Parallel Computing 1 (1984) 45–63
North-Holland

FFT algorithms for vector computers

Paul N. SWARZTRAUBER

*National Center for Atmospheric Research *, Boulder, CO 80307, U.S.A.*

Received March 1984

Abstract. The adaptation of the Cooley–Tukey, the Pease and the Stockham FFT's to vector computers is discussed. Each of these algorithms computes the same result namely, the discrete Fourier transform. They differ only in the way that intermediate computations are stored. Yet it is this difference that makes one or the other more appropriate depending on the application. This difference also influences the computational efficiency on a vector computer and motivates the development of methods to improve efficiency. Each of the FFT's is defined rigorously by a short expository FORTRAN program which provides the basis for discussions about vectorization. Several methods for lengthening vectors are discussed, including the case of multiple and multi-dimensional transforms where M sequences of length N can be transformed as a single sequence of length MN using a 'truncated' FFT. The implementation of an in place FFT on a computer with memory-to-memory architecture is made possible by in place matrix-vector multiplication.

Keywords. Cooley–Tukey FFT, Pease FFT, Stockham autosort FFT, vectorization.

1. Introduction

Current vector and parallel-computer architectures are capable of computational rates that are substantially higher than their predecessors. However, in practice, maximum computational rates are rarely achieved, and a gap, which appears to be widening, exists between expected and possible computational performance. The extent of this gap depends both on the problem and the user's ability to adapt its computational solution to the particular architecture. More responsibility for performance has been placed on the user. Indeed the actual performance of a computer will likely be one to two orders of magnitude below its maximum unless the computation is formulated in a way that takes advantage of the complex architectural features.

In this paper we consider the proper formulation, or more accurately, formulations, of a particular computation, namely the complex fast Fourier transform (FFT). However, before discussing the FFT, we will consider the following factors, which influence the performance of any vector computer.

(a) On a pipeline or vector computer the total time for a vector calculation is $s + N\Delta t$, where s is the start-up time, N is the length of the vector, and Δt is the time that is required to complete any segment of an operation. Therefore, the goal is to make N as large as possible in order to reduce the significance of the start-up time, s. For large N the maximum asymptotic rate is Δt^{-1} operations per second.

* Sponsored by the National Science Foundation.

(b) In order to make efficient use of either vector or parallel architectures, the computations should be independent of one another. For example, the computations $a_i = b_i + c_i$ for $i = 1,\ldots,N$ are independent, since the a_i's can be computed at exactly the same time. On the other hand, the computations $a_i = a_{i-1} + b_i$ are dependent, since a_i cannot be computed until a_{i-1} has been, and so forth. In most cases, including this one, dependent computations can be reformulated to a large extent as independent computations.

(c) The sequence and the type of computations influence the overall efficiency of the calculation. The goal is to match operations and operands with available units and memory accesses.

(d) Computations are usually more efficient for vectors that are stored in contiguous locations. Vectors that are not stored in contiguous locations may require expensive 'scatter' or 'gather' operations.

(e) If the program is written in FORTRAN, then it should be structured so that the compiler will make use of the vector architecture. The extent to which this is possible depends on the quality of the compiler, which varies significantly among vendors.

(f) Most importantly, the selection of an algorithm will determine the extent to which one can make efficient use of a particular architecture. We will describe a number of algorithms that are efficient on vector computers.

Some interesting historical notes about the FFT are presented in [5]. The most significant event was the publication of the Cooley–Tukey FFT in 1965 [4], which marked the beginning of developments that have produced a number of algorithms for a variety of applications. We will focus on four of these algorithms, namely, the Cooley–Tukey, the Pease, the Stockham, and a Stockham variant. Of course, all of these algorithms compute the same result, namely, the discrete Fourier transform. However, they differ in how the intermediate results are computed and stored. It is these important differences that make one or the other more attractive for a particular application.

The computational phase in the Cooley–Tukey and Pease algorithms computes the transform in a permuted order. When an ordered transform is required, a separate order phase is used to unscramble the coefficients. This order phase is relatively time-consuming on a vector computer, and, as a result, the Stockham autosort algorithms have received increased attention because they compute the transform in the proper order without an explicit order phase. This property is particularly important on a vector computer, where sorting or the transfer of data, particularly between noncontiguous locations, can be just as expensive as, if not more expensive than, arithmetic operations.

In a paper that is fundamental to the early FFT literature, Cochran et al. [3] introduce an autosort FFT algorithm which they attribute to Stockham. Three recent papers by Temperton [16,17,18] examine the Stockham algorithm in some detail. In [18] he describes the mixed-radix algorithm, in which the length of the sequence N is a product of (preferably small) primes. He also discusses an efficient implementation of the algorithm, in which formulas are used for transforms whose lengths are small primes, as well as efficient methods for two-dimensional transforms. In [16] Temperton examines the performance of Winograd's formulas [20] on the CRAY-1.

The efficient application of the Stockham algorithm for real sequences on the CRAY-1 and CYBER-205 is discussed in [17]. The Stockham algorithm has also been discussed in [3,15, and 20]. Korn and Lambiotte [11] discuss the use of both the Pease [12] and the Stockham algorithms for multiple transforms and show how to transform M sequences of length N as a single sequence of length MN. They discuss the implementation of their method on the CDC STAR-100 computer. Fornberg [8] also discusses the computation of multiple transforms on the CDC STAR-100 and develops a method that is based on an algorithm of Glassman [10]. Wang

[20] develops an algorithm for multiple and multi-dimensional algorithms based on the Stockham algorithm.

Temperton has developed efficient programs for multiple transforms on the CRAY-1 and CYBER-205. For M real transforms of length N, the time per transform on the CRAY-1 is less than half a millisecond. Another package for computing the transform of a single sequence is available from the National Center for Atmospheric Research. This package, called FFTPACK, contains 19 programs for transforming real and complex sequences, as well as sine, cosine and quarter wave transforms.

The transform of symmetric sequences, such as odd or even sequences and including real sequences, will not be discussed in this paper except to note that the transform of a real sequence takes about half the time required by a complex sequence. If the length N of the real sequence x_n is an even integer, than the transform of the complex sequence $c_n = x_{2n} + ix_{2n+1}$, with length $N/2$, can be postprocessed into the transform of a real sequence. A similar algorithm exists for the transform of two real sequences. The transforms of real and other symmetric sequences are discussed in [1,2,6,7,14,15 and 17]. The vector properties of the symmetric FFT algorithms are discussed in [15]. For the most part, the vector properties of the symmetric transforms are determined by the vector properties of the underlying complex transform.

