refer, below, to this figure in order to illustrate what happens when an FFT algorithm is applied to this real-symmetric zero-interleaved data.

3. Conjugate-pair FFT and DCT-II

Although the previous minimum flop count for DCT-II algorithms has been derived from the ordinary split-radix FFT algorithm [6, 7], here we will do the same thing using variant dubbed the “conjugate-pair” split-radix FFT, which was originally proposed to reduce the number of flops [2], but was later shown to have the same flop count as ordinary split-radix [3–5]. It turns out, however, that the conjugate-pair algorithm exposes symmetries in the multiplicative factors which can be exploited to reduce the flop count by an appropriate rescaling [1], which we will employ in the following sections.

Starting with the DFT of equation (3), the decimation-in-time conjugate-pair FFT splits it into three smaller DFTs: one of size $N/2$ of the even-indexed inputs, and two of size $N/4$:

$$X_k = \sum_{n_2=0}^{N/2-1} \omega_{N/2}^{n_2 k} x_{2n_2} + \omega_N^k \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k} x_{4n_4+1} + \omega_N^{-k} \sum_{n_4=0}^{N/4-1} \omega_{N/4}^{n_4 k} x_{4n_4-1}. \quad (8)$$

[In contrast, the ordinary split-radix FFT uses x_{4n_4+3} for the third sum (a cyclic shift of x_{4n_4-1}), with a corresponding multiplicative “twiddle” factor of ω_N^{3k} .] This decomposition is repeated recursively (until base cases of size $N = 1$ or $N = 2$ are reached), as shown by the pseudo-code in Algorithm 1. Here, we denote the results of the three sub-transforms of size $N/2$, $N/4$, and $N/4$ by U_k , Z_k , and Z'_k , respectively. The number of flops required by this algorithm, after certain simplifications (common subexpression elimination and constant folding) and not counting data-independent operations like the computation of ω_N^k , is $4N \log_2 N - 6N + 8$, identical to ordinary split radix.

In the following sections, we will also have to exploit further simplifications due to the symmetries of the twiddle

Algorithm 1 Standard conjugate-pair split-radix FFT algorithm of size N . Special-case optimizations for $k = 0$ and $k = N/8$, and the base cases, are omitted for simplicity.

```

function  $X_{k=0..N-1} \leftarrow \text{splitfft}_N(x_n)$ :
   $U_{k_2=0..N/2-1} \leftarrow \text{splitfft}_{N/2}(x_{2n_2})$ 
   $Z_{k_4=0..N/4-1} \leftarrow \text{splitfft}_{N/4}(x_{4n_4+1})$ 
   $Z'_{k_4=0..N/4-1} \leftarrow \text{splitfft}_{N/4}(x_{4n_4-1})$ 
  for  $k = 0$  to  $N/4 - 1$  do
     $X_k \leftarrow U_k + (\omega_N^k Z_k + \omega_N^{-k} Z'_k)$ 
     $X_{k+N/2} \leftarrow U_k - (\omega_N^k Z_k + \omega_N^{-k} Z'_k)$ 
     $X_{k+N/4} \leftarrow U_{k+N/4} - i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)$ 
     $X_{k+3N/4} \leftarrow U_{k+N/4} + i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)$ 
  end for

```

Algorithm 2 Standard conjugate-pair split-radix FFT algorithm of size N , as in Algorithm 1 but explicitly showing the eight terms that share the same twiddle factor $\omega_N^{\pm k}$. Special-case optimizations for $k = 0$ and $k = N/8$, and the base cases, are omitted for simplicity.

```

function  $X_{k=0..N-1} \leftarrow \text{splitfft}_N(x_n)$ :
   $U_{k_2=0..N/2-1} \leftarrow \text{splitfft}_{N/2}(x_{2n_2})$ 
   $Z_{k_4=0..N/4-1} \leftarrow \text{splitfft}_{N/4}(x_{4n_4+1})$ 
   $Z'_{k_4=0..N/4-1} \leftarrow \text{splitfft}_{N/4}(x_{4n_4-1})$ 
  for  $k = 0$  to  $N/8$  do
     $X_k \leftarrow U_k + (\omega_N^k Z_k + \omega_N^{-k} Z'_k)$ 
     $X_{k+N/2} \leftarrow U_k - (\omega_N^k Z_k + \omega_N^{-k} Z'_k)$ 
     $X_{k+N/4} \leftarrow U_{k+N/4} - i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)$ 
     $X_{k+3N/4} \leftarrow U_{k+N/4} + i(\omega_N^k Z_k - \omega_N^{-k} Z'_k)$ 
     $X_{N/4-k} \leftarrow U_{N/4-k}$ 
     $X_{3N/4-k} \leftarrow U_{N/4-k}$ 
     $X_{N/2-k} \leftarrow U_{N/2-k}$ 
     $X_{N-k} \leftarrow U_{N/2-k}$ 
     $-i(\omega_N^{-k} Z_{N/4-k} - \omega_N^k Z'_{N/4-k})$ 
     $+i(\omega_N^{-k} Z_{N/4-k} - \omega_N^k Z'_{N/4-k})$ 
     $-(\omega_N^{-k} Z_{N/4-k} + \omega_N^k Z'_{N/4-k})$ 
     $+(\omega_N^{-k} Z_{N/4-k} + \omega_N^k Z'_{N/4-k})$ 
  end for

```

factors: $\omega_N^{\pm(N/4-k)} = \mp i \omega_N^{\mp k}$. This allows us to combine the X_k summation into eight terms sharing the same twiddle factor $\omega_N^{\pm k}$ between k and $N/4 - k$. This sharing of twiddle factors is shown explicitly in Algorithm 2, which otherwise describes the same conjugate-pair FFT algorithm.

