Automatic Generation of Prime Length FFT Programs

Ivan W. Selesnick* and C. Sidney Burrus

Department of Electrical and Computer Engineering - MS 366
Rice University
Houston, TX 77251-1892

September 20, 1995

Abstract

We describe a set of programs for circular convolution and prime length FFTs that are relatively short, possess great structure, share many computational procedures, and cover a large variety of lengths. The programs make clear the structure of the algorithms and clearly enumerate independent computational branches that can be performed in parallel. Moreover, each of these independent operations is made up of a sequence of sub-operations which can be implemented as vector/parallel operations. This is in contrast with previously existing programs for prime length FFTs: they consist of straight line code, no code is shared between them, and they can not be easily adapted for vector/parallel implementations.

We have also developed a program that automatically generates these programs for prime length FFTs. This code generating program requires information only about a set of modules for computing cyclotomic convolutions.

Contact Address:
Ivan W. Selesnick
Electrical and Computer Engineering - MS 366
Rice University,
Houston, TX 77251-1892
Phone: (713) 527-8101 x3508
FAX: (713) 524-5237
email: selesi@ece.rice.edu

To appear: IEEE Transactions on Signal Processing

*This work has been supported by DARPA under an NDSEG fellowship.

List of Figures

List of Tables

1 Introduction

The development of algorithms for the fast computation of the Discrete Fourier Transform in the last 30 years originated with the radix 2 Cooley-Tukey FFT and the theory and variety of FFTs has grown significantly since then. Most of the work has focused on FFTs whose sizes are composite, for the algorithms depend on the ability to factor the length of the data sequence so that the transform can be found by taking the transform of smaller lengths. For this reason, algorithms for prime length transforms are building blocks for many composite length FFTs - the maximum length and the variety of lengths of a PFA or WFTA algorithm depend upon the availability of prime length FFT modules. As such, prime length Fast Fourier Transforms are a special, important and difficult case.

Fast algorithms designed for specific short prime lengths have been developed and have been written as straight line code [7, 9]. These dedicated programs rely upon an observation made in Rader's paper [15] in which he shows that a prime p length DFT can be found by performing a $p-1$ length circular convolution. Since the publication of that paper, Winograd had developed a theory of multiplicative complexity for transforms and designed algorithms for convolution that attain the minimum number of multiplications [20]. Although Winograd's algorithms are very efficient for small prime lengths, for longer lengths they require a large number of additions and the algorithms become very cumbersome to design. This has prevented the design of useful prime length FFT programs for lengths greater than 31. Although we have previously reported the design of programs for prime lengths greater than 31 [16] those programs required more additions than necessary and were long. Like the previously existing ones, they simply consisted of a long list of instructions and did not take advantage of the attainable common structures.

In this paper we describe a set of programs for circular convolution and prime length FFTs that are are short, possess great structure, share many computational procedures, and cover a large variety of lengths. Because the underlying convolution is decomposed into a set of disjoint operations they can be performed in parallel and this parallelism is made clear in the programs. Moreover, each of these independent operations is made up of a sequence of sub-operations of the form $I \otimes A \otimes I$ where $\otimes$ denotes the Kronecker product. These operations can be implemented as vector/parallel operations [5, 19]. Previous programs for prime length FFTs do not have these features: they consist of straight line code and are not amenable to vector/parallel implementations.

We have also developed a program that automatically generates these programs for circular convolution and prime length DFTs. This code generating program requires information only about a set of modules for computing cyclotomic convolutions. We compute these non-circular convolutions by computing a linear convolution and reducing the result. Furthermore, because these linear convolution algorithms can be built from smaller ones, the only modules needed are ones for the linear convolution of prime length sequences. It turns out that with linear convolution algorithms for only the lengths 2 and 3, we can generate a wide variety of prime length FFT algorithms. In addition, the code we generate is made up of calls to a relatively small set of functions. Accordingly, the subroutines can be designed and optimized to specifically suit a given architecture.

The programs we describe use Rader's conversion of a prime point DFT into a circular convolution, but this convolution we compute using the split nesting algorithm [14]. As Stasinski notes [17], this yields algorithms possessing greater structure, simpler programs and doesn't generally require more computation. We wish to note also, that Jones [10] has advocated the use of the Agarwal-Cooley algorithm for prime length FFTs.

1.1 Preliminaries

Because we compute prime length DFTs by converting them in to circular convolutions, most of this and the next section is devoted to an explanation of the split nesting convolution algorithm. In this section we introduce the various operations needed to carry out the split nesting algorithm. In particular, we describe the prime factor permutation that is used to convert a one-dimensional circular convolution into a multi-dimensional one. We also discuss the reduction operations needed when the Chinese Remainder Theorem for polynomials is used in the computation of convolution. The reduction operations needed for the split nesting algorithm are particularly well organized. We give an explicit matrix description of the reduction operations and give a program that implements the action of these reduction operations.

The presentation relies upon the notions of similarity transformations, companion matrices and Kronecker products. With them, we describe the split nesting algorithm in a manner that brings out its structure. We find that when companion matrices are used to describe convolution, the reduction operations block diagonalize the circular shift matrix.

The companion matrix of a monic polynomial, $M(s) = m_0 + m_1 s + \cdots + m_{n-1}s^{n-1} + s^n$ is given by

$$C_M = \begin{bmatrix} & & & -m_0 \\ 1 & & & -m_1 \\ & \ddots & & \vdots \\ & & 1 & -m_{n-1} \end{bmatrix}. \tag{1}$$

Its usefulness in the following discussion comes from the following relation which permits a matrix formulation of convolution:

$$Y(s) = \langle H(s)X(s)\rangle_{M(s)} \quad \Longleftrightarrow \quad y = \left(\sum_{k=0}^{n-1} h_k C_M^k\right) x \tag{2}$$

where x, h and y are the coefficients and C_M is the companion matrix of $M(s)$. In (2), we say y is the convolution of x and h with respect to $M(s)$. In the case of circular convolution, $M(s) = s^n - 1$ and C_{s^n-1} is the circular shift matrix denoted by S_n,

$$S_n = \begin{bmatrix} & & & 1 \\ 1 & & & \\ & \ddots & & \\ & & 1 & \end{bmatrix}.$$

Notice that any circulant matrix can be written as $\sum_k h_k S_n^k$ [19, 21].

Similarity transformations can be used to interpret the action of some convolution algorithms. If $C_M = T^{-1}AT$ for some matrix T (C_M and A are similar, denoted $C_M \sim A$), then (2) becomes

$$y = T^{-1}\left(\sum_{k=0}^{n-1} h_k A^k\right) Tx. \tag{3}$$

That is, by employing the similarity transformation given by T in this way, the action of S_n^k is replaced by that of A^k. Many circular convolution algorithms can be understood, in part, by understanding the manipulations made to S_n and the resulting new matrix A. If the transformation T is to be useful, it must satisfy two requirements: (1) Tx must be simple to compute, and (2) A must have some advantageous structure. For example, by the convolution property of the DFT, the DFT matrix F diagonalizes S_n and therefore it diagonalizes every circulant matrix. In this case, Tx can be computed by an FFT and the structure of A is the simplest possible: a diagonal.

The Winograd Structure can be described in this manner also. Suppose $M(s)$ can be factored as $M(s) = M_1(s)M_2(s)$ where M_1 and M_2 have no common roots, then $C_M \sim (C_{M_1} \oplus C_{M_2})$ where $\oplus$ denotes the matrix direct sum. Using this similarity and recalling (2), the original convolution can be decomposed into disjoint convolutions. This is, in fact, a statement of the Chinese Remainder Theorem for polynomials expressed in matrix notation. In the case of circular convolution, $s^n - 1 = \prod_{d|n} \Phi_d(s)$, so that S_n can be transformed to a block diagonal matrix,

$$S_n \sim \begin{bmatrix} C_{\Phi_1} & & & \\ & C_{\Phi_d} & & \\ & & \ddots & \\ & & & C_{\Phi_n} \end{bmatrix} = \left(\bigoplus_{d|n} C_{\Phi_d}\right) \tag{4}$$

where $\Phi_d(s)$ is the d^{th} cyclotomic polynomial. Here C_{Φ_d} is the companion matrix of the d^{th} cyclotomic polynomial. In this case, each block represents a convolution with respect to a cyclotomic polynomial, or a 'cyclotomic convolution'.