In order to develop notation and to discuss the vectorization of the FFT, it is necessary to review the algorithm itself. In the next section we discuss the Cooley–Tukey algorithms. Particular attention is directed to the intermediate sequences and transformations that are computed in the FFT, since their storage patterns determine the vector properties of the algorithm. These patterns, as well as the algorithm itself, are rigorously defined by a short FORTRAN program. The intermediate calculations and storage patterns are illustrated in tabular form for a random sequence of length $N = 8$. Throughout this paper we assume that N is a power of 2; however, all of the results can be generalized to the case in which N is a product of primes.

In Section 3 we discuss the vectorization of the FFT, with particular attention to the Cooley–Tukey algorithm. The Cooley–Tukey algorithm and the closely related Gentleman–Sande algorithm are unique in the sense that they can be performed in place without additional storage. However, for the memory-to-memory architecture of the CYBER-205, the inplace algorithm requires some additional effort. Also in Section 3, we show how to lengthen the vectors in both the Cooley–Tukey and the Stockham algorithms. We also show how to eliminate ordering for applications such as solving ordinary and partial-differential equations, where the transform itself is not visible to the user.

The Pease algorithm is discussed in Section 4 and a short expository FORTRAN program, as well as sample calculations, is given. The Pease algorithm has the longest vectors but requires more storage and the transform may have to be ordered, depending on the application.

The Stockham algorithms are presented in Section 5. These algorithms are auto-sorting and, therefore, quite attractive for use on a vector computer. However, they cannot be performed in place, which becomes a concern for two- and three-dimensional transforms, where a large number of transforms must be performed.

Multiple and multi-dimensional transforms are discussed in Section 6. Of particular interest are the 'truncated' FFT algorithms which permit the transform of M sequences of length N as a single sequence of length MN.

2. The Cooley–Tukey algorithm

In this section we will review the Cooley–Tukey FFT algorithm in order to define notation and to develop a framework on which to build a vectorizing strategy. We begin with a definition of the discrete Fourier transform. Given the complex sequence x_n, then the discrete Fourier transform X_k is given by

$$X_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n \, e^{-ikn(2\pi/N)}. \tag{1}$$

The discrete inverse Fourier transform x_n is given by

$$x_n = \sum_{k=0}^{N-1} X_k \, e^{ikn(2\pi/N)}. \tag{2}$$

The Cooley–Tukey algorithm for the FFT is a straightforward extension of the following splitting algorithm. The motivation for the splitting algorithm is simply that it cuts the computation from (1) in half. If N is even, then we can split the sum on the right side of (1) into two sums in which the subscripts are even and odd.

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} \, e^{-ik2n(2\pi/N)} + \sum_{n=0}^{N/2-1} x_{2n+1} \, e^{-ik(2n+1)(2\pi/N)}. \tag{3}$$

It is customary in the development of the FFT to ignore the scale factor of $1/N$ in (1). If we now define

$$Y_k = \sum_{n=0}^{N/2-1} x_{2n} \exp(-ikn2\pi/\tfrac{1}{2}N), \tag{4}$$

$$Z_k = \sum_{n=0}^{N/2-1} x_{2n+1} \exp(-ikn2\pi/\tfrac{1}{2}N) \tag{5}$$

then (3) has the form

$$X_k = Y_k + e^{-ik(2\pi/N)} Z_k \quad (k = 0, \ldots, N/2-1). \tag{6}$$

From (4) and (5) it can be determined that Y_k and Z_k are periodic sequences with period $N/2$. That is, $Y_{k+N/2} = Y_k$ and $Z_{k+N/2} = Z_k$. Also, since $\exp[-i(k + N/2)(2\pi/N)] = -\exp[-ik(2\pi/N)]$

$$X_{k+N/2} = Y_k - e^{-ik(2\pi/N)} Z_k \quad (k = 0, \ldots, N/2-1). \tag{7}$$

The splitting algorithm for computing X_k can now be given.

(i) Compute Y_k and Z_k from equations (4) and (5). This requires $N(N/2 + 1)$ multiplications and $N(N/2 - 1)$ additions.

(ii) Compute X_k from (6) and (7). This requires $N/2$ multiplications, $N/2$ additions, and $N/2$ subtractions.

The motivation for the splitting algorithm is now evident. For large N the computation in step (ii) is negligible compared to step (i). Therefore, the total number of operations that are required to compute X_k, using the splitting algorithm, is about half the number required to compute X_k directly from (1).

The Cooley–Tukey FFT algorithm results from the observation that (4) and (5) have the same form as (1) without the scale factor $1/N$. Therefore, the splitting algorithm can also be used to compute both Y_k and Z_k, thereby reducing the time required to compute them as well as

X_k. If $N = 2^m$ for some m, then the splitting algorithm can be applied m times.

The Cooley–Tukey algorithm for a random real sequence of length $N = 2^3 = 8$ is given in Table 1. The original sequence, x_n, is in column 1. In column 2 the sequence has been split into two sequences, x_{2n}, x_{2n+1}, with even and odd subscripts, respectively. Further splittings are given in columns 3 and 4. Since by definition (1), the transform of a sequence of length 1 is just itself, column 4 also contains the transforms. Four transforms of length 2 are computed in column 5. Each transform is computed from two transforms of length 1 in column 4 using equations (6) and (7) with $N = 2$. Next, two transforms of length 4 are computed in column 6. Each transform is computed from two transforms of length 2 in column 5 using (6) and (7) with $N = 4$. These transforms are Y_k and Z_k, respectively, which were defined above. Finally, X_k is computed in column 7 from Y_k and Z_k in column 6, using (6) and (7) with $N = 8$.

For the case $N = 2^m$, the FFT requires $mN/2 = (N/2)\log_2 N$ complex additions, subtractions, and multiplications. Note that the sequences in columns 5, 6, and 7 are the FFT's of the sequences in columns 3, 2, and 1, respectively. The permutations of the original sequence that occur in columns 2, 3, and 4 are called the order phase of the FFT. The computation of columns 5, 6, and 7 by repeated application of the splitting algorithm is referred to as the combine phase of the FFT.

In later sections we will define several FFT algorithms which have properties that make one or the other more attractive for a particular application. The only difference among these algorithms is the way in which the intermediate sequences and their corresponding transforms are stored.

In order to compare these storage patterns, it will be useful to rigorously define the Cooley–Tukey FFT, and to this end we will present a FORTRAN program. The program is for expository purposes only and is not intended to represent state-of-the art software for a vector computer. Current FFT packages consist of several thousand lines of code which perform a variety of Fourier transforms [15].

A program is given below for the order phase of the Cooley–Tukey algorithm, which corresponds to the orderings that are generated in columns 1 through 4 of Table 1 for the case $N = 8 = 2^3$. The original sequence is stored in the array C, which is overwritten on exit by the ordered array. A complex work array $WORK$ with N locations must be provided. NS is the number of sequences in the Lth column, and LS is the length of the sequences in the $(L+1)$st column.