If the inputs x_n are purely real, then $X_k = X_{-k}^*$ and one can save slightly more than half of the flops, both in ordinary split-radix [7, 42] and in the conjugate-pair split-radix [1], by eliminating redundant operations, to achieve a flop count of $2N \log_2 N - 4N + 6$. In particular note that in Algorithm 2, the sub-transforms also operate on real inputs, and therefore $Z_{N/4-k} = Z_k^*$, $Z'_{N/4-k} = Z'_k^*$, and $U_{N/2-k} = U_k^*$ —this makes every other the statement in the loop redundant.

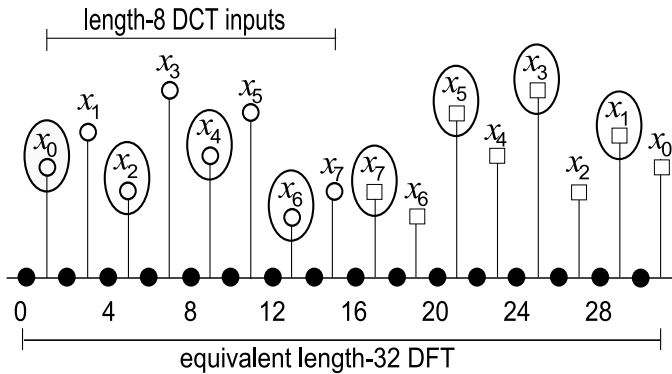


Figure 2. The DCT-II of $N = 8$ points x_n (open dots) is computed, in a conjugate-pair FFT of the $\tilde{N} = 32$ extended data \tilde{x}_n (squares) from figure 1, via the DFT Z_k of the circled points \tilde{x}_{4n+1} , corresponding to the even-indexed x_n followed by the odd-indexed x_n .

3.1. Fast DCT-II from split-radix

Before we derive our fast DCT-II with a reduced flop count, we first derive a DCT-II algorithm with the *same* flop count as previously published algorithms, but starting with the conjugate-pair split-radix algorithm. This algorithm will then be modified in section 5, below, to reduce the number of multiplications.

In eq. (5), we showed that a DCT-II of size N , with inputs x_n and outputs C_k , can be expressed as a DFT of size $\tilde{N} = 4N$ of real-even inputs \tilde{x}_n . Now, consider what happens when we evaluate this DFT via the conjugate-pair split decomposition in eq. (8) and Algorithm 2. In this algorithm, we compute three smaller DFTs. First, U_k is the DFT of size $\tilde{N}/2 = 2N$ of the \tilde{x}_{2n} inputs, but these are all zero and so U_k is *zero*. Second, Z_k is the DFT of size $\tilde{N}/4 = N$ of the \tilde{x}_{4n+1} inputs, which by eq. (7) correspond to the original data x_n by the formula:

$$\tilde{y}_n = \tilde{x}_{4n+1} = \begin{cases} x_{2n} & 0 \leq n < N/2 \\ x_{2N-1-2n} & N/2 \leq n < N \end{cases}, \quad (9)$$

where we have denoted them by \tilde{y}_n ($0 \leq n < N$) for convenience. That is, Z_k is the *real-input* DFT of the even elements of x_n followed by the odd elements of x_n in reverse order. For example, this is shown for $N = 8$ in figure 2 with the circled points corresponding to the \tilde{y}_n , which are clearly the even-indexed x_n followed by the odd-indexed x_n in reverse. The second DFT, Z'_k , of size $\tilde{N}/4 = N$ is redundant: it is the DFT of \tilde{x}_{4n-1} , but by the even symmetry of \tilde{x}_n this is equal to $\tilde{x}_{4(-n)+1}$, and therefore $Z'_k = Z_k^*$ (the complex conjugate of Z_k).

