TECHNICAL MEMORANDUM

66

Self sorting mixed radix fast Fourier transforms

C. Temperton

Research Department

October 1982

European Centre for Medium-Range Weather Forecasts
Europäisches Zentrum für mittelfristige Wettervorhersage
Centre européen pour les prévisions météorologiques à moyen

Abstract

Some examples are given of the uses of the FFT algorithm in numerical weather prediction. The algorithm is derived by means of matrix factorization, in such a way that the many possible variants can be obtained in a unified manner. Particular attention is paid to self-sorting variants of the algorithm, but skeleton Fortran routines are included for all the "canonical" forms.

The FFT works by decomposing the discrete Fourier transform into a sequence of short transforms. Algorithms for these short transforms are presented, and used to derive operation counts for the whole algorithm.

The implementation of the FFT on vector computers is described, and in the final section it is demonstrated how savings can be achieved in the case of two-dimensional transforms.

1. INTRODUCTION

Much of the progress made in computational physics during the past fifteen years would not have been possible without the Fast Fourier Transform (FFT) algorithm. The examples quoted here are taken from the field of numerical weather prediction, but readers working in other areas will be able to supply similar examples.

At the European Centre for Medium Range Weather Forecasts (ECMWF), a 10-day numerical weather forecast is run each day using a three-dimensional finite-difference model of the global atmosphere with 15 levels in the vertical and a regular 96 x 192 latitude-longitude grid in the horizontal. The computational task of integrating such a large model is made feasible only by using special methods to lengthen the maximum stable timestep. This is achieved in two ways, in both of which the FFT plays an important role.

Firstly, the maximum stable timestep is proportional to the physical distance between gridpoints; the convergence of the meridians implies that the gridpoints become very close together in the zonal direction as the grid approaches the poles. One remedy, now almost conventional in NWP models based on a regular latitude-longitude grid, is to filter out the small-scale Fourier components of the model variables (or their time-tendencies) near the poles, thus increasing the effective zonal separation of the gridpoints without sacrificing the regularity of the grid [14, 28]. This filtering is most efficiently achieved by Fourier transforming the fields, setting the high-wavenumber coefficients to zero, and then transforming back to physical space. An alternative technique, currently implemented in the ECMWF gridpoint model, is to include implicit latitude-dependent zonal diffusion terms. This technique too is most efficiently implemented via the FFT.

Secondly, the maximum timestep is inversely proportional to the frequency of the most rapidly oscillating solutions of the model equations. These are the inertia-gravity waves which have only small amplitudes in the real atmosphere and are of little significance for NWP. By treating certain linear terms in the model using a Crank-Nicholson time-integration procedure rather than the usual explicit centred (leapfrog) scheme, the troublesome small-scale high-frequency gravity waves can be artificially slowed down. The resulting semi-implicit time integration algorithm [19] remains stable with a timestep typically 6 times longer than would be possible with an explicit scheme, while yielding practically identical results. Implementation of the semi-implicit algorithm in a multi-level model requires the solution at each timestep of a three-dimensional discrete elliptic equation. Diagonalizing the vertical part of the finite-difference operator decouples this equation into a set of two-dimensional Helmholtz equations, which are solved by combining a fast direct solver in spherical geometry [23] with the technique described in [4] for large out-of-core problems. Most of the work in the fast two-dimensional elliptic solver consists of Fourier transforms between gridpoint and wavenumber space.

With the model resolution currently employed, a 10-day forecast requires approximately 15 million real Fourier transforms, each of length N=192.

Even greater use of the FFT is made by the spectral model developed as a possible alternative to the gridpoint model. Here the horizontal fields are represented not by values at an array of gridpoints but by coefficients of spherical harmonic basis functions. Spectral models are free of the pole problem, and the semi-implicit algorithm is easily implemented since spherical harmonics are eigenfunctions of the Helmholtz operator. However, the model's nonlinear terms are most efficiently computed by first transforming to physical space, evaluating the terms on a suitable grid, and then transforming back to spectral space [3, 17]. The zonal part of the two-dimensional transforms between gridpoint and spectral space again consists of Fourier transforms. At the current quasi-operational resolution of the ECMWF spectralmodel, a 10-day forecast requires about 24 million real Fourier transforms of length N=192.