The Agarwal-Cooley Algorithm utilizes the fact that

$$S_n \sim (S_{n_1} \otimes S_{n_2}) \tag{5}$$

where $n = n_1 n_2$, $(n_1, n_2) = 1$ [1]. This converts the one dimensional circular convolution of length n to a two dimensional one of length n_1 along one dimension and length n_2 along the second. Then an n_1 point and an n_2 point circular convolution algorithm can be combined to obtain an n point algorithm. The Agarwal-Cooley convolution algorithm is described using tensor product formalism in [19, chap. 7].

The Split-Nesting algorithm [14] combines the structures of the Winograd and Agarwal-Cooley methods, so that S_n is transformed to a block diagonal matrix as in (4),

$$S_n \sim \bigoplus_{d|n} \Psi(d). \tag{6}$$

Here $\Psi(d) = \bigotimes_{p|d, p\in\mathcal{P}} C_{\Phi_{H_d(p)}}$ where $H_d(p)$ is the highest power of p dividing d, and $\mathcal{P}$ is the set of primes.

Example 1:

$$S_{45} \sim \begin{bmatrix} 1 & & & & & \\ & C_{\Phi_3} & & & & \\ & & C_{\Phi_9} & & & \\ & & & C_{\Phi_5} & & \\ & & & & C_{\Phi_3} \otimes C_{\Phi_5} & \\ & & & & & C_{\Phi_9} \otimes C_{\Phi_5} \end{bmatrix} \tag{7}$$

In this structure a multi-dimensional cyclotomic convolution, represented by $\Psi(d)$, replaces each cyclotomic convolution in Winograd's algorithm (represented by C_{Φ_d} in (4)). Indeed, if the product of $b_1, \ldots, b_k$ is d and they are pairwise relatively prime, then $C_{\Phi_d} \sim C_{\Phi_{b_1}} \otimes \cdots \otimes C_{\Phi_{b_k}}$.

The split nesting algorithm therefore combines cyclotomic convolutions to compute a longer circular convolution. It is like the Agarwal-Cooley method but requires fewer additions [14].

1.2 Prime Factor Permutations

The permutation of the prime factor FFT [1, 13] can be used to obtain $S_{n_1} \otimes S_{n_2}$ from $S_{n_1 n_2}$ when $(n_1, n_2) = 1$. The permutation is described by Zalcstein [21], among others. Let e_k denote the k^{th} standard basis vector.

Lemma 1 *If $n = n_1 \cdots n_k$ and $n_1, \ldots, n_k$ are pairwise relatively prime, then $S_n = P^t(S_{n_k} \otimes \cdots \otimes S_{n_1})P$ where P is the permutation matrix given by $Pe_k = e_{\langle k \rangle_{n_1} + n_1 \langle k \rangle_{n_2} + \cdots + n_1 \cdots n_{k-1} \langle k \rangle_{n_k}}$.*

This useful permutation will be denoted here as $P_{n_k,\ldots,n_1}$. If $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ then this permutation yields the matrix, $S_{p_1^{e_1}} \otimes \cdots \otimes S_{p_k^{e_k}}$. This product can be written simply as $\otimes_{i=1}^k S_{p_i^{e_i}}$, so that one has $S_n = P_{n_1,\ldots,n_k}^t \left(\otimes_{i=1}^k S_{p_i^{e_i}} \right) P_{n_1,\ldots,n_k}$.

1.3 Reduction Operations

The Chinese Remainder Theorem for polynomials can be used to decompose a convolution of two sequences (the polynomial product of two polynomials evaluated modulo a third polynomial) into smaller convolutions (smaller polynomial products) [20]. The Winograd n point circular convolution algorithm requires that polynomials are reduced modulo the cyclotomic polynomial factors of $s^n - 1$, $\Phi_d(s)$ for each d dividing n.

When n has several prime divisors the reduction operations become quite complicated and writing a single program to implement them is difficult. However, when n is a prime power, the reduction operations are very structured and can be done in a straightforward manner. Therefore, by converting a one-dimensional convolution to a multi-dimensional one, in which the length along each dimension is a prime power, the split nesting algorithm avoids the need for complicated reductions operations. This is one advantage the split nesting algorithm has over the Winograd algorithm.

By applying the reduction operations appropriately to the circular shift matrix, we are able to obtain a block diagonal form, just as in the Winograd convolution algorithm. However, in the split nesting algorithm, each diagonal block represents multi-dimensional cyclotomic convolution rather than a one-dimensional one. By forming multi-dimensional convolutions out of one-dimensional ones, it is possible to combine algorithms for smaller convolutions (see the next section). This is a second advantage split nesting algorithm has over the Winograd algorithm. The split nesting algorithm, however, generally uses more than the minimum number of multiplications.

Below we give an explicit matrix description of the required reduction operations, give a program that implements them, and give a formula for the number of additions required. (No multiplications are needed.)

To obtain the block diagonal form of (6) and (7) let $\underline{1}_p$ be a column vector of p 1's and let

G_p be the $(p-1) \times p$ matrix:

$$G_p = \begin{bmatrix} 1 & & & & -1 \\ & 1 & & & -1 \\ & & \ddots & & \vdots \\ & & & 1 & -1 \end{bmatrix} \tag{8}$$

then

$$R \left(S_{p_1^{e_1}} \otimes \cdots \otimes S_{p_k^{e_k}} \right) R^{-1} = \bigoplus_{d|n} \Psi(d) \tag{9}$$

where $R = R_{p_1^{e_1},\ldots,p_k^{e_k}}$ is given by

$$R_{p_1^{e_1},\ldots,p_k^{e_k}} = \prod_{i=k}^{1} Q(m_i, p_i^{e_i}, n_i) \tag{10}$$

with $m_i = \prod_{j=1}^{i-1} p_j^{e_j}$, $n_i = \prod_{j=i+1}^{k} p_j^{e_j}$ and

$$Q(a, p^e, c) = \prod_{j=0}^{e-1} \begin{bmatrix} I_a \otimes \underline{1}_p^t \otimes I_{cp^j} & \\ I_a \otimes G_p \otimes I_{cp^j} & \\ & I_{ac(p^e - p^{j+1})} \end{bmatrix}. \tag{11}$$

The number of additions incurred by R is given by $2n \left(k - \sum_{i=1}^{k} \frac{1}{p_i^{e_i}} \right)$ where $n = p_1^{e_1} \ldots p_k^{e_k}$.

Example 2:

$$R\,(S_9 \otimes S_5)\,R^{-1} = \begin{bmatrix} 1 & & & & & \\ & C_{\Phi_3} & & & & \\ & & C_{\Phi_9} & & & \\ & & & C_{\Phi_5} & & \\ & & & & C_{\Phi_3} \otimes C_{\Phi_5} & \\ & & & & & C_{\Phi_9} \otimes C_{\Phi_5} \end{bmatrix} \tag{12}$$

where $R = R_{9,5}$ and can be implemented with 152 additions.

Each block in eq (9) and (12) represents a multi-dimensional cyclotomic convolution.