Therefore, a fast DCT-II is obtained merely by computing a *single* real-input DFT of size N to obtain $Z_k = Z_{N-k}^*$, and then combining this according to Algorithm 2 to obtain $C_k = X_k$ for $k = 0 \dots N-1$. In particular, in the loop of Algorithm 2, all but the X_k and $X_{\tilde{N}/4-k}$ terms correspond to subscripts $\geq N$ and are not needed. Moreover the $\omega_{\tilde{N}}^k Z_k + \omega_{\tilde{N}}^{-k} Z_k^*$ (for X_k) and $i(\omega_{\tilde{N}}^{-k} Z_{\tilde{N}/4-k} - \omega_{\tilde{N}}^k Z_{\tilde{N}/4-k}^*) = -i(\omega_{4N}^k Z_k - \omega_{4N}^{-k} Z_k^*)$ (for X_{N-k}) terms correspond to twice

Algorithm 3 Fast DCT-II algorithm, matching previous best flop count, derived from Algorithm 2 by discarding redundant operations.

function $C_{k=0..N-1} \leftarrow \text{splitdctII}_N(x_n)$:

for $n = 0$ to $N/2 - 1$ **do**

$\tilde{y}_n \leftarrow x_{2n}$

$\tilde{y}_{N-1-n} \leftarrow x_{2n+1}$

end for

$Z_{k=0..N-1} \leftarrow \text{splitfft}_N(\tilde{y}_n)$

$C_0 \leftarrow 2Z_0$

for $k = 1$ to $N/2 - 1$ **do**

$C_k \leftarrow 2\Re(\omega_{4N}^k Z_k)$

$C_{N-k} \leftarrow -2\Im(\omega_{4N}^k Z_k)$

end for

$C_{N/2} \leftarrow \sqrt{2}Z_{N/2}$

the real and imaginary parts of $\omega_{4N}^k Z_k$, respectively. The resulting algorithm, including the special-case optimizations for $k = 0$ (where $\omega_{4N}^0 = 1$ and Z_0 is real) and $k = N/2$ (where $\omega_{4N}^{N/2} = (1-i)/\sqrt{2}$ and $Z_{N/2}$ is real), is shown in Algorithm 3.

In fact, Algorithm 3 is equivalent to an algorithm derived in a different way by Makhoul [30] to express a DCT-II in terms of a real-input DFT of the same length. Here, we see that this algorithm is exactly equivalent to a conjugate-pair split-radix FFT on the “logical” DFT of length $4N$. Makhoul obtained a suboptimal flop count $\sim \frac{5}{2}N \log_2 N$ only because he used a suboptimal real-input DFT (split-radix being then almost unknown). Using a real-input split-radix DFT to compute Z_k , the flop count for Z_k is $2N \log_2 N - 4N + 6$ from above. To get the total flop count for Algorithm 3, we need to add $N/2 - 1$ general complex multiplications by $2\omega_{4N}^k$ (6 flops each) plus two real multiplications (2 flops), for a total of $2N \log_2 N - N + 2$ flops. This matches the best-known flop count in the literature (where the +2 can be removed merely by choosing a different normalization as discussed in section 6).

4. New FFT

Based on the conjugate-pair split-radix FFT from section 3, a new FFT algorithm with a reduced number of flops can be derived by scaling the subtransforms [1]. We will not reproduce the derivation here, but will simply summarize the results. In particular, the original conjugate-pair split-radix Algorithm 1 is split into four mutually recursive algorithms, each of which has the same split-radix structure but computes a DFT scaled by a different factor. These algorithms are shown in Algorithms 4–5, in which the scaling factors are combined with the twiddle factors ω_N^k to reduce the total number of multiplications.

The key aspect of these algorithms is the scale factor $s_{N,k}$, where the subtransforms compute the DFT scaled by $1/s_{\ell N,k}$ for $\ell = 1, 2, 4$. This scale factor is defined for $N = 2^m$ by the following recurrence, where $k_4 = k \bmod \frac{N}{4}$:

Algorithm 4 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 5 (four recursive functions in all, which differ in their rescalings).

function $X_{k=0..N-1} \leftarrow \mathbf{newfft}_N(x_n)$:
 {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**
 $X_k \leftarrow U_k + (\omega_N^k s_{N/4,k} Z_k + \omega_N^{-k} s_{N/4,k} Z'_k)$
 $X_{k+N/2} \leftarrow U_k - (\omega_N^k s_{N/4,k} Z_k + \omega_N^{-k} s_{N/4,k} Z'_k)$
 $X_{k+N/4} \leftarrow U_{k+N/4}$
 $\quad - i (\omega_N^k s_{N/4,k} Z_k - \omega_N^{-k} s_{N/4,k} Z'_k)$
 $X_{k+3N/4} \leftarrow U_{k+N/4}$
 $\quad + i (\omega_N^k s_{N/4,k} Z_k - \omega_N^{-k} s_{N/4,k} Z'_k)$
end for

function $X_{k=0..N-1} \leftarrow \mathbf{newfftS}_N(x_n)$:
 {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**
 $X_k \leftarrow U_k + (t_{N,k} Z_k + t_{N,k}^* Z'_k)$
 $X_{k+N/2} \leftarrow U_k - (t_{N,k} Z_k + t_{N,k}^* Z'_k)$
 $X_{k+N/4} \leftarrow U_{k+N/4} - i (t_{N,k} Z_k - t_{N,k}^* Z'_k)$
 $X_{k+3N/4} \leftarrow U_{k+N/4} + i (t_{N,k} Z_k - t_{N,k}^* Z'_k)$
end for

$$s_{N=2^m,k} = \begin{cases} 1 & \text{for } N \leq 4 \\ s_{N/4,k_4} \cos(2\pi k_4/N) & \text{for } k_4 \leq N/8 \\ s_{N/4,k_4} \sin(2\pi k_4/N) & \text{otherwise} \end{cases} \cdot (10)$$