Table 1
The Cooley–Tukey FFT for $N = 8$

1	2	3	4	5	6	7
0.580	0.580	0.580	0.580	1.034	2.096 + 0.000i	3.656 + 0.000i
0.951	0.786	0.454	0.454	0.126	0.126 − 0.511i	0.800 − 1.173i
0.786	0.454	0.786	0.786	1.062	−0.028 + 0.000i	−0.028 − 0.354i
0.298	0.276	0.276	0.276	0.511	0.126 + 0.511i	−0.547 − 0.151i
0.454						0.536 + 0.000i
0.006	0.951	0.951	0.951	0.957	1.560 + 0.000i	−0.547 + 0.151i
0.276	0.298	0.006	0.006	0.944	0.944 + 0.008i	−0.028 + 0.354i
0.306	0.006	0.298	0.298	0.603	0.354 + 0.000i	0.800 + 1.173i
	0.306	0.306	0.306	−0.008	0.944 − 0.008i	

```
      SUBROUTINE CTSORT (M,C,WORK)
C
C     ORDER PHASE FOR THE COOLEY-TUKEY FFT
C
      COMPLEX C(1), WORK(1)
      N = 2**M
      DO 100 L = 1,M
      NS = 2**(L-1)
      LS = N/(LS+LS)
      CALL CTSRT1 (LS,NS,C,WORK)
      DO 100 I = 1,N
      C(I) = WORK(I)
  100 CONTINUE
      RETURN
      END
      SUBROUTINE CTSRT1 (LS,NS,C,CH)
      COMPLEX C(2,LS,NS), CH(LS,2,NS)
      DO 200 J = 1,NS
      DO 200 I = 1,LS
      CH(I,1,J) = C(1,I,J)
      CH(I,2,J) = C(2,I,J)
  200 CONTINUE
      RETURN
      END
```

The first four columns of Table 1 are generated in loop 100. The key loop is 200, in which the Jth sequence of length $2LS$ is split into two sequences, each of length LS. The ordered array is stored in the original array after returning to CTSORT in loop 100.

With the advent of the vector computer, the time required by the order phase has increased in relation to the time required by the combine phase. We will show that in most cases the order phase of the FFT can be avoided. Nevertheless, we have presented the order phase for the Cooley–Tukey algorithm since it defines the way in which the intermediate sequences and their transforms are stored. These storage patterns vary from one FFT algorithm to the other and indeed serve to define the FFT algorithm.

The program for the combine phase is given below. $IS = -1$ corresponds to the forward transform given in equation (1), but without the scale factor of $1/N$. It is assumed that the input array C has been ordered by a call to subroutine CTSORT. On exit the unscaled transform is returned in the C array. Note that a work array is not required. Probably the most important property of the Cooley–Tukey algorithm is that it can be performed in place without additional storage. However, with the advent of certain vector-computer architecture, it has become more difficult to code an in place FFT. This topic, together with an in place matrix vector multiply algorithm, will be discussed in the next section. Columns 5 through 7 in Table 1 are computed in the program that follows. A call of CTCMBN with $IS = -1$ computes NX_k where X_k is defined in (1). The inverse transform (2) is computed when $IS = 1$.

```
      SUBROUTINE CTCMBN (IS,M,C)
C
C     COMBINE PHASE FOR THE COOLEY-TUKEY FFT
C
      COMPLEX C(1)
```

```
      N = 2 * * M
      DO 100 L = 1,M
      LS = 2 * *(L - 1)
      NS = N/(LS + LS)
      CALL CTCMB1 (IS,LS,NS,C)
  100 CONTINUE
      RETURN
      END
      SUBROUTINE CTCMB1 (IS,LS,NS,C)
      COMPLEX OMEGA,OMEGK,WYK,C(LS,2,NS)
      ANGLE = FLOAT(IS) * 4. * ATAN(1.)/FLOAT(LS)
      OMEGA = CMPLX(COS(ANGLE),SIN(ANGLE))
      DO 200 J = 1,NS
      OMEGK = 1.
      DO 200 I = 1,LS
      WYK = OMEGK * C(I,2,J)
      C(I,1,J) = C(I,1,J) + WYK
      C(I,2,J) = C(I,1,J) - WYK
      OMEGK = OMEGA * OMEGK
  200 CONTINUE
      RETURN
      END
```

Loop 100 on L corresponds to the column index. For the case $M = 3$ columns 5, 6, and 7 in Table 1 correspond to $L = 1$, 2, and 3. The splitting formulas in equations (6) and (7) are implemented in loop 200.

In conclusion, a few additional remarks about the Cooley–Tukey FFT are presented.

– The order phase of the Cooley–Tukey FFT as implemented in subroutime CTSORT is not in place since it uses the additional array *WORK*. Of course, the ordering can be done in place using the bit-reversal algorithm when N is a power of 2 or using Singleton's method [13] when N is a product of squared primes. To some extent the issues associated with ordering are academic since ordering can be eliminated by using the autosort algorithms that are presented in Section 5 or by using a method which is applicable when the transform itself is not visible to the user. This method is discussed in the next section.

– The multiplicative recurrence relation for computing OMEGK in loop 200 will not vectorize. The usual approach is to compute and store these values in an array using the vectorized trigonometric functions. Once these values are tabulated, they can be used repeatedly to compute subsequent transforms, as long as N remains unchanged.

3. Vectorizing the Cooley–Tukey algorithm

In this section we discuss several methods that can be used to improve the performance of the Cooley–Tukey FFT on a vector computer. Some of these methods are also applicable to the FFT algorithms that are given in the later sections. The discussion on how to eliminate the order phase is also applicable to the Pease FFT in the next section, and the discussion on how to increase the length of the vectors also applies to the autosort FFT's in Section 5. However, we begin with a topic that applies only to the in-place algorithms and, in particular, to the Cooley–Tukey FFT.

A significant, perhaps the only, advantage of the Cooley–Tukey algorithm is that it can be performed in place without additional storage. However, the situation has become somewhat more complex in the context of vector and pipeline architectures. To see this we turn our attention to loop 200 of subroutine CTCMB1 in the previous section and, in particular, to the first statement inside the loop in which WYK is computed. Earlier FORTRAN compilers on the CRAY-1 would not vectorize this loop, since the computation of WYK was not independent of the rest of the computation in the loop. However, later versions of the compiler produce vector code by storing WYK as an array in the CRAY-1 vector registers.