A Matlab program that carries out the operation $R_{p_1^{e_1},\ldots,p_k^{e_k}}$ in eq (9) is `KRED()`.

```
function x = KRED(P,E,K,x)
% P : P = [P(1),...,P(K)]
% E : E = [E(K),...,E(K)]
% x : length(x) == prod(P.^E)
for i = 1:K
   a = prod(P(1:i-1).^E(1:i-1));
   c = prod(P(i+1:K).^E(i+1:K));
   p = P(i);
```

```
    e = E(i);
    for j = e-1:-1:0
       x(1:a*c*(p^(j+1))) = RED(p,a,c*(p^j),x(1:a*c*(p^(j+1))));
    end
end
```

It calls the Matlab program `RED()`.

```
function y = RED(p,a,c,x)
% x : length(x) == a*c*p
y = zeros(a*c*p,1);
for i = 0:c:(a-1)*c
   for j = 0:c-1
      y(i+j+1) = x(i*p+j+1);
      for k = 0:c:c*(p-2)
         y(i+j+1) = y(i+j+1) + x(i*p+j+k+c+1);
         y(i*(p-1)+j+k+a*c+1) = x(i*p+j+k+1) - x(i*p+j+c*(p-1)+1);
      end
   end
end
```

These two Matlab programs are not written to execute as fast as they could be. They are a 'naive' coding of $R_{p_1^{e_1},\ldots,p_k^{e_k}}$ and are meant to serve as a basis for more efficient programs. In particular, the indexing and the loop counters can be modified to improve the efficiency. However, the modifications that minimize the overhead incurred by indexing operations depends on the programming language, the compiler and the computer used. These two programs are written with simple loop counters and complicated indexing operations so that appropriate modifications can be easily made.

It will also be important to have a program that carry out the transpose of these reduction operations. A Matlab program that carries out the operation $R^t_{p_1^{e_1},\ldots,p_k^{e_k}}$ is `tKRED()`.

```
function x = tKRED(P,E,K,x)
% x = tKRED(P,E,K,x);
% (transpose)
% P : P = [P(1),...,P(K)];
% E : E = [E(K),...,E(K)];
```

```
for i = K:-1:1
   a = prod(P(1:i-1).^E(1:i-1));
   c = prod(P(i+1:K).^E(i+1:K));
   p = P(i);
   e = E(i);
   for j = 0:e-1
      x(1:a*c*(p^(j+1))) = tRED(p,a,c*(p^j),x(1:a*c*(p^(j+1))));
   end
end
```

It calls the Matlab program `tRED()`.

```
% y = tRED(p,a,c,x);
% (transpose)
y = zeros(a*c*p,1);
for i = 0:c:(a-1)*c
   for j = 0:c-1
      y(i*p+j+c*(p-1)+1) = x(i+j+1);
      for k = 0:c:c*(p-2)
         y(i*p+j+k+1) = x(i+j+1) + x(i*(p-1)+j+k+a*c+1);
         y(i*p+j+c*(p-1)+1) = y(i*p+j+c*(p-1)+1) - x(i*(p-1)+j+k+a*c+1);
      end
   end
end
```

In using the similarity (6) and (7) it is necessary to implement R^{-1}. To this end we note that the inverse of R_p has the form

$$R_p^{-1} = \frac{1}{p}\begin{bmatrix} 1 & p-1 & -1 & -1 & -1 \\ 1 & -1 & p-1 & -1 & -1 \\ 1 & -1 & -1 & p-1 & -1 \\ 1 & -1 & -1 & -1 & p-1 \\ 1 & -1 & -1 & -1 & -1 \end{bmatrix}. \tag{13}$$

The inverse of the matrix R described by eqs (9), (10) and (11) is therefore given by

$$R^{-1} = \prod_{i=1}^{k} Q(m_i, p_i^{e_i}, n_i)^{-1} \tag{14}$$

with $m_i = \prod_{j=1}^{i-1} p_j^{e_j}$, $n_i = \prod_{j=i+1}^{k} p_j^{e_j}$ and

$$Q(a, p^e, c)^{-1} = \prod_{j=e-1}^{0} \begin{bmatrix} I_a \otimes \underline{1}_p^t \otimes I_{cp^j} & I_a \otimes V_p \otimes I_{cp^j} & \\ & & I_{ac(p^e - p^{j+1})} \end{bmatrix} \tag{15}$$

where V_p denotes the matrix in eq (13) without its first column.

We have written programs for R^{-1}, R^t and R^{-t}. They are similar to the programs for R above. It should be noted that by using the *matrix exchange* property below, the relevant operations that need to be implemented turn out to be R, R^t and R^{-t}.

2 Bilinear Forms for Circular Convolution

A basic technique in fast algorithms for convolution is interpolation: two polynomials are evaluated at some common points, these values are multiplied, and by computing the polynomial interpolating these products, the product of the two original polynomials is determined [2, 12, 14]. This interpolation method is often called the Toom-Cook method and it is given by two matrices that describe a bilinear form.

We use bilinear forms to give a matrix formulation of the split nesting algorithm. The split nesting algorithm combines smaller convolution algorithms to obtain algorithms for longer lengths. We use the Kronecker product to explicitly describe the way in which smaller convolution algorithms are appropriately combined.

2.1 The Toom-Cook Method

Recall that the linear convolution of h and x can be represented by a matrix vector product. When $n = 3$:

$$\begin{bmatrix} h_0 & & \\ h_1 & h_0 & \\ h_2 & h_1 & h_0 \\ & h_2 & h_1 \\ & & h_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \tag{16}$$

This linear convolution matrix can be written as $h_0 H_0 + h_1 H_1 + h_2 H_2$ where H_k are clear. This product $y = \sum_{k=0}^{n-1} h_k H_k x$ can be found using the Toom-Cook algorithm. One writes

$$\sum_{k=0}^{n-1} h_k H_k x = C\left\{Ah * Ax\right\} \tag{17}$$

where $*$ denotes point by point multiplication. The terms Ah and Ax are the values of $H(s)$ and $X(s)$ at some points $i_1, \ldots i_{2n-1}$. The point by point multiplication gives the values $Y(i_1), \ldots, Y(i_{2n-1})$.

The operation of C obtains the coefficients of $Y(s)$ from its values at these points of evaluation. This is the bilinear form and it implies that

$$H_k = Cdiag(Ae_k)A. \tag{18}$$

where e_k is the k^{th} standard basis vector. However, A and C do not need to be Vandermonde matrices as suggested above. As long as A and C are matrices such that $H_k = Cdiag(Ae_k)A$, then the linear convolution of x and h is given by the bilinear form $y = C\{Ah * Ax\}$. More generally, as long as A, B and C are matrices satisfying $H_k = Cdiag(Be_k)A$, then $y = C\{Bh * Ax\}$ computes the linear convolution of h and x. For convenience, if $C\{Bh * Ax\}$ computes the n point linear convolution of h and x (both h and x are n point sequences), then we say "(A, B, C) describes a bilinear form for n point linear convolution." For example, (A, A, C) describes a 2 point linear convolution where

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ -1 & -1 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{19}$$

Similarly, we can write a bilinear form for cyclotomic convolution. Let d be any positive integer and let $X(s)$ and $H(s)$ be polynomials of degree $\phi(d) - 1$ where $\phi(\cdot)$ is the Euler totient function. If A, B and C are matrices satisfying $(C_{\Phi_d})^k = Cdiag(Be_k)A$ for $0 \leq k \leq \phi(d) - 1$, then the coefficients of $Y(s) = \langle X(s)H(s)\rangle_{\Phi_d(s)}$ are given by $y = C\{Bh * Ax\}$. As above, for such A, B, and C, we say "(A, B, C) describes a bilinear form for $\Phi_d(s)$ convolution."

But since $\langle X(s)H(s)\rangle_{\Phi_d(s)}$ can be found by computing the product of $X(s)$ and $H(s)$ and reducing the result, a cyclotomic convolution algorithm can always be derived by following a linear convolution algorithm by the appropriate reduction operation: If G is the appropriate reduction matrix and if (A, B, F) describes a bilinear form for a $\phi(d)$ point *linear* convolution, then (A, B, GF) describes a bilinear form for $\Phi_d(s)$ convolution. That is, $y = GF\{Bh * Ax\}$ computes the coefficients of $\langle X(s)H(s)\rangle_{\Phi_d(s)}$.