This definition has the properties: $s_{N,0} = 1$, $s_{N,k+N/4} = s_{N,k}$, and $s_{N,N/4-k} = s_{N,k}$. When these scale factors are combined with the twiddle factors ω_N^k , we obtain terms of the form

$$t_{N,k} = \omega_N^k \frac{s_{N/4,k}}{s_{N,k}}, \quad (11)$$

which is always a complex number of the form $\pm 1 \pm i \tan \frac{2\pi k}{N}$ or $\pm \cot \frac{2\pi k}{N} \pm i$ and can therefore be multiplied with two fewer real multiplications than are required to multiply by ω_N^k . Because of the symmetry $s_{N,N/4-k} = s_{N,k}$, it is possible to write Algorithms 4-5 in a form similar to Algorithm 2, where the constant multiplicative factors are shared between k and $N/4 - k$.

The resulting flop count, for arbitrary complex data x_n , is then reduced from $4N \log_2 N - 6N + 8$ to

Algorithm 5 Rescaled FFT subroutines called recursively from Algorithm 4. 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 4.

function $X_{k=0..N-1} \leftarrow \mathbf{newfftS2}_N(x_n)$:
 {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**
 $X_k \leftarrow U_k + (t_{N,k} Z_k + t_{N,k}^* Z'_k) \cdot (s_{N,k}/s_{2N,k})$
 $X_{k+N/2} \leftarrow U_k - (t_{N,k} Z_k + t_{N,k}^* Z'_k) \cdot (s_{N,k}/s_{2N,k})$
 $X_{k+N/4} \leftarrow U_{k+N/4}$
 $\quad - i (t_{N,k} Z_k - t_{N,k}^* Z'_k) \cdot (s_{N,k}/s_{2N,k+N/4})$
 $X_{k+3N/4} \leftarrow U_{k+N/4}$
 $\quad + i (t_{N,k} Z_k - t_{N,k}^* Z'_k) \cdot (s_{N,k}/s_{2N,k+N/4})$
end for

function $X_{k=0..N-1} \leftarrow \mathbf{newfftS4}_N(x_n)$:
 {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**
 $X_k \leftarrow [U_k + (t_{N,k} Z_k + t_{N,k}^* Z'_k)] \cdot (s_{N,k}/s_{4N,k})$
 $X_{k+N/2} \leftarrow [U_k - (t_{N,k} Z_k + t_{N,k}^* Z'_k)]$
 $\quad \cdot (s_{N,k}/s_{4N,k+N/2})$
 $X_{k+N/4} \leftarrow [U_{k+N/4} - i (t_{N,k} Z_k - t_{N,k}^* Z'_k)]$
 $\quad \cdot (s_{N,k}/s_{4N,k+N/4})$
 $X_{k+3N/4} \leftarrow [U_{k+N/4} + i (t_{N,k} Z_k - t_{N,k}^* Z'_k)]$
 $\quad \cdot (s_{N,k}/s_{4N,k+3N/4})$
end for

$$\frac{34}{9} N \log_2 N - \frac{124}{27} N - 2 \log_2 N - \frac{2}{9} (-1)^{\log_2 N} \log_2 N + \frac{16}{27} (-1)^{\log_2 N} + 8. \quad (12)$$

In particular, assuming that complex multiplications are implemented as four real multiplications and two real additions, then the savings are purely in the number of real multiplications. The number $M(N)$ of real multiplications saved is given by:

$$M(N) = \frac{2}{9} N \log_2 N - \frac{38}{27} N + 2 \log_2 N + \frac{2}{9} (-1)^{\log_2 N} \log_2 N - \frac{16}{27} (-1)^{\log_2 N}. \quad (13)$$

If the data are purely real, it was shown that that $M(N)/2$ multiplications are saved over the corresponding real-input split-radix algorithm. These flop counts are to compute the original, unscaled definition of the DFT. If one is allowed to scale the outputs by any factor desired, then scaling by $1/s_{N,k}$ (corresponding to $\mathbf{newfftS}_N(x_n)$ in Algorithm 4),

Algorithm 6 New DCT-II algorithm of size $N = 2^m$, based on discarding redundant operations from the new FFT algorithm of size $4N$, achieving new record flop count.

function $C_{k=0..N-1} \leftarrow \mathbf{newdctII}_N(x_n)$:

for $n = 0$ to $N/2 - 1$ **do**

$\tilde{y}_n \leftarrow x_{2n}$

$\tilde{y}_{N-1-n} \leftarrow x_{2n+1}$

end for

$Z_{k=0..N-1} \leftarrow \mathbf{newfftS}_N(\tilde{y}_n)$

$C_0 \leftarrow 2Z_0$

for $k = 1$ to $N/2 - 1$ **do**

$C_k \leftarrow 2\Re(\omega_{4N}^k s_{N,k} Z_k)$

$C_{N-k} \leftarrow -2\Im(\omega_{4N}^k s_{N,k} Z_k)$

end for

$C_{N/2} \leftarrow \sqrt{2}Z_{N/2}$

saves an additional $M_S(N) - M(N) \geq 0$ multiplications for complex inputs, where:

$$M_S(N) = \frac{2}{9}N \log_2 N - \frac{20}{27}N + \frac{2}{9}(-1)^{\log_2 N} \log_2 N - \frac{7}{27}(-1)^{\log_2 N} + 1. \quad (14)$$

For real inputs, one similarly saves $M_S(N)/2$ flops for $\mathbf{newfftS}_N(x_n)$ operating on real x_n .