In addition to its many applications in computational physics and related fields such as signal processing, the FFT has found a number of uses in computational complex analysis, as recently reviewed by Henrici [13].

While the general principle of how the FFT works is reasonably straightforward and quite well-known, the precise details of the algorithm (or family of algorithms, for there are many variants) are somewhat involved, and thanks to the widespread availability of "black box" routines many users have been able to avoid coming to terms with them. This is unfortunate for several reasons. First, the black boxes are not always as efficient or as general as they might be, and myths persist in some quarters that, for example, the algorithm is only efficient if N is a power of 2, or that a permutation of the data is necessary either before or after the transform. Second, scientists faced with transferring their calculations to new computer architectures need to understand the algorithm in order to exploit the transfer to the full. A third, less utilitarian, reason is simply that the FFT is a very interesting algorithm.

Many papers have been written explaining the details of the algorithm, and the only excuse for adding to the total is that the author believes the approach presented here is more general and flexible than most. Also, routines based on the ideas discussed here (particularly as implemented on the vector machines Cray-1 and Cyber 205) have attracted some interest in the numerical weather prediction community, and these ideas may have wider application.

This paper will consider the Fast Fourier Transform of purely complex data; the specialization to the real case will be left for a later publication. The FFT will be of the conventional type in which no restrictions are placed on the factors of N, so we will not treat the special case of mutually prime factors due to Good [12] and developed more recently by Winograd [29]. Attention will be focussed in particular on "self-sorting" variants of the algorithm which do not require any scrambling or unscrambling of the data before or after the transform.

2. DEVELOPMENT OF THE ALGORITHM

(a) General remarks

There are basically two approaches to the problem of explaining the FFT algorithm. The first takes the defining equation:

$$x_j = \sum_{k=0}^{N-1} z_k \exp(2ijk\pi/N), \quad 0 \leqslant j \leqslant N-1 \tag{1}$$

and proceeds rather directly by manipulation of the indices j and k. Examples of this approach are the papers by Cooley and Tukey [6], Gentleman and Sande [9], and the book by Brigham [2]. If N is the product of two factors, this approach allows a rather simple explanation of what is going on. However, the extension to many factors requires a forest of summation signs and multiple subscripts which obscures the details and hides the many alternatives which are available.

The alternative is to write Eq.(1) in matrix form:

$$\underset{\sim}{x} = W_N \underset{\sim}{z} \tag{2}$$

and to demonstrate that the matrix W_N can be factorized as a product of sparse matrices [8, 10, 15, 18, 26]. We refer to W_N as the DFT (Discrete Fourier Transform) matrix of order N. This approach has a more abstract flavour, and can become somewhat unwieldy. However, it is possible to develop a compact notation which enables us to treat many different variants of the FFT algorithm in a unified manner. Strang [22, p.303] has made the nice observation that "certain special matrices will fall apart when they are dropped the right way". We shall see that, in the case of the DFT matrix, the pieces can be reassembled in many different ways.

In the development which follows, element (j,k) of a matrix A will be denoted by $[A](j,k)$, with the convention that the rows and columns of an N x N matrix are indexed from 0 to N-1. For example, from Eqs.(1) and (2) we have

$$[W_N](j,k) = \omega^{jk} \tag{3}$$

where $\omega = \exp(2i\pi/N)$. Our aim will be to show that if

$$N = n_1 n_2 \dots n_{k-1} n_k$$

then

$$W_N = T_k T_{k-1} \dots T_2 T_1 \tag{4}$$

and to determine explicitly the form of the factors T_i.