2.2 Circular Convolution

Consider p^e point circular convolution. Since $S_{p^e} = R_{p^e}^{-1}\left(\oplus_{i=0}^{e} C_{\Phi_{p^i}}\right) R_{p^e}$, the circular convolution is decomposed into a set of $e + 1$ disjoint $\Phi_{p^i}(s)$ convolutions. If $(A_{p^i}, B_{p^i}, C_{p^i})$ describes a bilinear form for $\Phi_{p^i}(s)$ convolution and if

$$A = 1 \oplus A_p \oplus \cdots \oplus A_{p^e} \tag{20}$$

$$B = 1 \oplus B_p \oplus \cdots \oplus B_{p^e} \tag{21}$$

$$C = 1 \oplus C_p \oplus \cdots \oplus C_{p^e} \tag{22}$$

where $\oplus$ denotes the matrix direct sum, then $(AR_{p^e}, BR_{p^e}, R_{p^e}^{-1}C)$ describes a bilinear form for p^e point circular convolution. In particular, if (D_d, E_d, F_d) describes a bilinear form for d point linear convolution, then A_{p^i}, B_{p^i} and C_{p^i} can be taken to be

$$A_{p^i} = D_{\phi(p^i)} \quad B_{p^i} = E_{\phi(p^i)} \quad C_{p^i} = G_{p^i} F_{\phi(p^i)} \tag{23}$$

where G_{p^i} represents the appropriate reduction operation and $\phi(\cdot)$ is the Euler totient function. Specifically, G_{p^i} has the following form

$$G_{p^i} = \begin{bmatrix} I_{(p-1)p^{i-1}} & -\underline{1}_{p-1} \otimes I_{p^{i-1}} & \begin{bmatrix} I_{(p-2)p^{i-1}-1} \\ 0_{p^{i-1}+1,(p-2)p^{i-1}-1} \end{bmatrix} \end{bmatrix} \tag{24}$$

if $p \geq 3$, while

$$G_{2^i} = \begin{bmatrix} I_{2^{i-1}} & \begin{bmatrix} -I_{2^{i-1}-1} \\ 0_{1,2^{i-1}-1} \end{bmatrix} \end{bmatrix}. \tag{25}$$

Note that the matrix R_{p^e} block diagonalizes S_{p^e} and each block on the diagonal represents a cyclotomic convolution. Correspondingly, the matrices A, B and C of the bilinear form also have a block diagonal structure.

2.3 A Matrix Formulation of the Split Nesting Algorithm

We now describe the split nesting algorithm for general length circular convolution [14]. Let $n = p_1^{e_1} \cdots p_k^{e_k}$ where p_i are distinct primes. We have seen that

$$S_n = P^t R^{-1} \left(\bigoplus_{d|n} \Psi(d) \right) RP \tag{26}$$

where P is the prime factor permutation $P = P_{p_1^{e_1},\ldots,p_k^{e_k}}$ and $R = R_{p_1^{e_1},\ldots,p_k^{e_k}}$ represents the reduction operations. For example, see eq (12). RP block diagonalizes S_n and each block on the diagonal represents a multi-dimensional cyclotomic convolution. To obtain a bilinear form for a multi-dimensional convolution, we can combine bilinear forms for one-dimensional convolutions as follows: If $(A_{p_j^i}, B_{p_j^i}, C_{p_j^i})$ describes a bilinear form for $\Phi_{p_j^i}(s)$ convolution and if

$$A = \oplus_{d|n} A_d \quad B = \oplus_{d|n} B_d \quad C = \oplus_{d|n} C_d \tag{27}$$

with

$$A_d = \otimes_{p|d, p\in\mathcal{P}} A_{H_d(p)} \tag{28}$$

$$B_d = \otimes_{p|d, p\in\mathcal{P}} B_{H_d(p)} \tag{29}$$

$$C_d = \otimes_{p|d, p\in\mathcal{P}} C_{H_d(p)} \tag{30}$$

where $H_d(p)$ is the highest power of p dividing d, and $\mathcal{P}$ is the set of primes, then $(ARP, BRP, P^tR^{-1}C)$ describes a bilinear form for n point circular convolution. That is

$$y = P^tR^{-1}C\{BRPh * ARPx\} \tag{31}$$

computes the circular convolution of h and x.

As above $(A_{p_j^i}, B_{p_j^i}, C_{p_j^i})$ can be taken to be $(D_{\phi(p_j^i)}, E_{\phi(p_j^i)}, G_{p_j^i}F_{\phi(p_j^i)})$ where (D_d, E_d, F_d) describes a bilinear form for d point *linear* convolution. This is one particular choice for $(A_{p_j^i}, B_{p_j^i}, C_{p_j^i})$ - other bilinear forms for cyclotomic convolution that are not derived from linear convolution algorithms exist [2, 14].

Example 3: A 45 point circular convolution algorithm:

$$y = P^tR^{-1}C\{BRPh * ARPx\} \tag{32}$$

where $P = P_{9,5}$, $R = R_{9,5}$,

$$\begin{aligned} A &= 1 \oplus A_3 \oplus A_9 \oplus A_5 \oplus (A_3 \otimes A_5) \oplus (A_9 \otimes A_5) && (33) \\ B &= 1 \oplus B_3 \oplus B_9 \oplus B_5 \oplus (B_3 \otimes B_5) \oplus (B_9 \otimes B_5) && (34) \\ C &= 1 \oplus C_3 \oplus C_9 \oplus C_5 \oplus (C_3 \otimes C_5) \oplus (C_9 \otimes C_5) && (35) \end{aligned}$$

and where $(A_{p_j^i}, B_{p_j^i}, C_{p_j^i})$ describes a bilinear form for $\Phi_{p_j^i}(s)$ convolution.

2.4 The Matrix Exchange Property

The matrix exchange property is a useful technique that allows one to save computation in carrying out the action of bilinear forms [8]. When h is known and fixed in eq (31), $BRPh$ can be pre-computed and stored so that y can be found using only the operations represented by $P^tR^{-1}C$ and ARP and the point by point multiplications denoted by $*$. The operation of BRP is absorbed into the multiplicative constants. Note that in eq (31), $P^tR^{-1}C$ is more complicated than is BRP and it is therefore advantageous to absorb the work of $P^tR^{-1}C$ instead of BRP into the multiplicative constants. Let J be the reversal matrix (the anti-identity matrix). Applying the matrix exchange property to eq (31) one gets

$$y = JP^tR^tB^t\left\{C^tR^{-t}PJh * ARPx\right\}. \tag{36}$$

Example 4: A 45 point circular convolution algorithm:

$$y = JP^tR^tB^t\{u * ARPx\} \tag{37}$$

where $u = C^t R^{-t} P J h$, $P = P_{9,5}$, $R = R_{9,5}$,

$$\begin{aligned} A &= 1 \oplus A_3 \oplus A_9 \oplus A_5 \oplus (A_3 \otimes A_5) \oplus (A_9 \otimes A_5) && (38) \\ B^t &= 1 \oplus B_3^t \oplus B_9^t \oplus B_5^t \oplus (B_3^t \otimes B_5^t) \oplus (B_9^t \otimes B_5^t) && (39) \\ C^t &= 1 \oplus C_3^t \oplus C_9^t \oplus C_5^t \oplus (C_3^t \otimes C_5^t) \oplus (C_9^t \otimes C_5^t) && (40) \end{aligned}$$

and where $(A_{p_j^i}, B_{p_j^i}, C_{p_j^i})$ describes a bilinear form for $\Phi_{p_j^i}(s)$ convolution.