On the CDC CYBER-205, it is necessary to explicitly designate WYK as an array, $WYK(I)$, since the architecture is memory-to-memory rather than memory-to-register. The significance of this observation is that the Cooley–Tukey FFT cannot be performed in place on the CYBER-205. If we choose to overwrite $C(I,2,J)$ with $WYK(I)$ and then to overwrite $C(I,1,J)$ with $C(I,1,J) - WYK(I)$, then $C(I,1,J)$ is not available for computing $C(I,2,J)$. The difficulty can be expressed somewhat more succinctly. Assume that we have two arrays, $A(I)$ and $B(I)$, and that we wish to compute two other arrays, $C(I)$ and $D(I)$, given by

$$C(I) = A(I) + B(I), \tag{8}$$

$$D(I) = A(I) - B(I). \tag{9}$$

Suppose further that we wish to do this calculation in place, with $C(I)$ and $D(I)$ overwriting $A(I)$ and $B(I)$. If $C(I)$ overwrites either $A(I)$ or $B(I)$, then one or the other is not available for computing $D(I)$. On the other hand, Temperton (private communication) has observed that if $C(I)$ overwrites $A(I)$, then $D(I)$ can still be computed as

$$D(I) = A(I) - B(I) - B(I), \tag{10}$$

which can then overwrite $B(I)$.

More generally, if N has a prime factor p, then the equations (8) and (9) take the form of a matrix vector product and the problem becomes one of computing this product in place. Given a p vector x and a $p \times p$ unitary matrix A, then we wish to compute

$$y = Ax \tag{11}$$

without the use of additional storage by overwriting x with y. This problem can be solved by using the LU decomposition of A. Let $A = LU$ be the decomposition of U into lower and upper triangular matrices; then

$$y = L(Ux). \tag{12}$$

This provides a way of computing y in place, since both $z = Ux$ and $y = Lz$ can be computed in place.

This procedure for in place matrix-vector multiplication is quite general. Unlike the case in which the LU decomposition is used for the solution of a linear system of equations, the matrix product can be computed even when A is a singular matrix. However, some of the bottom rows of U will be zero. For our application, A is well conditioned. It is a unitary matrix with order p, which is chosen as a small prime in order to maintain the efficiency of the FFT. Nevertheless, for problems in which repeated transforms are made, it is worth noting that the error properties of this algorithm will be different from those of the Cooley–Tukey algorithm.

We now turn our attention to the order phase of the FFT, which is expensive on a vector computer, where the movement of data can be almost as time consuming as the arithmetic operations. Fortunately, in most cases ordering can be avoided. In Section 5 we will discuss the Stockham autosort algorithms, in which the order phase is eliminated. Unfortunately, the Stockham algorithms cannot be performed in place, and therefore they may not be the most appropriate algorithms for all applications.

We will now discuss a method for eliminating the order phase that is applicable to a very large class of problems. This class includes any problem in which the transform X_k is not visible to the user. For example, suppose we are given the sequence x_n, which is a discrete approximation to some periodic function $x(\theta)$. Suppose further that we wish to determine an approximate derivative of this function. We can proceed by using the FFT to compute the transform X_k of the sequence x_n. We can then formally differentiate the trigonometric approximation to $x(\theta)$ by computing ikX_k. Next, we use the inverse FFT to tabulate an approximate derivative, $\dot{x}_n$. Note that it is important to formally differentiate the unaliased form of the discrete Fourier transform and not the aliased form (1); see [15].

This application is like most applications of the Fourier method in which interest is focused on obtaining a solution in physical rather than Fourier space, and the Fourier transform X_k remains invisible. Other applications include solving ordinary and partial-differential equations, filtering, and pattern recognition. Before we show how to use this fact to eliminate ordering, it will be necessary to derive another FFT algorithm. We begin with a discussion of inverse FFT's.

There are two ways to generate an inverse transform from the forward transform. By comparing equations (1) and (2) we observe that one way is simply to change the sign of the exponent. We have made use of this in the FORTRAN program CTCMBN where the parameter *IS* determines the sign of the exponent and thus determines whether the transform or its inverse is to be computed. Another way to compute the inverse transform is to reverse the order of the operations in the Cooley–Tukey algorithm. That is, we assume that column 7 in Table 1 is given and then proceed to reconstruct the columns in reverse order until we obtain the original sequence in column 1. This procedure requires the inverse splitting algorithm which is obtained by inverting equations (6) and (7)

$$Y_k = \tfrac{1}{2}\left(X_k + X_{k+N/2}\right), \tag{13}$$

$$Z_k = \tfrac{1}{2}e^{ik(2\pi/N)}\left(X_k - X_{k+N/2}\right). \tag{14}$$

If both methods of inversion are applied to the Cooley–Tukey algorithm we obtain a new forward transform which is called the Gentleman–Sande FFT [9]. This is indeed a different FFT algorithm since its combine phase precedes its order phase. This will be important in our effort to eliminate ordering. The double inversion method can also be used to generate new FFT algorithms from the remaining three FFT algorithms that will be presented in this paper.

We will now show, by an example, how the Cooley–Tukey and Gentleman–Sande FFT's can be used to eliminate the order phase for the class of applications that was discussed above. Again consider the application in which we wish to approximate the derivative of a discrete function, x_n. Two methods will be described. In the first method we use the combine phase of the Gentleman–Sande FFT to compute the transform $X_{p(k)}$ in a permuted order where $p(k)$ is a known permutation of the integers 0 through $N-1$. Next we use the order phase of the Gentleman–Sande FFT to obtain the ordered transform X_k. We now formally differentiate the trigonometric representation of the discrete function by computing ikX_k. Next, we use the order phase of the inverse Cooley–Tukey FFT to compute $ip(k)X_{p(k)}$ and, finally, the combine phase of the inverse Cooley–Tukey FFT to tabulate the approximate derivative $\dot{x}_n$.

In the second method we make use of the fact that the order phases of the Gentleman–Sande and Cooley–Tukey FFT's are the inverses of one another. Therefore, both of the order phases can be eliminated if we compute with the unordered coefficients in Fourier space. As in the first method, we begin with the combine phase of the Gentleman–Sande FFT to compute $X_{p(k)}$. Next we compute $ip(k)X_{p(k)}$ and, finally, $\dot{x}_n$ using the combine phase of the Cooley–Tukey FFT. In this manner, both order phases can be eliminated for any application in which the X_k are 'invisible'.