(b) The two-factor case

Let N=pq. Then for any $0 \leqslant j \leqslant N-1$, we can write $j = \ell p + m$, where $0 \leqslant \ell \leqslant q-1$, $0 \leqslant m \leqslant p-1$. Similarly for any $0 \leqslant k \leqslant N-1$, we can write $k = rq + s$, where $0 \leqslant r \leqslant p-1$, $0 \leqslant s \leqslant q-1$.

Let I_r be the identity matrix of order r. Then for $0 \leqslant p \leqslant r-1$, $0 \leqslant q \leqslant r-1$,

$$[I_r](p,q) = \delta_{pq} \tag{5}$$

where δ is the Kronecker delta.

Following Bellman [1], define a Kronecker matrix product by $A \times B = (a_{ij}B)$. In the present notation, let A and B be square matrices, of order p and q respectively. Then

$$[A \times B](tq + u, rq + s) = [A](t,r).[B](u,s). \tag{6}$$

Define a permutation matrix P^p_q of order pq by

$$\begin{aligned}[P^p_q](j,k) &= 1 \text{ if } j = rp + s \text{ and } k = sq + r \\ &= 0 \text{ otherwise.}\end{aligned}$$

Finally, define a diagonal matrix D_q^p of order pq by

$$[D_q^p](j,k) = \omega^{sm} \text{ if } j = k = sq + m$$

$$= 0 \text{ otherwise}$$

where $\omega = \exp(2i\pi/pq)$.

We can now establish the following matrix factorization theorem:

$$W_{pq} = (W_q \times I_p)\, P_q^p\, D_q^p\, (W_p \times I_q) \tag{7}$$

Proof. From Eqs. (3), (5) and (6),

$$[W_p \times I_q]\,(tq + u,\ rq + s) = \omega^{qrt}\, \delta_{su} \tag{8}$$

(Here $\omega^q = \exp(2i\pi/p)$).

Premultiplication by a diagonal matrix multiplies the rows, so from Eq. (8) and the definition of D_q^p we have:

$$[D_q^p\,(W_p \times I_q)]\,(tq + u,\ rq + s) = \omega^{tu}\, \omega^{qrt}\, \delta_{su}\,.$$

Premultiplication by the permutation matrix P_q^p takes row $(tq + u)$ to row $(up + t)$, hence

$$[P_q^p\, D_q^p\,(W_p \times I_q)]\,(up + t,\ rq + s) = \omega^{tu}\, \omega^{qrt}\, \delta_{su}. \tag{9}$$

Now, we also have from Eqs. (3), (5) and (6):

$$[W_q \times I_p]\,(\ell p + m,\ up + t) = \omega^{\ell pu}\, \delta_{mt}\,. \tag{10}$$

By the usual rules of matrix multiplication, Eqs. (9) and (10) give:

$$[(W_q \times I_p)\, P_q^p\, D_q^p\,(W_p \times I_q)]\,(\ell p + m,\ rq + s)$$

$$= \sum_{t=0}^{p-1} \sum_{u=0}^{q-1} \omega^{\ell pu}\, \omega^{tu}\, \omega^{qrt}\, \delta_{mt}\, \delta_{su}\,.$$

The presence of the factors δ_{mt} and δ_{su} ensures that there is only one non-zero contribution to this sum, namely:

$$\omega^{\ell ps}\ \omega^{ms}\ \omega^{mqr}$$

$$= \omega^{\ell ps}\ \omega^{ms}\ \omega^{mqr}\ \omega^{\ell pqr} \quad \text{since } \omega^{pq} = 1$$

$$= \omega^{(\ell p+m)(rq + s)}$$

Thus element $(j,k) = (\ell p + m,\ rq + s)$ of the product matrix is ω^{jk}, i.e. it is equal to element (j,k) of W_{pq}, and the theorem is proved.