3 A Bilinear Form for the DFT

A bilinear form for a prime length DFT can be obtained by making minor changes to a bilinear form for circular convolution. This relies on Rader's observation that a prime p point DFT can be computed by computing a $p-1$ point circular convolution and by performing some extra additions [15]. It turns out that when the Winograd or the split nesting convolution algorithm is used, only two extra additions are required.

3.1 Rader's Permutation

To explain Rader's conversion of a prime p point DFT into a $p-1$ point circular convolution [3, 15] we recall the definition of the DFT

$$y(k) = \sum_{n=0}^{p-1} x(n) W^{kn} \tag{41}$$

with $W = \exp(-\jmath 2\pi/p)$. Also recall that a primitive root of p is an integer r such that $\langle r^m \rangle_p$ maps the integers $m = 0, \ldots, p-2$ to the integers $1, \ldots, p-1$. Letting $n = r^{-m}$ and $k = r^l$, where r^{-m} is the inverse of r^m modulo p, the DFT becomes

$$y(r^l) = x(0) + \sum_{m=0}^{p-2} x(r^{-m}) W^{r^l r^{-m}} \tag{42}$$

for $l = 0, \ldots, p-2$. The 'DC' term is given by $y(0) = \sum_{n=0}^{p-1} x(n)$. By defining new functions $x'(m) = x(r^{-m})$, $y'(m) = y(r^m)$ and $W'(m) = W^{r^m}$ which are simply permuted versions of the original sequences, the DFT becomes

$$y'(l) = x(0) + \sum_{m=0}^{p-2} x'(m) W'(l-m) \tag{43}$$

for $l = 0, \ldots, p-2$. This equation describes circular convolution and therefore any circular convolution algorithm can be used to compute a prime length DFT. It is only necessary to (i) permute the input, the roots of unity and the output, *(ii)* add $x(0)$ to each term in (43) and *(iii)* compute the DC term.

Define a permutation matrix Q for the permutation above. If p is a prime and r is a primitive root of p, then let Q_r be the permutation matrix defined by $Qe_{\langle r^k\rangle_p-1} = e_k$ for $0 \leq k \leq p-2$ where e_k is the k^{th} standard basis vector. Let the $\tilde{w}$ be a $p-1$ point vector of the roots of unity: $\tilde{w} = (W^1, \ldots, W^{p-1})^t$. If s is the inverse of r modulo p (that is, $rs = 1$ modulo p) and $\tilde{x} = (x(1), \ldots, x(p-1))^t$, then the circular convolution of eq (43) can be computed with the bilinear form of eq (36):

$$Q_s^t J P^t R^t B^t \left\{ C^t R^{-t} P J Q_s \tilde{w} * A R P Q_r \tilde{x} \right\}. \tag{44}$$

This bilinear form does not compute $y(0)$, the DC term. Furthermore it is still necessary to add the $x(0)$ term to each of the elements of (44) to obtain $y(1), \ldots, y(p-1)$.

3.2 Calculation of the DC term

The computation of $y(0)$ turns out to be very simple when the bilinear form (44) is used to compute the circular convolution in eq (43). The first element of $ARPQ_r\tilde{x}$ in eq (44) is the residue modulo the polynomial $s-1$, that is, the first element of this vector is the sum of the elements of $\tilde{x}$. Therefore, the DC term can be computed by adding the first element of $ARPQ_r\tilde{x}$ to $x(0)$. Hence, when the Winograd or split nesting algorithm is used to perform the circular convolution of eq (44), the computation of the DC term requires only one extra complex addition for complex data.

The addition $x(0)$ to each of the elements of (44) also requires only one complex addition. By adding $x(0)$ to the first element of $\{C^t R^{-t} P J Q_s \tilde{w} * A R P Q_r \tilde{x}\}$ in eq (44) and applying $Q_s^t J P^t R^t$ to the result, $x(0)$ is added to each element.

Although the DFT can be computed by making these two extra additions, this organization of additions does not yield a bilinear form. However, by making a minor modification, a bilinear form can be retrieved. The method described above can be illustrated in fig 1 with $u = C^t R^{-t} P J Q_s \tilde{w}$. Clearly, the structure highlighted in the dashed box can be replaced by the structure in fig 2. By substituting the second structure for the first, a bilinear form is obtained. The resulting bilinear form for a prime length DFT is

$$y = \begin{bmatrix} 1 & \\ & Q_s^t J P^t R^t B^t \end{bmatrix} U_p^t \left\{ V_p \begin{bmatrix} 1 & \\ & C^t R^{-t} P J Q_s \end{bmatrix} w * U_p \begin{bmatrix} 1 & \\ & A R P Q_r \end{bmatrix} x \right\} \tag{45}$$

where $w = (W^0, \ldots, W^{p-1})^t$, $x = (x(0), \ldots, x(p-1))^t$, and where U_p and V_p are the matrices with the forms

$$U_p = \begin{bmatrix} 1 & 1 & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix} \qquad V_p = \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}. \tag{46}$$

4 Implementing Kronecker Products Efficiently

In the algorithm described above we encountered expressions of the form $A_1 \otimes A_2 \otimes \cdots \otimes A_n$ which we denote by $\otimes_{i=1}^n A_i$. To calculate the product $(\otimes_i A_i)x$ it is computationally advantageous to factor $\otimes_i A_i$ into terms of the form $I \otimes A_i \otimes I$ [1]. For the Kronecker product $\otimes_{i=1}^n A_i$ there are $n!$ possible different ways in which to order the operations A_i. To find the best factorization of $\otimes_i A_i$ it is necessary only to compute the ratios $(rows_i - cols_i)/cost_i$ and to order them in an non-decreasing order [1].

4.1 Vector/Parallel Interpretation

The command $I \otimes A \otimes I$ where $\otimes$ is the Kronecker (or tensor) product can be interpreted as a vector/parallel command [5, 19]. In these references, the implementation of these commands is discussed in detail and it was found that the tensor product is "an extremely useful tool for matching algorithms to computer architectures [5]."

In the programs we have written in conjunction with this paper we implement the commands $y = (I \otimes A \otimes I)x$ with loops in a set of subroutines. The circular convolution and prime length FFT programs we present, however, explicitly use the form $I \otimes A \otimes I$ to make clear the structure of the algorithm, to make them more modular and simpler, and to make them amenable to implementation on special architectures. In fact, in [5] it is suggested that it might be practical to develop tensor product compilers. The FFT programs described here will be well suited for such compilers.

5 Programs for Circular Convolution

In writing a program that computes the circular convolution of h and x using the bilinear form (36) we have written subprograms that carry out the action of P, P^t, R, R^t, A and B^t. We are assuming, as is usually done, that h is fixed and known so that $u = C^t R^{-t} P J h$ can be pre-computed and stored. To compute these multiplicative constants u we need additional subprograms to carry out the action of C^t and R^{-t} but the efficiency with which we compute u is unimportant since this is done beforehand and u is stored.

In section 1.2 we discussed the permutation P and a program implementing this permutation is not difficult to write. The reduction operations R, R^t and R^{-t} we have described in section 1.3 and programs for these reduction operations we have described above. To carry out the operation of A and B^t we need to be able to carry out the action of $A_{d_1} \otimes \cdots \otimes A_{d_k}$ and this was discussed in section 4. Note that since A and B^t are block diagonal, each diagonal block can be done separately. However, since they are rectangular, it is necessary to be careful so that the correct indexing is used.

To facilitate the discussion of the programs we generate, it is useful to consider an example. Take as an example the 45 point circular convolution algorithm. From eq (32) we find that we need to compute $x = P_{9,5}x$ and $x = R_{9,5}x$.