The final topic to be discussed in this section is the lengthening of the inner loops in the

FFT. The Cooley–Tukey and Stockham autosort algorithms given in Section 5 are 'pseudo' vectorizable in the sense that the lengths of the sequences or vectors change as the calculation proceeds. This becomes evident when we examine the inner loop on index I in loop 200 in the Cooley–Tukey subroutine CTCMB1 in the previous section. For $L = 1$ the length LS of the inner loop is 1, with the result that the calculation is inefficient on a vector computer. The minimum length of the inner loop can be increased to about $N^{1/2}$ by the following procedure: first, the nested loops 200 are duplicated, but with the J and I loops interchanged. Next, an IF statement is inserted prior to this code which compares the relative size of LS and NS and executes the code with the largest inner loop. This procedure is called loop inversion and results in a 40% decrease in computing time on the CRAY-1. Loop 200 in CTCMB1 is replaced by the following code:

```
      IF (LS.LT.NS) GO TO 300
      DO 200 J = 1,NS
      OMEGK = 1.
      DO 200 I = 1,LS
      WYK = OMEGK * C(I,2,J)
      C(I,1,J) = C(I,1,J) + WYK
      C(I,2,J) = C(I,2,J) - WYK
      OMEGK = OMEGA * OMEGK
  200 CONTINUE
      RETURN
  300 OMEGK = 1.
      DO 500 I = 1,LS
      DO 400 J = 1,NS
      WYK = OMEGK * C(I,2,J)
      C(I,1,J) = C(I,1,J) + WYK
      C(I,2,J) = C(I,2,J) - WYK
  400 CONTINUE
      OMEGK = OMEGA * OMEGK
  500 CONTINUE
```

Note that the data in loop 400 are not accessed or stored in contiguous locations. Thus the loop inversion must be accompanied by scatter-gather operations on the CYBER-205. This reduces the efficiency of loop 400 and will influence the crossover point at which loop 500 is computed in preference to loop 200.

4. The Pease algorithm

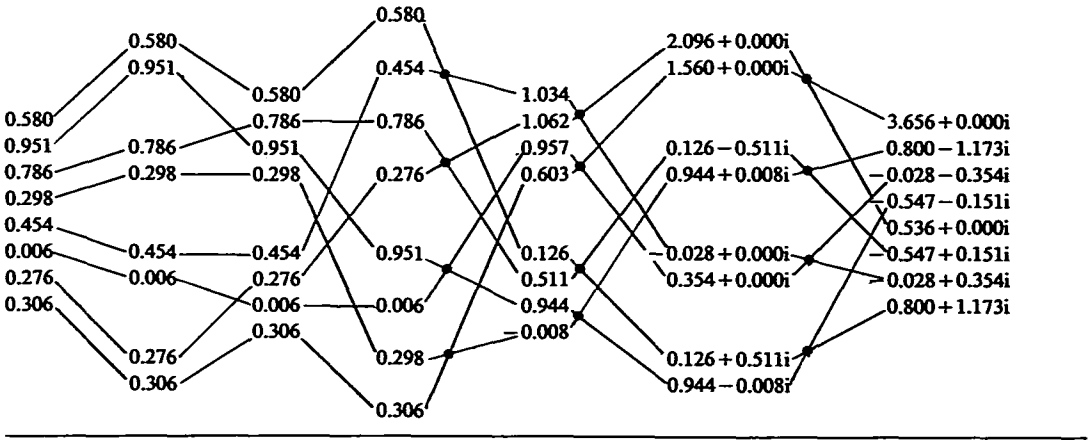
In this section we will describe the Pease FFT algorithm, which differs from the Cooley–Tukey algorithm in the following two ways: its vector lengths do not decrease as the computation proceeds, and it requires additional storage in proportion to $N \log_2 N$. The Pease algorithm is derived from the Cooley–Tukey algorithm simply by changing the way in which the intermediate sequences are stored. If we compare column 2 in Tables 1 and 2, we observe that in the Pease algorithm both first elements appear first, followed by both second elements, and so forth. If we define the elements in a column of the Cooley–Tukey algorithm by $c_{i,j}$, where i ranges over the length of a sequence and j ranges over the number of sequences, then these elements are transposed in the Pease algorithm and stored as $c_{j,i}$.

The final orderings produced by the Pease and Cooley–Tukey FFT are the same. Neverthe-

less, we choose to differentiate between these algorithms in an effort to clearly define the way in which the intermediate sequences and their transforms are stored. The difference is evident when the FORTRAN program given below is compared with the program for the Cooley–Tukey algorithm. We begin with the order phase of the Pease algorithm.

```
      SUBROUTINE PSORT (M,C,WORK)
C
C     ORDER PHASE FOR THE PEASE FFT
C
      COMPLEX C(1), WORK(1)
      N = 2**M
      DO 100 L = 1,M
      NS = 2**(L-1)
      LS = N/(NS+NS)
      CALL PSRT1 (LS,NS,C,WORK)
      DO 100 I = 1,N
      C(I) = WORK(I)
  100 CONTINUE
      RETURN
      END
      SUBROUTINE PSRT1 (LS,NS,C,CH)
      COMPLEX C(NS,2,LS), CH(2,NS,LS)
      DO 200 J = 1,NS
      DO 200 I = 1,LS
      CH(1,J,I) = C(J,1,I)
      CH(2,J,I) = C(J,2,I)
  200 CONTINUE
      RETURN
      END
```

Table 2
The Pease FFT for $N = 8$



Columns 2, 3, and 4 of Table 2 are computed in loop 100. In loop 200 each sequence is split into two sequences that are stored in the work array. These sequences are stored in the original array, C, following the return to PSORT. The combine phase, in which columns 5, 6, and 7 of Table 2 are computed, is given below.

```
      SUBROUTINE PCMBN (IS,M,C,WORK)
C
C     COMBINE PHASE FOR THE PEASE FFT
C
      COMPLEX C(1), WORK(1)
      N = 2**M
      DO 100 L = 1,M
      LS = 2**(L-1)
      NS = N/(LS+LS)
      CALL PCMB1 (IS,LS,NS,C,WORK)
      DO 100 I = 1,N
      C(I) = WORK(I)
  100 CONTINUE
      RETURN
      END
      SUBROUTINE PCMB1 (IS,LS,NS,C,CH)
      COMPLEX OMEGA,OMEGK,WYK,C(2,NS,LS),CH(NS,LS,2)
      ANGLE = FLOAT(IS)*4.*ATAN(1.)/FLOAT(LS)
      OMEGA = CMPLX(COS(ANGLE),SIN(ANGLE))
      DO 100 J = 1,NS
      OMEGK = 1.
      DO 100 I = 1,LS
      WYK = OMEGK*C(2,J,I)
      CH(J,I,1) = C(1,J,I)+WYK
      CH(J,I,2) = C(1,J,I)-WYK
      OMEGK = OMEGA*OMEGK
  100 CONTINUE
      RETURN
      END
```

In loop 100 two transforms of length LS are combined into a single transform of length $2LS$, using equations (6) and (7). At this point the advantage of the Pease algorithm is not evident, since the inner loop of loop 100 changes in length, as in the Cooley–Tukey FFT. However, we note that within this loop the indices J,I always occur together and in the same order. Hence, they can be replaced by a single index, which we will define as JI and which ranges from 1 to $LS*NS = N/2$.