Although the division by a cosine or sine in the scale factor may, at first glance, seem as if it may exacerbate floating-point errors, this is not the case. The new FFT algorithm has the same $O(\sqrt{\log N})$ mean error growth, and $O(\log N)$ error bounds, as for other Cooley–Tukey-based FFT algorithms in finite-precision floating-point arithmetic [1]. The reason for this is straightforward: it never adds or subtracts scaled and unscaled values. Instead, wherever the original FFT would compute $a + b$, the new FFT computes $s \cdot (a + b)$ for some scale factor s .

5. Fast DCT-II from new FFT

To derive the new DCT-II algorithm based on the new FFT of the previous section, we follow exactly the same process as in Sec. 3.1. We express the DCT-II of length N in terms of a DFT of length $4N$, apply the FFT algorithm, and discard the redundant operations. As before, the U_k subtransform is identically zero, Z_k is the transform of the even elements of x_n followed by the odd elements in reverse order, and $Z'_k = Z_{N-k}^*$ is redundant.

The resulting algorithm is shown in Algorithm 6, and differs from the original fast DCT-II of Algorithm 3 in only two ways. First, instead of calling the ordinary split-radix (or conjugate-pair) FFT for the subtransform, it calls $\mathbf{newfftS}_N(x_n)$. Second, because this subtransform is scaled by $1/s_{N,k}$, the twiddle factor ω_{4N}^k in Algorithm 6 is multiplied by $s_{N,k}$.

To derive the flop count for Algorithm 6, we merely need to add the flop count for the subtransform (which saves $M_S(N)/2$ flops compared to ordinary real-input split radix)

with the number of flops in the loop, where the latter is exactly the same as for Algorithm 3 because the products $\omega_{4N}^k s_{N,k}$ can be precomputed. Therefore, we save a total of $M_S(N)/2$ flops compared to the previous best flop count of $2N \log_2 N - N + 2$, resulting in the flop count of eq. (1). This formula matches the flop count that was achieved by an automatic code generator in Ref. [1].

Because the new DCT algorithm is mathematically merely a special set of inputs for the underlying FFT, it will have the same favorable logarithmic error bounds as discussed in the previous section.

6. Normalizations

In the above definition of DCT-II, we use a scale factor “2” in front of the \sum summation in order to make the DCT-II directly equivalent to the corresponding DFT. But in some circumstances, it is useful to multiply by other factors, and different normalizations lead to slightly different flop counts. For example, one could use the unitary normalization from eq. (2), which replaces 2 by $\sqrt{2/N}$ or $\sqrt{1/N}$ (for $k = 0$) and requires the same number of flops. If one uses the unitary normalization multiplied by \sqrt{N} , then one saves two multiplications in the $k = 0$ and $k = N/2$ terms (in this normalization, $C_0 = Z_0$ and $C_{N/2} = Z_{N/2}$) for both Algorithm 3.1 and Algorithm 5. (This leads to a commonly quoted $2N \log_2 N - N$ formula for the previous flop count.)

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 [40], and in this case the scaling can save 8 multiplications [39] over the 42 flops required for an unitary 8-point DCT-II. Since our $\mathbf{newfftS}_N(x_n)$ attempts to be the optimal scaled FFT, we should be able to derive this scaled DCT-II by using $\mathbf{newfftS}_N(x_n)/2$ for our length- $4N$ DFT instead of $\mathbf{newfft}_N(x)$, and again pruning the redundant operations. Doing so, we obtain an algorithm identical to Algorithm 6 except that $2\omega_{4N}^k s_{N,k}$ is replaced by $t_{4N,k}$, which requires fewer multiplications, and also we now obtain $C_0 = Z_0$ and $C_{N/2} = Z_{N/2}$. This algorithm computes $C_k/2s_{4N,k}$ instead of C_k . In particular, we save exactly N multiplications over Algorithm 6, which matches the result by Ref. [39] for $N = 8$ but generalizes it to all N . This savings of N multiplications for a scaled DCT-II also matches the flop count that was achieved by an automatic code generator in Ref. [1].

7. Fast DCT-III, DST-II, and DST-III

Given any algorithm for the DCT-II, one can immediately obtain algorithms for the DCT-III, DST-II, and DST-III with exactly the same number of arithmetic operations. In this way, any improvement in the DCT-II, such as the one described in this paper, immediately leads to improved algorithms for those transforms and vice versa. In this section, we review the equivalence between those transforms.