We noted above that bilinear forms for linear convolution, (D_d, E_d, F_d), can be used for cyclotomic convolutions - specifically we can take $A_{p^i} = D_{\phi(p^i)}$, $B_{p^i} = E_{\phi(p^i)}$ and $C_{p^i} = G_{p^i} F_{\phi(p^i)}$. In this case eq (33) becomes

$$A = 1 \oplus D_2 \oplus D_6 \oplus D_4 \oplus (D_2 \otimes D_4) \oplus (D_6 \otimes D_4). \tag{47}$$

When we use bilinear forms for convolution obtained by nesting [2] one can take $D_4 = D_2 \otimes D_2$ and $D_6 = D_2 \otimes D_3$. Then

$$A = 1 \oplus D_2 \oplus (D_2 \otimes D_3) \oplus (D_2 \otimes D_2) \oplus (D_2 \otimes D_2 \otimes D_2) \oplus (D_2 \otimes D_3 \otimes D_2 \otimes D_2). \tag{48}$$

From the discussion above, we found that the Kronecker products like $D_2 \otimes D_2 \otimes D_2$ appearing in these expressions are best carried out by factoring the product in to factors of the form $I_a \otimes D_2 \otimes I_b$. Therefore we have written programs to program to carry out $(I_a \otimes D_2 \otimes I_b)x$ and $(I_a \otimes D_3 \otimes I_b)x$. These function are called `ID2I(a,b,x)` and `ID3I(a,b,x)` and appear in the program in the appendix. The transposed form, $(I_a \otimes D_2^t \otimes I_b)x$, is called `ID2tI(a,b,x)`.

To compute the multiplicative constants we need C^t. Using $C_{p^i} = G_{p^i} F_{\phi(p^i)}$ we get

$$\begin{aligned} C^t &= 1 \oplus F_2^t G_3^t \oplus F_6^t G_9^t \oplus F_4^t G_5^t \oplus (F_2^t G_3^t \otimes F_4^t G_5^t) \oplus (F_6^t G_9^t \otimes F_4^t G_5^t) & (49) \\ &= 1 \oplus F_2^t G_3^t \oplus F_6^t G_9^t \oplus F_4^t G_5^t \oplus (F_2^t \otimes F_4^t)(G_3^t \otimes G_5^t) \oplus (F_6^t \otimes F_4^t)(G_9^t \otimes G_5^t). & (50) \end{aligned}$$

We have written programs to carry out the operation $F_{d_1} \otimes \cdots F_{d_K}$ and $G_{p_1^{e_1}} \otimes \cdots G_{p_K^{e_K}}$.

5.1 Operation Counts

Table 1 lists operation counts for some of the circular convolution algorithms we have generated. The operation counts do not include any arithmetic operations involved in the index variable or loops. They include only the arithmetic operations that involve the data sequence x in the convolution of x and h.

Table 3.5 in [14] for the split nesting algorithm gives very similar arithmetic operation counts. For all lengths not divisible by 9, the algorithms we have developed use the same number of multiplications and the same number or fewer additions. For lengths which are divisible by 9, the algorithms described in [14] require fewer additions than do ours. This is because the algorithms whose operation counts are tabulated in table 3.5 in [14] use a special $\Phi_9(s)$ convolution algorithm. It should be noted, however, that the efficient $\Phi_9(s)$ convolution algorithm of [14] is not constructed from smaller algorithms using the Kronecker product, as is ours. As we have discussed above, the use of the Kronecker product facilitates adaptation to special computer architectures and yields a very compact program with function calls to a small set of functions.

It is possible to make further improvements to the operation counts given in table 1 [13, 14]. Specifically, algorithms for prime power cyclotomic convolution based on the polynomial transform, although more complicated, will give improvements for the longer lengths listed [13, 14]. These improvements can be included in the code generating program we have developed.

6 Programs for Prime Length FFTs

Using the circular convolution algorithms described above, we can easily design algorithms for prime length FFTs. The only modifications that needs to be made involve the permutation of Rader [15] and the correct calculation of the DC term ($y(0)$). These modifications are easily made to the above described approach. It simply requires a few extra commands in the programs. Note that the multiplicative constants are computed directly, since we have programs for all the relevant operations.

In the version we have currently implemented and verified for correctness, we precompute the multiplicative constants, the input permutation and the output permutation. From eq (45), the multiplicative constants are given by $V_p(1 \oplus C^t R^{-t} P J Q_s)w$, the input permutation is given by $1 \oplus PQ_r$, and the output permutation is given by $1 \oplus Q_s^t J P^t$. The multiplicative constants, the input and output permutation are each stored as vectors. These vectors are then passed to the prime length FFT program which consists of the appropriate function calls (see the appendix). In previous prime length FFT modules, the input and output permutations are completely absorbed in to the computational instructions. This is possible because they are written as straight line code. It is possible to modify the code generating program we have implemented so that it produces straight line code and absorbs the permutations in to the computational program instructions.

In an in-place in-order prime factor algorithm for the DFT [4, 18], the necessary permuted forms of the DFT can be obtained by modifying the multiplicative constants. This can be easily done by permuting the roots of unity, w, in the expression for the multiplicative constants [4, 8], nothing else in the structure of the algorithm needs to be changed. By changing the multiplicative constants, it is not possible, however, to omit the permutation required for Rader's conversion of the prime length DFT in to circular convolution.

6.1 Operation Counts

Table 2 lists the arithmetic operations incurred by the FFT programs we have generated. Note that the number of additions and multiplications incurred by the programs we have generated are the same as previously existing programs for prime lengths up to and including 13. For $p = 17$ a program with 70 multiplications and 314 additions has been written, and for $p = 19$ a program with 76 multiplications and 372 additions has been written [9]. Thus for the length $p = 17$, the program we have generated requires fewer total arithmetic operations, while for $p = 19$, ours uses more.

The focus of [11] is the implementation of prime point FFT on various computer architectures and the advantage that can be gained from matching algorithms with architectures. Although we have not executed the programs described in this paper on special architectures, they are, as mentioned above, designed to be easily adapted to parallel/vector computers.

There are several tables of operation counts in [11], each table corresponding to a different variation of the algorithms used in that paper. For most variations, the algorithms we have described use fewer additions and fewer multiplications. According to table VIII of [11], however, a 13 point DFT can be implemented using 172 additions and 90 multiplications. This is 16 fewer additions but 50 more multiplications than the operation counts of table 2 here for this length. On many RISC processors the 13 point algorithm of [11] is the more efficient, because on such machines, multiplications can often be hidden in additions via dual operations.

6.2 On the Row-Column Method

In the context of the DFT, the row-column method computes the transform of a data array by computing the DFT of the rows and by then computing the DFT of the columns. Some algorithms for prime length FFTs for which operation counts are given [17] assume that a row-column method can be used for multi-dimensional convolution. Unfortunately, however, the convolution of two sequences can not be found in general by forming two arrays, by convolving their rows, and by then convolving their columns. It should be noted that the row-column convolution method of [11] apparently refers to the nested polynomial multiplication method.

7 Conclusion

We have found that by using the split nesting algorithm for circular convolution a new set of efficient prime length DFT modules that cover a wide variety of lengths can be developed. We have also exploited the structure in the split nesting algorithm to write a program that automatically generates compact readable code for convolution and prime length FFT programs.

The resulting code makes clear the organization and structure of the algorithm and clearly enumerates the disjoint convolutions into which the problem is decomposed. These independent convolutions can be executed in parallel and, moreover, the individual commands are of the form $I \otimes A \otimes I$ which can be executed as parallel/vector commands on appropriate computer architectures[19]. By recognizing also that the algorithms for different lengths share many of the same computational structures, the code we generate is made up of calls to a relatively small set of functions. Accordingly, the subroutines can be designed to specifically suit a given architecture.

The number of additions and multiplications incurred by the programs we have generated are the same as or are competitive with existing prime length FFT programs. We note that previously, prime length FFTs were made available for primes only up to 29. As in the original Winograd short convolution algorithms, the efficiency of the resulting prime p point DFT algorithm depends largely upon the factorability of $p-1$. For example, if $p-1$ is two times a prime, then an efficient p point DFT algorithm is more difficult to develop.