In order to implement this change, we must first precompute $OMEGA**(I-1)$ and also make copies for each sequence and each column or index, L. This computation does not need to be repeated as long as N remains unchanged and can therefore be included in a separate program. Note that $\frac{1}{2}N \log_2 N$ locations must be provided to store these values. $NS2$ should be set equal to $N/2$ and the trigonometric tables are stored in the array $OMEGK$.

```
      SUBROUTINE CANGLE (IS,M,NS2,OMEGK)
C
C     CANGLE INITIALIZES THE TRIGONOMETRIC TABLES FOR
C     THE PEASE ALGORITHM
```

```
C
      COMPLEX OMEGK(NS2,1)
      N = 2**M
      DO 100 L = 1,M
      LS = 2**(L-1)
      NS = N/(LS+LS)
      CALL INITP1 (IS,LS,NS,OMEGK(1,L))
  100 CONTINUE
      RETURN
      END
      SUBROUTINE INITP1 (IS,LS,NS,OMEGK)
      COMPLEX OMEGA,OMEGK(NS,LS)
      ANGLE = FLOAT(IS)*4.*ATAN(1.)/FLOAT(LS)
      OMEGA = CMPLX(COS(ANGLE),SIN(ANGLE))
      OMEGH = 1.
      DO 200 I = 1,LS
      D) 300 J = 1,NS
      OMEGK(J,I) = OMEGH
  300 CONTINUE
      OMEGH = OMEGA*OMEGH
  200 CONTINUE
      RETURN
      END
```

Once the array *OMEGK* is computed, then the combine phase of the Pease algorithm has the form

```
      SUBROUTINE PCMBNL(M,C,WORK,NS2,OMEGK)
C
C     COMBINE PHASE FOR LONG LOOP PEASE FFT
C
      COMPLEX C(1),WORK(1),OMEGK(NS2,1)
      N = 2**M
      CALL PCMBL1(NS2,OMEGK(1,L),C,WORK)
      DO 100 I = 1,N
      C(I) = WORK(I)
  100 CONTINUE
      RETURN
      END
      SUBROUTINE PCMBL1(NS2,OMEGK,C,CH)
      COMPLEX C(2,NS2),CH(NS2,2),OMEGK(1),WYK
      DO 200 JI = 1,NS2
      WYK = OMEGK(JI)*C(2,JI)
      CH(JI,1) = C(1,JI)+WYK
      CH(JI,2) = C(1,JI)-WYK
  200 CONTINUE
      RETURN
      END
```

In conclusion, several additional remarks about the Pease algorithm are presented.

– In order to obtain the long vectors, it is necessary to initialize and store the angles, which requires $\frac{1}{2}N \log_2 N$ locations.

– The Pease algorithm cannot be performed in place, and therefore an additional N locations are required for work storage.

– If the transform must be ordered using the order phase, then there is no advantage to using the Pease algorithm, since the vectors in the order phase get small as the computation proceeds. However, recall from Section 2 that ordering can be eliminated in any application where the transform X_k is not visible to the user.

– The restore-data transfer at the end of loop 100 can be eliminated by duplicating the CALL PCMBL1 statement, but with the arguments C and $WORK$ interchanged. These two call statements are executed alternately, with a final restore required after loop 100 only if M is an odd integer.

5. The Stockham autosort algorithms

The Stockham FFT algorithms are distinguished by the fact that the transform is computed in the proper order without an explicit order phase. This is a significant advantage on a vector computer, where the order phase requires a substantial amount of computing time. Indeed, one might ask why we even consider the other algorithms. The answer is that each of the four transforms discussed in this paper has characteristics that may make it more appropriate for a certain application than the other three, depending on the application. For example, the vectors in the Stockham algorithms change in length as the calculation proceeds (this is different from the Pease algorithm and the same as the Cooley–Tukey). Also, the Stockham algorithms cannot be performed in place (this is like the Pease algorithm and unlike the Cooley–Tukey). Nevertheless, the Stockham algorithm is attractive, particularly for general-purpose software, when one cannot predict whether or not ordering will be necessary. Again, like all the other FFT's, the Stockham algorithms have a unique way of storing the intermediate sequences and their transforms.

A variant of the Stockham algorithm is given in Table 3. Since the result of the ordering phase in column 4 is the same as the order of the original sequence, the order phase can be eliminated. We note, however, that the intermediate sequences are not in the original order; yet knowledge of these intermediate storage patterns is required for a method of transforming multiple sequences which is given in the next section. For this reason we will rigorously define these patterns in the following program.

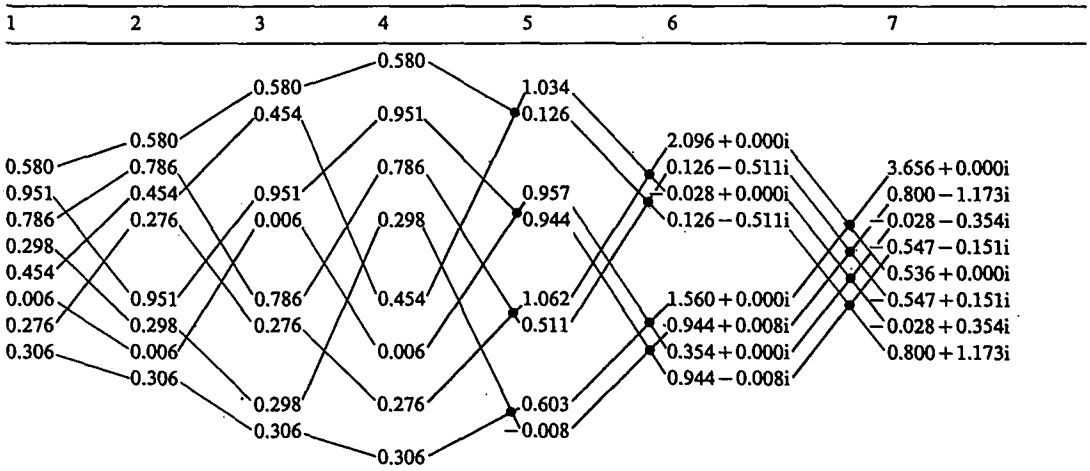
```
      SUBROUTINE VSSORT (M,C,WORK)
C
C     ORDER PHASE FOR THE STOCKHAM VARIANT AUTO-SORT
C     FFT (NOT USED FOR A SINGLE TRANSFORM)
C
      COMPLEX C(1), WORK(1)
      N = 2**M
      DO 100 L = 1,M
      NS = 2**(L-1)
      LS = N/(LS+LS)
      CALL VSSRT1 (LS,NS,C,WORK)
      DO 100 I = 1,N
```

```
      C(I) = WORK(I)
  100 CONTINUE
      RETURN
      END
      SUBROUTINE VSSRT1 (LS,NS,C,WORK)
      COMPLEX C(2,LS,NS), CH(LS,2,NS)
      DO 200 J = 1,NS
      DO 200 I = 1,LS
      CH(I,J,1) = C(1,I,J)
      CH(I,J,2) = C(2,I,J)
  200 CONTINUE
      RETURN
      END
```