In keeping with Eq. (4), we write Eq. (7) in the form

$$W_{pq} = T_2\ T_1\ .$$

The generalization to more factors depends on how this is done. For the time being, we write

$$T_1 = P^p_q\ D^p_q\ (W_p \times I_q)$$

$$T_2 = (W_q \times I_p)$$

and note that since $P^q_1 = D^q_1 = I_q$, we could write:

$$T_2 = (P^q_1\ D^q_1 \times I_p)(W_q \times I_p)\ .$$

This arrangement leads to a version of the algorithm known as "decimation in frequency" [11].

(c) Extension to more factors

The extension to multiple factors is now simply a matter of matrix algebra. For example, if N = pqr, we have from Eq. (7):

$$W_{pqr} = W_{p(qr)} = (W_{qr} \times I_p)\ P^p_{qr}\ D^p_{qr}\ (W_p \times I_{qr})$$

and using Eq. (7) again to expand W_{qr} ,

$$W_{pqr} = (W_r \times I_{pq})(P_r^q D_r^q \times I_p)(W_q \times I_{pr}) P_{qr}^p D_{qr}^p (W_p \times I_{qr}) . \qquad (11)$$

Writing this in the form

$$W_{pqr} = T_3 T_2 T_1$$

we have

$$T_1 = P_{qr}^p D_{qr}^p (W_p \times I_{qr})$$

$$T_2 = (P_r^q D_r^q \times I_p)(W_q \times I_{pr})$$

$$T_3 = (W_r \times I_{pq})$$

and we could write

$$T_3 = (P_1^r D_1^r \times I_{pq})(W_r \times I_{pq}) .$$

The pattern is now beginning to emerge. Suppose $N = n_1 n_2 \ldots . n_k$. Define $\ell_1 = 1$, $\ell_{i+1} = n_i \ell_i$ for $1 \leqslant i \leqslant k$, and $m_i = N/\ell_{i+1}$.
Then the general form is given by Eq. (4), with

$$T_1 = (P_{m_i}^{n_i} D_{m_i}^{n_i} \times I_{\ell_i})(W_{n_i} \times I_{N/n_i}) \qquad (12)$$

Eq. (12) may be proved formally by induction. Each stage of the transform algorithm, corresponding to a factor n of N, consists of a set of transforms of order n, then a multiplication by a diagonal matrix, and finally a permutation of the results.

The above development may be mathematically convincing, but it does not convey immediately what the corresponding computer program would look like; most of us think more easily in Fortran. The virtue of the matrix formalism is that, having developed one form of the FFT algorithm, many other variants can easily be deduced.

A skeleton Fortran program to implement Eqs. (4) and (12) will now be presented. Suppose that the factors of N have been stored in an array IFAX(1) to IFAX (NFAX), and that a complex array of trigonometric function values has been defined by

$$\text{TRIGS}\ (K + 1) = \exp(2iK\pi/N), \quad 0 \leqslant K \leqslant N-1 \ .$$

Suppose that the data to be transformed is in an array A, and that a work array C is provided. Each array acts alternately as input and output for successive stages of the algorithm. The outer structure of the FFT routine is then:

```
C      DECIMATION IN FREQUENCY
       COMPLEX A(N), C(N), TRIGS(N)
       INTEGER IFAX (NFAX)
       LA = 1
       DO 1Ø I = 1, NFAX
       IFAC = IFAX(I)
       CALL PASS (A,C,TRIGS,IFAC,LA,N)
C      [now reverse roles of A and C]
       LA = LA * IFAC
    1Ø CONTINUE
       STOP
       END
```