It should be noted too that the programs for convolution developed above are useful in the convolution of long integer sequences when exact results are needed. This is because all multiplicative constants in an n point integer convolution are integer multiples of $1/n$ and this division by n can be delayed until the last stage or can simply be omitted if a scaled version of the convolution is acceptable.

By developing a library of prime point FFT programs we can extend the maximum length and the variety of lengths of a prime factor algorithm or a Winograd Fourier transform algorithm. Furthermore, because the approach taken in this paper gives a bilinear form, it can be incorporated into the dynamic programming technique for designing optimal composite length FFT algorithms [7]. The programs described in this paper can also be adapted to obtain discrete cosine transform (DCT) algorithms by simply permuting the input and output sequences [6].

A A 31 Point FFT Program

As an example, we list a 31 point FFT program. The matrix D_2, used in the program, is part of the bilinear form for 2 point linear convolution in eq (19):

$$D_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

```
function y = fft31(x,u,ip,op)
% y = fft31(x,u,ip,op)
% y  : the 31 point DFT of x
% u  : a vector of precomputed multiplicative constants
% ip : input permutation
% op : output permutation

y = zeros(31,1);
x = x(ip);                                      % input permutation
x(2:31) = KRED([2,3,5],[1,1,1],3,x(2:31));      % reduction operations
y(1) = x(1)+x(2);                               % DC term calculation
% -------------------- block : 1 ------------------------------------------------
y(2) = x(2)*u(1);
% -------------------- block : 2 ------------------------------------------------
y(3) = x(3)*u(2);
% -------------------- block : 3 ------------------------------------------------
v = ID2I(1,1,x(4:5));           % v = (I(1) kron D2 kron I(1)) * x(4:5)
v = v.*u(3:5);
y(4:5) = ID2tI(1,1,v);          % y(4:5) = (I(1) kron D2' kron I(1)) * v
% -------------------- block : 6 = 2 * 3 ----------------------------------------
v = ID2I(1,1,x(6:7));           % v = (I(1) kron D2 kron I(1)) * x(6:7)
v = v.*u(6:8);
y(6:7) = ID2tI(1,1,v);          % y(6:7) = (I(1) kron D2' kron I(1)) * v
% -------------------- block : 5 ------------------------------------------------
v = ID2I(1,2,x(8:11));          % v = (I(1) kron D2 kron I(2)) * x(8:11)
v = ID2I(3,1,v);                % v = (I(3) kron D2 kron I(1)) * v
v = v.*u(9:17);
v = ID2tI(1,3,v);               % v = (I(1) kron D2' kron I(3)) * v
y(8:11) = ID2tI(2,1,v);         % y(8:11) = (I(2) kron D2' kron I(1)) * v
% -------------------- block : 10 = 2 * 5 ---------------------------------------
v = ID2I(1,2,x(12:15));         % v = (I(1) kron D2 kron I(2)) * x(12:15)
v = ID2I(3,1,v);                % v = (I(3) kron D2 kron I(1)) * v
v = v.*u(18:26);
v = ID2tI(1,3,v);               % v = (I(1) kron D2' kron I(3)) * v
y(12:15) = ID2tI(2,1,v);        % y(12:15) = (I(2) kron D2' kron I(1)) * v
% -------------------- block : 15 = 3 * 5 ---------------------------------------
v = ID2I(1,4,x(16:23));         % v = (I(1) kron D2 kron I(4)) * x(16:23)
v = ID2I(3,2,v);                % v = (I(3) kron D2 kron I(2)) * v
v = ID2I(9,1,v);                % v = (I(9) kron D2 kron I(1)) * v
v = v.*u(27:53);
v = ID2tI(1,9,v);               % v = (I(1) kron D2' kron I(9)) * v
v = ID2tI(2,3,v);               % v = (I(2) kron D2' kron I(3)) * v
y(16:23) = ID2tI(4,1,v);        % y(16:23) = (I(4) kron D2' kron I(1)) * v
% -------------------- block : 30 = 2 * 3 * 5 -----------------------------------
v = ID2I(1,4,x(24:31));         % v = (I(1) kron D2 kron I(4)) * x(24:31)
v = ID2I(3,2,v);                % v = (I(3) kron D2 kron I(2)) * v
v = ID2I(9,1,v);                % v = (I(9) kron D2 kron I(1)) * v
v = v.*u(54:80);
v = ID2tI(1,9,v);               % v = (I(1) kron D2' kron I(9)) * v
v = ID2tI(2,3,v);               % v = (I(2) kron D2' kron I(3)) * v
y(24:31) = ID2tI(4,1,v);        % y(24:31) = (I(4) kron D2' kron I(1)) * v
% ------------------------------------------------------------------------------
y(2) = y(1)+y(2);                               % DC term calculation
```

```
y(2:31) = tKRED([2,3,5],[1,1,1],3,y(2:31));        % transpose reduction operations
y = y(op);                                          % output permutation

% For complex data -
% Total Number of Real Multiplications : 160
% Total Number of Real Additions: 776
```

The multiplicative constants for the 31 point FFT are given by

```
u = [
      -1.033333333333333
       0.185592145427667*I
       0.251026872929094
       0.638094290379888
      -0.296373721102994
      -0.462201919825109*I
       0.155909426230360*I
       0.102097497864916*I
      -0.100498239164838
      -0.217421331841463
      -0.325082164955763
       0.798589508696894
      -0.780994042074251
      -0.256086011899669
       0.169494392220932
       0.711997889018157
      -0.060064820876732
      -1.235197570427205*I
      -0.271691369288525*I
       0.541789612349592*I
       0.329410560797314*I
       1.317497505049809*I
      -0.599508803858381*I
       0.093899154219231*I
      -0.176199088841836*I
       0.028003825226279*I
       1.316699050305790
       1.330315270540553
      -0.385122753006171
      -2.958666546021397
      -2.535301995146201
       2.013474028487015
       1.081897731187396
       0.136705213653014
      -0.569390844064251
      -0.262247009112805
       2.009855570455675
      -1.159348599757857
       0.629367699727360
       1.229312102919654
      -1.479874670425178
      -0.058279061554516
      -0.908786032252333
       0.721257672797977
      -0.351484013730995
      -1.113390280332076
       0.514823784254676
```

```
        0.776432948764679
        0.435329964075516
       -0.177866452687279
       -0.341206223210960
        0.257360272866440
       -0.050622276244575
       -2.745673340229639*I
        2.685177424507523*I
        0.880463026400118*I
       -5.028851220636894*I
       -0.345528375980267*I
        1.463210769729252*I
        3.328421083558774*I
       -0.237219367348867*I
       -1.086975102467855*I
       -1.665522956385442*I
        1.628826188810638*I
        0.534088072762272*I
       -3.050496586573981*I
       -0.209597199290132*I
        0.887582325001072*I
        2.019017208624242*I
       -0.143897052948668*I
       -0.659358110687783*I
        1.470398765538361*I
       -1.438001204439387*I
       -0.471517033054130*I
        2.693115935736959*I
        0.185041858423467*I
       -0.783597698243441*I
       -1.782479430727672*I
        0.127038806765845*I
        0.582111071051880*I
]
```

The input permutation is given by

```
ip = [1 2 17 9 5 3 26 29 15 8 20 6 19 10 21 11 31 16 24 28 30 7 4 18 25 13 27 14 23 12 22].
```

The output permutation is given by

```
op = [1 31 30 2 29 26 6 19 28 23 25 9 5 7 18 12 27 3 22 20 24 10 8 13 4 21 11 14 17 15 16].
```

References

[1] R. C. Agarwal and J. W. Cooley. New algorithms for digital convolution. *IEEE Trans. on Acoust., Speech, Signal Proc.*, 25(5):392–410, October 1977.