A FORTRAN program for the variant of the Stockham algorithm is given below. The order of the subscripts on the variables C and CH serves to distinguish this FFT algorithm from the others.

```
      SUBROUTINE STOCKV (IS,M,C,WORK)
C
C     VARIANT OF STOCKHAM AUTO-SORT FFT
C
      COMPLEX C(1), WORK(1)
      N = 2**M
      DO 300 L = 1,M
      LS = 2**(L-1)
      NS = N/(LS+LS)
      CALL STCKV1 (IS,LS,NS,C,WORK)
      DO 300 I = 1,N
      C(I) = WORK(I)
  300 CONTINUE
      RETURN
      END
```

Table 3
The Stockham Variant FFT for $N = 8$



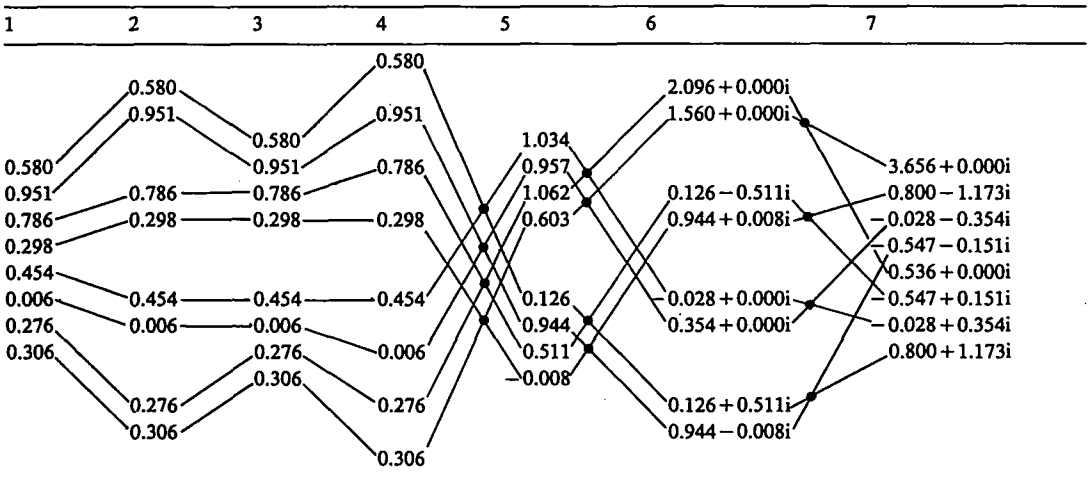
```
      SUBROUTINE STCKV1 (IS,LS,NS,C,CH)
      COMPLEX OMEGA,OMEGK,WYK,C(LS,NS,2), CH(LS,2,NS)
      ANGLE = FLOAT(IS)*4.*ATAN(1.)/FLOAT(LS)
      OMEGA = CMPLX(COS(ANGLE),SIN(ANGLE))
      DO 400 J = 1,NS
      OMEGK = 1.
      DO 400 I = 1,LS
      WYK = OMEGK * C(I,J,2)
      CH(I,1,J) = C(I,J,1) + WYK
      CH(I,2,J) = C(I,J,1) - WYK
      OMEGK = OMEGA * OMEGK
  400 CONTINUE
      RETURN
      END
```

The Stockham algorithm is given in Table 4. If we compare column 2 in Tables 3 and 4, we can observe that in Table 4 both first elements appear, followed by both second elements, and so forth. If we define the elements in a column of Table 3 by $c_{i,j}$, where i ranges over the length of a sequence and j ranges over the number of sequences, then these elements are transposed in Table 4 and stored as $c_{j,i}$. The result is that the ordering is unchanged throughout the order phase. We will make use of this fact in the next section, where multiple and multidimensional transforms are discussed.

A FORTRAN program for the Stockham algorithm is given below. Since the order of the original sequence is unchanged throughout the order phase, only the combine phase is given.

```
      SUBROUTINE STOCK(IS,M,C,WORK)
C
C     THE STOCKHAM AUTOSORT FFT
C
      COMPLEX C(1), WORK(1)
      N = 2**M
```

Table 4
The Stockham FFT for $N = 8$



```
      DO 100 L = 1,M
      LS = 2**(L-1)
      NS = N/(LS+LS)
      CALL STOCK1(IS,LS,NS,C,WORK)
      DO 100 I = 1,N
      C(I) = WORK(I)
  100 CONTINUE
      RETURN
      END
      SUBROUTINE STOCK1(IS,LS,NS,C,CH)
      COMPLEX OMEGA,WYK,C(NS,2,LS), CH(NS,LS,2)
      ANGLE = FLOAT(IS)*4.*ATAN(1.)/FLOAT(LS)
      OMEGA = CMPLX(COS(ANGLE),SIN(ANGLE))
      DO 200 J = 1,NS
      OMEGK = 1.
      DO 200 I = 1,LS
      WYK = OMEGK*C(J,2,I)
      CH(J,I,1) = C(J,1,I) + WYK
      CH(J,I,2) = C(J,1,I) - WYK
      OMEGK = OMEGA*OMEGK
  200 CONTINUE
      RETURN
      END
```

In conclusion, several additional remarks about the Stockham algorithms are presented.

– The shortest vector or inner loop of loop 100 can be lenghtned to $N^{1/2}$ using the method that was described in Section 3.

– The Stockham algorithm cannot be performed in place, and requires an additional complex array *WORK* of length *N*.

– As with the Pease algorithm, the transfer of the array *WORK* to the *C* array at the end of loop 100 can be eliminated by duplicating the CALL STOCK1 statement, but with the arguments *C* and *WORK* interchanged. These two CALL statements are executed alternatively, with a final transfer required after loop 100 only if *M* is an odd integer.

6. Multiple and multi-dimensional transforms

Probably the most satisfactory situation from a vectorizing point of view is the one in which a number of transforms are to be computed. In this situation it is possible to develop methods, some of which are quite straightforward, which yield vectors that are long enough for efficient computation on all architectures. Multiple transforms are required when one uses the Fourier method to solve partial-differential equations [14] where it is particularly important to develop techniques that make the most efficient use of the resource. In what follows, we will present two methods for the efficient computation of multiple transforms. Variants and combinations of these methods are used to compute multi-dimensional transforms.