7.1. DCT-III

To obtain a DCT-III algorithm from a DCT-II algorithm, one can exploit two facts. First, the DCT-III, viewed as a matrix, is simply the transpose of the DCT-II matrix. Second, any linear algorithm can be viewed as a *linear network*, and the transpose operation is computed by the *network transposition* of this algorithm [43]. To review, a linear network represents the algorithm by a directed acyclic graph (dag), where the edges represent multiplications by constants and the vertices represent additions of the incoming edges. Network transposition simply consists of reversing the direction of every edge. We prove below that the transposed network requires the same number of additions and multiplications for networks with the same number of inputs and outputs, and therefore the DCT-III can be computed in the same number of flops as the DCT-II by the transposed algorithm. The DCT-III corresponding to the transpose of eq. (4) is

$$C_k^T = 2 \sum_{n=0}^{N-1} x_n \cos \left[\frac{\pi}{N} n \left(k + \frac{1}{2} \right) \right]. \quad (15)$$

A proof that the transposed network requires the same number of flops is as follows. Clearly, the number of multiplications, the number of edges with weight $\neq \pm 1$, is unchanged by transposition. The number of additions is given by the sum of indegree -1 for all the vertices, except for the N input vertices which have indegree zero. That is, for a set V of vertices and a set E of edges:

$$\begin{aligned} \# \text{ adds} &= N + \sum_{v \in V} [\text{indegree}(v) - 1] \\ &= N + |E| - |V|, \end{aligned} \quad (16)$$

using the fact that the sum of the indegree over all vertices is simply the number $|E|$ of edges. Because the above expression is obviously invariant under network transposition as long as N is unchanged (i.e. same number of inputs and outputs), the number of additions is invariant.

More explicitly, a fast DCT-III algorithm derived from the transpose of our new Algorithm 6 is shown in Algorithm 7. Whereas for the DCT-II we computed the real-input DFT (with conjugate-symmetric output) of the rearranged inputs \tilde{y}_n and then post-processed the complex outputs $Z_k = Z_{N-k}^*$, now we do the reverse. We first preprocess the inputs to form a complex array $Z_k = Z_{N-k}^*$, then perform a real-output, scaled-input DFT to obtain \tilde{y}_k , and finally assign the even and odd elements of the result C_k^T from the first and second halves of \tilde{y}_k . The real-output (scaled, conjugate-symmetric input) version of $\text{newfftS}_N(x_n)$ can be derived by network transposition of the real-input case. Equivalently, whereas the $\text{newfftS}_N(x_n)$ was a scaled-output decimation-in-time algorithm, $\text{newfftS}_N^T(x_n)$ is a scaled-input decimation-in-frequency algorithm, in which the real-output case can be derived by discarding the redundant operations.

Algorithm 7 New DCT-III algorithm of size $N = 2^m$, based on network transposition of Algorithm 6, with the same flop count.

function $D_{k=0..N-1} \leftarrow \text{newdctIII}_N(x_n)$:

```

 $Z_0 \leftarrow 2x_0$ 
for  $n = 1$  to  $N/2 - 1$  do
   $Z_n \leftarrow 2\omega_{4N}^{-n} s_{N,n}(x_n - ix_{N-n})$ 
   $Z_{N-n} \leftarrow Z_n^*$ 
end for
 $Z_{N/2} \leftarrow \sqrt{2}x_{N/2}$ 
 $\tilde{y}_k \leftarrow \text{newfftS}_N^T(Z_n)$ 
for  $k = 0$  to  $N/2 - 1$  do
   $C_{2k}^T \leftarrow \tilde{y}_k$ 
   $C_{2k+1}^T \leftarrow \tilde{y}_{N-1-k}$ 
end for

```

7.2. DST-II and DST-III

The DST-II is defined, using a unitary normalization, as:

$$S_k^{\text{II}} = \sqrt{\frac{2 - \delta_{k,N}}{N}} \sum_{n=0}^{N-1} x_n \sin \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right] \quad (17)$$

for $k = 1, \dots, N$ (not $0, \dots, N-1$). As in section 2, it is more convenient to develop algorithms for an unnormalized variation:

$$S_k = 2 \sum_{n=0}^{N-1} x_n \sin \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right] \quad (18)$$

similar to our normalization of C_k and C_k^T above. Although we could derive fast algorithms for S_k directly by treating it as a DFT of length $4N$ with odd symmetry, interleaved with zeros, and discarding redundant operations similar to above, it turns out there is a simpler technique. The DST-II is *exactly* equivalent to a DCT-II in which the outputs are reversed and every other input is multiplied by -1 [11–13]:

$$S_{N-k} = 2 \sum_{n=0}^{N-1} (-1)^n x_n \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right] \quad (19)$$

for $k = 0, \dots, N-1$. It therefore follows that a DST-II can be computed with the same number of flops as a DCT-II of the same size, assuming that multiplications by -1 are free—the reason for this is that sign flips can be absorbed at no cost by converting additions into subtractions or vice versa in the subsequent algorithmic steps. Therefore, our new flop count (1) immediately applies to the DST-II.