[2] R. E. Blahut. *Fast Algorithms for Digital Signal Processing.* Addison-Wesley, 1985.

[3] C. S. Burrus. Efficient Fourier transform and convolution algorithms. In Jae S. Lim and Alan V. Oppenheim, editors, *Advanced Topics in Signal Processing.* Prentice Hall, 1988.

[4] C. S. Burrus and P. W. Eschenbacher. An in-place, in-order prime factor FFT algorithm. *IEEE Trans. on Acoust., Speech, Signal Proc.*, 29(4):806–817, August 1981.

[5] J. Granata, M. Conner, and R. Tolimieri. The Tensor product: A mathematical programming language for FFTs and other fast DSP operations. *IEEE Signal Processing Magazine*, 9(1):40–48, January 1992.

[6] M. T. Heideman. Computation of an odd-length DCT from a real-valued DFT of the same length. *IEEE Trans. on Signal Processing*, 40(1):54–59, January 1992.

[7] H. W. Johnson and C. S. Burrus. The design of optimal DFT algorithms using dynamic programming. *IEEE Trans. on Acoust., Speech, Signal Proc.*, 31(2):378–387, April 1983.

[8] H. W. Johnson and C. S. Burrus. On the structure of efficient DFT algorithms. *IEEE Trans. on Acoust., Speech, Signal Proc.*, 33(1):248–254, February 1985.

[9] H. W. Johnson and S. Burrus. Large DFT Modules: 11, 13, 17, 19, 25. Technical Report 8105, Rice University, 1981.

[10] K. J. Jones. Prime number DFT computation via parallel circular convolvers. *IEE Proceedings, Part F*, 137(3):205–212, June 1990.

[11] C. Lu, J. W. Cooley, and R. Tolimieri. FFT algorithms for prime transform sizes and their implementations of VAX, IBM3090VF, and IBM RS/6000. *IEEE Trans. on Acoust., Speech, Signal Proc.*, 41(2):638–648, February 1993.

[12] D. G. Myers. *Digital Signal Processing: Efficient Convolution and Fourier Transform Techniques.* Prentice Hall, 1990.

[13] H. J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Trans. on Acoust., Speech, Signal Proc.*, 28(2):205–215, April 1980.

[14] H. J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, 1982.

[15] C. M. Rader. Discrete Fourier transform when the number of data samples is prime. *Proc. IEEE*, 56(6):1107–1108, June 1968.

[16] I. W. Selesnick and C. S. Burrus. Automating the design of prime length FFT programs. In *Proc. IEEE Int. Symp. Circuits and Systems*, volume 1, pages 133–136, 1992.

[17] R. Stasinski. Easy generation of small-n discrete Fourier transform algorithms. *IEE Proceedings, Part G*, 133(3):133–139, June 1986.

[18] C. Temperton. Implementation of a self-sorting in-place prime factor FFT algorithm. *Journal of Computational Physics*, 58:283–299, 1985.

[19] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transform and Convolution.* Springer-Verlag, 1989.

[20] S. Winograd. *Arithmetic Complexity of Computations.* SIAM, 1980.

[21] Y. Zalcstein. A note on fast cyclic convolution. *IEEE Trans. on Comput.*, 20:665–666, June 1971.

Table 1: Operation counts for split nesting circular convolution algorithms

N	muls	adds	N	muls	adds	N	muls	adds	N	muls	adds
2	2	4	24	56	244	80	410	1546	240	1640	6508
3	4	11	27	94	485	84	320	1712	252	1520	7920
4	5	15	28	80	416	90	380	1858	270	1880	9074
5	10	31	30	80	386	105	640	2881	280	2240	9516
6	8	34	35	160	707	108	470	2546	315	3040	13383
7	16	71	36	95	493	112	656	2756	336	2624	11132
8	14	46	40	140	568	120	560	2444	360	2660	11392
9	19	82	42	128	718	126	608	3378	378	3008	16438
10	20	82	45	190	839	135	940	4267	420	3200	14704
12	20	92	48	164	656	140	800	3728	432	3854	16430
14	32	170	54	188	1078	144	779	3277	504	4256	19740
15	40	163	56	224	1052	168	896	4276	540	4700	21508
16	41	135	60	200	952	180	950	4466	560	6560	25412
18	38	200	63	304	1563	189	1504	7841	630	6080	28026
20	50	214	70	320	1554	210	1280	6182	720	7790	30374
21	64	317	72	266	1250	216	1316	6328	756	7520	38144

Table 2: Operation counts for prime length FFTs

P	muls	adds	P	muls	adds	P	muls	adds
3	4	12	41	280	1140	241	3280	13020
5	10	34	43	256	1440	271	3760	18152
7	16	72	61	400	1908	281	4480	19036
11	40	168	71	640	3112	337	5248	22268
13	40	188	73	532	2504	379	6016	32880
17	82	274	109	940	5096	421	6400	29412
19	76	404	113	1312	5516	433	7708	32864
29	160	836	127	1216	6760	541	9400	43020
31	160	776	181	1900	8936	631	12160	56056
37	190	990	211	2560	12368	757	15040	76292

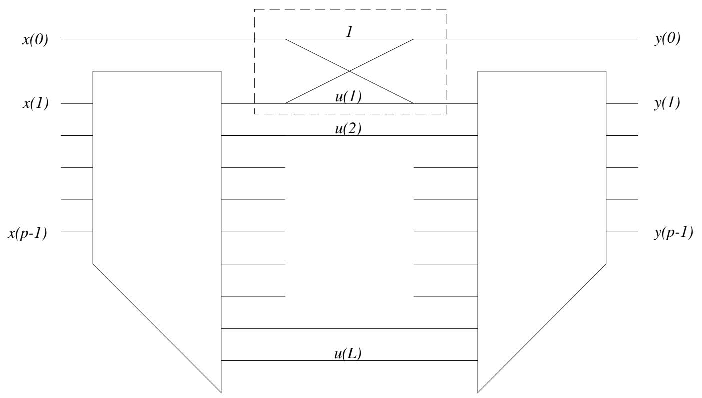

Figure 1: The flow graph for the computation of the DFT.

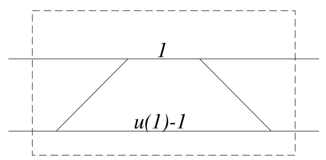

Figure 2: The flow graph for the bilinear form.

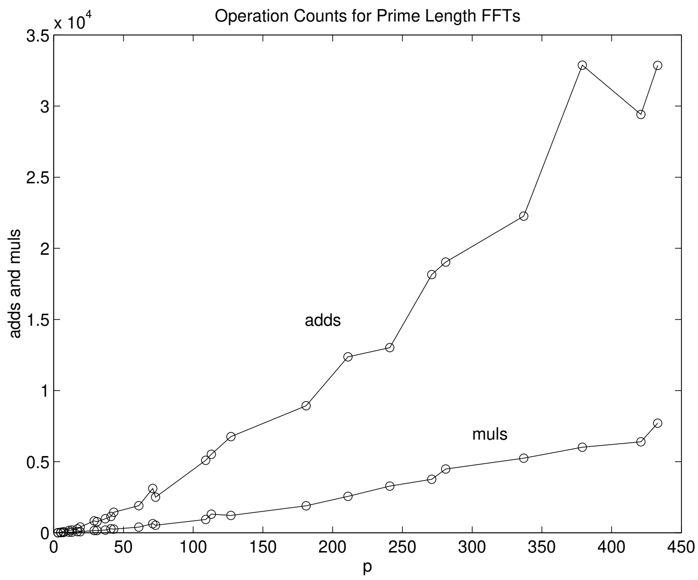

Figure 3: Plot of additions and multiplications incurred by prime length FFTs.