Method A. Assume that *M* transforms of length *N* are to be computed. Then one can apply each operation in the FFT to all *M* sequences. Thus, any of the FFT programs can be converted to a multiple FFT program by including inner loops, all of length *M*, which apply each operation of

the FFT to all M sequences. The properties of such a multiple-transform program are derived, for the most part, from the properties of the underlying FFT. Unlike Method B given below, Method A becomes less efficient for small M.

Method B. Again we assume that M transforms of length N are to be computed. If we examine columns 2 through 6 of Table 1 in Section 2, we see that these four columns constitute a multiple-transform problem in which $N = 4$ and $M = 2$. If we ignore columns 1 and 6, and assume that the two sequences in column 2 were given, then their transforms are computed in column 6. Similarly, we note that columns 3 through 5 constitute a multiple-transform problem for which $N = 2$ and $M = 4$. Thus, we can compute the transforms of all M sequences by combining them into a single sequence, which is transformed using a 'truncated' version of any one of the FFT algorithms. This method, which was presented in [11], reduces to the FFT algorithm for a single sequence when $M = 1$. Therefore, unlike Method A, it is at least as efficient as the FFT for a single sequence.

From Table 3, Section 5, we see that Method B for multiple transforms can be applied to the variant of the Stockham algorithm, but without the autosort feature. The reason is that although the order of the sequence in column 4 is the same as in column 1, the intermediate orderings in columns 2, 3, and 4 are different. This difficulty is eliminated by using the Stockham algorithm given in Table 4. Since the order is the same throughout the order phase, Method B can be used without an ordering phase. Note, however, that the intermediate sequences and transforms are assumed to be in transposed order, i.e., the first elements in each sequence are grouped together and appear first, followed by all second elements in the sequences, and so forth. This implies that if the sequences are stored column-wise, as they are in a FORTRAN array, then the transforms will be performed row-wise. That is, if we apply the 'truncated' Stockham algorithm to a FORTRAN array $C(I, J)$, where $I = 1, \ldots, N$ and $J = 1, \ldots, M$, then N transforms will be computed, each in the row or J direction.

The intermediate sequences and transforms are stored column-wise in the Cooley–Tukey FFT and the variant of the Stockham algorithm, whereas the intermediate sequences and transforms are stored row-wise in the Pease and Stockham algorithms.

The multi-dimensional transform of an $N \times M$ array consists of N row-wise transforms of length M followed by M column-wise transforms of length N. Of course, the order can be interchanged, i.e., the column-wise transforms can be performed prior to the row-wise transforms. In any event, it would be reasonable to select either the Cooley–Tukey or the Stockham variant for the column-wise transforms and the Pease or Stockham algorithm for the row-wise transforms. The difficulty with this approach is that if the transforms must be ordered, this would offset any savings from not having to transpose the array. The exception is to use the Stockham algorithm with Method B to perform the row-wise transforms and also the Stockham algorithm or its variant combined with Method A to do the column-wise transforms. Indeed, this approach has been suggested by Wang [19]. Korn and Lambiotte [11] examine the Pease and Stockham algorithms for multiple transforms on the CDC STAR-100. Fornberg [8] examines the performance of Glassman's FFT applied to multiple transforms on the same computer.

Up to this point, we have not discussed what perhaps is the key issue associated with multiple or multi-dimensional transforms, namely, minimizing the use of storage by using an in-place transform. The only option is to use Cooley–Tukey algorithm together with Method A. This approach can be used for both column-wise and row-wise transforms. Ordering can be avoided using the method described in Section 3 for problems in which the transform remains invisible. However, if ordering is required, it can be done in place using bit reversal when N is a power of two or using Singleton's method [13] when N is a product of squared primes. However, either method of ordering is expensive on a vector computer.

References

[1] G.D. Bergland, A fast Fourier transform for real-valued series, *Comm. ACM*, **11** (1968) 703–710.

[2] E.O. Brigham, *The Fast Fourier Transform* (Prentice-Hall, Englewood Cliffs, NJ, 1974).

[3] W.T. Cochran, J.W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, G.C. Maling, Jr., D.E. Nelson, C.M. Rader and P.D. Welch, What is the fast Fourier transform? *IEEE Trans. Audio Electroacoustics* **Au-15** (1967) 45–55.

[4] J.W. Cooley and J.W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Math. Comput.* **19** (1965) 297–301.

[5] J.W. Cooley, P.A.W. Lewis and P.D. Welch, Historical notes on the fast Fourier transform, *Proc. IEEE* **55** (1967) 1675–1677.

[6] J.W. Cooley, P.A.W. Lewis and P.D. Welch, The fast Fourier transform algorithm: Programming considerations in the calculation of sine, cosine and Laplace transforms, *J. Sound Vib.* **12** (1970) 315–337.

[7] J. Dollimore, Some algorithms for use with the fast Fourier transform, *J. Inst. Maths. Appl.* **12** (1973) 115–117.

[8] B. Fornberg, A vector implementation of the fast Fourier transform algorithm, *Math. Comput.* **36** (1981) 189–191.

[9] W.M. Gentleman and G. Sande, Fast Fourier transforms for fun and profit, *1966 Fall Joint Computer Conference, AFIPS Proc.* **29** (1966) 563–578.

[10] J.A. Glassman, A generalization of the fast Fourier transform, *IEEE Trans. Comput.* **C-19** (1970) 105–116.

[11] D.G. Korn and J.J. Lambiotte, Jr., Computing the fast Fourier transform on a vector computer, *Math. Comput.* **33** (1979) 977–992.

[12] M.C. Pease, An adaptation of the fast Fourier transform for parallel processing, *J. ACM* **15** (1968) 252–264.

[13] R.C. Singleton, An algorithm for computing the mixed radix fast Fourier transform, *IEEE Trans. Audio Electroacoustics* **Au-17** (1969) 93–103.

[14] P.N. Swarztrauber, The methods of cyclic reduction, Fourier analysis and the FACR algorithm for the discrete solution of Poisson's equation on a rectangle, *SIAM Rev.* **19** (1977) 490–501.

[15] P.N. Swarztrauber, Vectorizing the FFT's, in: G. Rodrigue, Ed., *Parallel Computations* (Academic Press, New York, 1982) 490–501.

[16] C. Temperton, A note on prime factor FFT algorithm, *J. Comput. Phys.* **52** (1983) 198–204.

[17] C. Temperton, Fast mixed-radix real Fourier transforms, *J. Comput. Phys.* **52** (1983) 340–350.

[18] C. Temperton, Self-sorting mixed-radix fast Fourier transforms, *J. Comput. Phys.* **52** (1983) 1–23.

[19] H.H. Wang, On vectorizing the fast Fourier transform, *BIT* **20** (1980) 233–243.

[20] S. Winograd, On computing the discrete Fourier transform, *Math. Comput.* **32** (1978) 175–199.