Similarly, a DST-III is given by the transpose of the DST-II (which is the inverse, for the unitary normalization). In unnormalized form, this is

$$S_k^T = 2 \sum_{n=1}^N x_n \sin \left[\frac{\pi}{N} n \left(k + \frac{1}{2} \right) \right] \quad (20)$$

for $k = 0, \dots, N-1$. This must have the same number of flops as the DST-II by the network transposition argument above. More explicitly, it can be obtained from the DCT-III

by reversing the inputs and multiplying every other output by -1 [12]:

$$S_k^T = (-1)^k \sum_{n=0}^{N-1} x_{N-n} \cos \left[\frac{\pi}{N} n \left(k + \frac{1}{2} \right) \right], \quad (21)$$

which can be obtained from C_k^T with the same number of flops.

8. Conclusion

We have shown that a improved count of real additions and multiplications (flops), eq. (1), can be obtained for the DCT/DST types II/III by pruning redundant operations from a recent improved FFT algorithm. We have also shown how to save N multiplications by rescaling the outputs of a DCT-II (or the inputs of a DCT-III), generalizing a well-known technique for $N = 8$ [39]. However, we do not claim that our new count is optimal—it may be that further gains can be obtained by, for example, extending the rescaling technique of [1] to greater generality. In particular, as has been pointed out by other authors [8], any improvement in the arithmetic complexity or flop counts for the DFT immediately leads to corresponding improvements in DCTs/DSTs, and vice versa. Our most important result, we believe, is the fact that there suddenly appears to be new room for improvement in problems that had seen no reductions in flop counts for many years.

The question of the minimal operation counts for basic linear transformations such as DCTs and DSTs is of long-standing theoretical interest. The practical impact of a few percent improvement in flop counts, admittedly, is less clear because computation time on modern hardware is often not dominated by arithmetic [10]. Nevertheless, minimal-arithmetic hard-coded DCTs of small sizes are often used in audio and image compression [40, 41], and the availability of any new algorithm with regular structure amenable to implementation leads to rich new opportunities for performance optimization.

Similar arithmetic improvements also occur for other types of DCT and DST [1], as well as for the modified DCT (MDCT, closely related to a DCT-IV), and the explicit description of those algorithms is the subject of future manuscripts currently in preparation.

Acknowledgements

This work was supported in part by a grant from the MIT Undergraduate Research Opportunities Program. The authors are also grateful to M. Frigo, co-author of FFTW with SGJ [10], for many helpful discussions.

References

[1] S. G. Johnson, M. Frigo, A modified split-radix FFT with fewer arithmetic operations, *IEEE Trans. Signal Processing* 55 (1) (2007) 111–119.

[2] I. Kamar, Y. Elcherif, Conjugate pair fast Fourier transform, *Electron. Lett.* 25 (5) (1989) 324–325.

[3] R. A. Gopinath, Comment: Conjugate pair fast Fourier transform, *Electron. Lett.* 25 (16) (1989) 1084.

[4] H.-S. Qian, Z.-J. Zhao, Comment: Conjugate pair fast Fourier transform, *Electron. Lett.* 26 (8) (1990) 541–542.

[5] A. M. Krot, H. B. Minervina, Comment: Conjugate pair fast Fourier transform, *Electron. Lett.* 28 (12) (1992) 1143–1144.

[6] M. Vetterli, H. J. Nussbaumer, Simple FFT and DCT algorithms with reduced number of operations, *Signal Processing* 6 (4) (1984) 267–278.

[7] P. Duhamel, Implementation of “split-radix” FFT algorithms for complex, real, and real-symmetric data, *IEEE Trans. Acoust., Speech, Signal Processing* 34 (2) (1986) 285–295.

[8] P. Duhamel, M. Vetterli, Fast Fourier transforms: a tutorial review and a state of the art, *Signal Processing* 19 (1990) 259–299.

[9] R. Vuduc, J. Demmel, Code generators for automatic tuning of numerical kernels: experiences with FFTW, in: *Proc. Semantics, Application, and Implementation of Code Generators Workshop*, Montreal, 2000.

[10] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, *Proc. IEEE* 93 (2) (2005) 216–231.

[11] Z. Wang, A fast algorithm for the discrete sine transform implemented by the fast cosine transform, *IEEE Trans. Acoust., Speech, Signal Processing* 30 (5) (1982) 814–815.

[12] S. C. Chan, K. L. Ho, Direct methods for computing discrete sinusoidal transforms, *IEE Proceedings* 137 (6) (1990) 433–442.

[13] P. Lee, F.-Y. Huang, Restructured recursive DCT and DST algorithms, *IEEE Trans. Signal Processing* 42 (7) (1994) 1600–1609.

[14] R. Yavne, An economical method for calculating the discrete Fourier transform, in: *Proc. AFIPS Fall Joint Computer Conf.*, Vol. 33, 1968, pp. 115–125.

[15] P. Duhamel, H. Hollmann, Split-radix FFT algorithm, *Electron. Lett.* 20 (1) (1984) 14–16.