The subroutine PASS has the following structure:

```
C      SELF-SORTING, DECIMATION IN FREQUENCY
       SUBROUTINE PASS(A,C,TRIGS,IFAC,LA,N)
       COMPLEX A(N),C(N), TRIGS(N)
       M = N/IFAC
C      Define IFAC base addresses in A
       IA=Ø, IB=M, IC=2*M, ID=3*M,...

C      Define IFAC base addresses in C
       JA=Ø, JB=LA, JC=2*LA, JD=3*LA,...

C      ......
```

```
      I=1
      J=1
      JUMP = (IFAC-1)*LA
      DO 2Ø K=Ø, M-LA, LA
      DO 1Ø L=1, LA
      C(J) = Ω(K)*(W(IFAC)*A(I))
      I=I+1
      J=J+1
   1Ø CONTINUE
      J=J+JUMP
   2Ø CONTINUE
      RETURN
      END
```

(In the statement $C(J) = \Omega(K)*(W(IFAC)*A(I))$, $\underset{\sim}{C}$ and $\underset{\sim}{A}$ are vectors.)

In the inner loop, W(IFAC) is the DFT matrix of order IFAC, Ω(K) is a diagonal matrix defined by

$$\Omega(K) = \text{diag}(\text{TRIGS}(1),\ \text{TRIGS}(K+1),\ \text{TRIGS}(2*K+1),\ldots),$$

$\underset{\sim}{A}(I)$ and $\underset{\sim}{C}(J)$ are vectors of length IFAC defined by

$$\underset{\sim}{A}(I) = \begin{bmatrix} A(IA+I) \\ A(IB+I) \\ A(IC+I) \\ \vdots \end{bmatrix}, \quad \underset{\sim}{C}(J) = \begin{bmatrix} C(JA+J) \\ C(JB+J) \\ C(JC+J) \\ \vdots \end{bmatrix}.$$

In practice the inner loop would of course be written out as a sequence of scalar operations, preferably in real arithmetic, using various tricks to simplify the multiplication by W(IFAC) (see Section 3), and taking account of the fact that the first element of Ω(K) is always 1. For example, if IFAC=2 the inner loop (in complex arithmetic) becomes simply:

```
      DP 1Ø L=1, LA
      C(JA+J) = A(IA+I) + A(IB+I)
      C(JB+J) = TRIGS(K+1)*(A(IA+I) - A(IB+I))
      I=I+1
      J=J+1
   1Ø CONTINUE
```

In many circumstances it is worth providing a special case for K=0, since $\Omega(0)$ is just the identity matrix.

Notice that the final results of the transform emerge in their natural order; the required internal permutations are implemented simply by the indexing scheme presented above. This is the essence of the "self-sorting" FFT, which distinguishes it from other variants in which a separate data permutation is required either before or after the transform. The only penalty is that two arrays are needed, since in general the computations cannot be performed in place.

(d) Alternative self-sorting algorithm

Let us return to Eq. (7), and split the two-factor case in a different way:

$$T_1 = (W_p \times I_q)$$

$$T_2 = (W_q \times I_p)\, P_q^p\, D_q^p$$

and notice that we could have written:

$$T_1 = (W_p \times I_q)(P_p^1\, D_p^1 \times I_q)\ .$$

This rearrangement gives rise to the form of the algorithm known as "decimation in time". From Eq. (7) we have:

$$W_{pqr} = W_{(pq)r} = (W_r \times I_{pq})\, P_r^{pq}\, D_r^{pq}\, (W_{pq} \times I_r)$$

and using Eq. (7) again to expand W_{pq},

$$W_{pqr} = (W_r \times I_{pq})\, P_r^{pq}\, D_r^{pq} (W_q \times I_{pr})(P_q^p\, D_q^p \times I_r)(W_p \times I_{qr}) \qquad (13)$$

which we can write as the product of three factors:

$$T_1 = (W_p \times I_{qr})(P_p^1\, D_p^1 \times I_{qr})$$

$$T_2 = (W_q \times I_{pr})(P_q^p\, D_q^p \times I_r)$$

$$T_3 = (W_r \times I_{pq})\, P_r^{pq}\, D_r^{pq}\ .$$

Again a pattern emerges, but this time the pattern is somewhat different. As before let $N = n_1 n_2 \ldots n_k$, $\ell_1 = 1$, $\ell_{i+1} = n_i \ell_i$ for $1 \leqslant i \leqslant k$, and $m_i = N/\ell_{i+1}$. Then the general form is given by Eq. (4), with

$$T_i = (W_{n_i} \times I_{N/n_i})(P_{n_i}^{\ell_i} D_{n_i}^{\ell_i} \times I_{m_i}) . \tag{14}$$