[16] J. B. Martens, Recursive cyclotomic factorization—a new algorithm for calculating the discrete Fourier transform, *IEEE Trans. Acoust., Speech, Signal Processing* 32 (4) (1984) 750–761.

[17] B. G. Lee, A new algorithm to compute the discrete cosine transform, *IEEE Trans. Acoust., Speech, Signal Processing* 32 (6) (1984) 1243–1245.

[18] Z. Wang, On computing the discrete Fourier and cosine transforms, *IEEE Trans. Acoust., Speech, Signal Processing* 33 (4) (1985) 1341–1344.

[19] N. Suehiro, M. Hatori, Fast algorithms for the DFT and other sinusoidal transforms, *IEEE Trans. Acoust., Speech, Signal Processing* 34 (3) (1986) 642–644.

[20] H. S. Hou, A fast algorithm for computing the discrete cosine transform, *IEEE Trans. Acoust., Speech, Signal Processing* 35 (10) (1987) 1455–1461.

[21] F. Arguello, E. L. Zapata, Fast cosine transform based on the successive doubling method, *Electronic letters* 26 (19) (1990) 1616–1618.

[22] C. W. Kok, Fast algorithm for computing discrete cosine transform, *IEEE Trans. Signal Processing* 45 (3) (1997) 757–760.

[23] J. Takala, D. Akopian, J. Astola, J. Saarinen, Constant geometry algorithm for discrete cosine transform, *IEEE Trans. Signal Processing* 48 (6) (2000) 1840–1843.

[24] M. Püschel, J. M. F. Moura, The algebraic approach to the discrete cosine and sine transforms and their fast algorithms, *SIAM J. Computing* 32 (5) (2003) 1280–1316.

[25] M. Püschel, Cooley–Tukey FFT like algorithms for the DCT, in: *Proc. IEEE Int’l Conf. Acoustics, Speech, and Signal Processing*, Vol. 2, Hong Kong, 2003, pp. 501–504.

[26] R. M. Haralick, A storage efficient way to implement the discrete cosine transform, *IEEE Trans. Comput. (Corresp.)* 25 (1976) 764–765.

- [27] W.-H. Chen, C. H. Smith, S. C. Fralick, A fast computational algorithm for the discrete cosine transform, *IEEE Trans. Commun.* 25 (9) (1977) 1004–1009.
- [28] M. J. Narasimha, A. M. Peterson, On the computation of the discrete cosine transform, *IEEE Trans. Commun.* 26 (6) (1978) 934–936.
- [29] B. D. Tseng, W. C. Miller, On computing the discrete cosine transform, *IEEE Trans. Comput. (Corresp.)* 27 (10) (1978) 966–968.
- [30] J. Makhoul, A fast cosine transform in one and two dimensions, *IEEE Trans. Acoust., Speech, Signal Processing* 28 (1) (1980) 27–34.
- [31] H. S. Malvar, Fast computation of the discrete cosine transform and the discrete Hartley transform, *IEEE Trans. Acoust., Speech, Signal Processing* 35 (10) (1987) 1484–1485.
- [32] W. Li, A new algorithm to compute the DCT and its inverse, *IEEE Trans. Signal Processing* 39 (6) (1991) 1305–1313.
- [33] G. Steidl, M. Tasche, A polynomial approach to fast algorithms for discrete Fourier-cosine and Fourier-sine transforms, *Math. Comp.* 56 (193) (1991) 281–296.
- [34] E. Feig, S. Winograd, On the multiplicative complexity of discrete cosine transforms, *IEEE Trans. Info. Theory* 38 (4) (1992) 1387–1391.
- [35] J. Astola, D. Akopian, Architecture-oriented regular algorithms for discrete sine and cosine transforms, *IEEE Trans. Signal Processing* 47 (4) (1999) 1109–1124.
- [36] Z. Guo, B. Shi, N. Wang, Two new algorithms based on product system for discrete cosine transform, *Signal Processing* 81 (2001) 1899–1908.
- [37] G. Plonka, M. Tasche, Fast and numerically stable algorithms for discrete cosine transforms, *Lin. Algebra and its Appl.* 394 (2005) 309–345.
- [38] M. Frigo, A fast Fourier transform compiler, in: *Proc. ACM SIGPLAN’99 Conference on Programming Language Design and Implementation (PLDI)*, Vol. 34, ACM, Atlanta, Georgia, 1999, pp. 169–180.
- [39] Y. Arai, T. Agui, M. Nakaajima, A fast DCT-SQ scheme for images, *Trans. IEICE* 71 (11) (1988) 1095–1097.
- [40] W. B. Pennebaker, J. L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993.
- [41] K. R. Rao, P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications*, Academic Press, Boston, MA, 1990.
- [42] H. V. Sorensen, D. L. Jones, M. T. Heideman, C. S. Burrus, Real-valued fast Fourier transform algorithms, *IEEE Trans. Acoust., Speech, Signal Processing* 35 (6) (1987) 849–863.
- [43] R. E. Crochiere, A. V. Oppenheim, Analysis of linear digital networks, *Proc. IEEE* 63 (4) (1975) 581–595.