Again, Eq. (14) can be proved formally by induction. Notice also that Eq. (13) for three factors may be derived from Eq. (11) by a different route. Since W_{pqr} is symmetric we can transpose the factorization of Eq. (11); using the identities $(AB)^T = B^T A^T$, $(A \times B)^T = A^T \times B^T$, $(P_q^p)^T = P_p^q$, and the fact that the W, I and D matrices are themselves symmetric, and finally interchanging p and r, gives Eq. (13). In fact the same process can be applied to the general case, Eq. (12), to give Eq. (14).

This version of the algorithm gives rise to a program similar to that already presented, except that the variable LA now decreases on successive passes, the indexing of the input and output arrays within the subroutine PASS is interchanged, and the multiplication by Ω(K) is done before the multiplication by W(IFAC). For completeness we give the corresponding routine below:

```
C      DECIMATION IN TIME
       COMPLEX A(N), C(N), TRIGS(N)
       INTEGER IFAX(NFAX)
       LA=N
       DO 1Ø I=1,NFAX
       IFAC=IFAX(I)
       LA=LA/IFAC
       CALL PASS(A,C,TRIGS,IFAC,LA,N)
C      [now reverse roles of A and C ]
    1Ø CONTINUE
       STOP
       END
```

Subroutine PASS is the same as before, except that the inner loop is replaced by:

$$\underset{\sim}{C}(I) = W(IFAC)*(\Omega(K)*\underset{\sim}{A}(J)).$$

In this self-sorting variant of the FFT, the intermediate results can be characterized in a manner which is reminiscent of the way the algorithm is often derived in the field of signal processing [11]. For example in the three-factor case N=pqr, the first stage consists of DFT's of length p on qr interleaved 'samples' of the data; in the second stage we combine these into DFT's of length pq on r samples of the data, and in the final stage these are combined into a single DFT of length pqr. (Hence the name "decimation in time").

There is no such simple interpretation of the intermediate results for the first self-sorting variant (decimation in frequency). However we can make a "dual" observation, namely that the last stage is equivalent to pq interleaved transforms of length r; the second and third stages together are equivalent to p interleaved transforms of length qr; and of course the three stages together constitute one transform of length pqr. This observation is important for the vectorization techniques discussed in Section 4.

(e) Other forms

We now develop a form of the FFT algorithm in which all the permutations are delayed until the end. We show that if

$$N = n_1 n_2 \dots n_{k-1} n_k$$

then

$$W_N = P_1 P_2 \dots P_{k-1} P_k T_k T_{k-1} \dots T_2 T_1 \tag{15}$$

First note the following useful lemma. If A_p, B_q are any square matrices of order p and q respectively, then

$$P_q^p (A_p \times B_q) = (B_q \times A_p) P_q^p. \tag{16}$$

Using Eq. (16), Eq. (7) can be rewritten as

$$W_{pq} = P_q^p (I_p \times W_q) D_q^p (W_p \times I_q) \tag{17}$$

We can now write

$$W_{pq} = P_1\, P_2\, T_2\, T_1$$

with

$$T_1 = D^p_q\, (W_p \times I_q)$$

$$T_2 = (I_p \times D^q_1)\,(I_p \times W_q)$$

$$P_1 = P^p_q$$

$$P_2 = I_p \times P^q_1 \quad (= I_{pq}) \ .$$

For N=pqr, we have

$$W_{p(qr)} = P^p_{qr}\, (I_p \times W_{qr})\, D^p_{qr}\, (W_p \times I_{qr})$$

and using Eq. (17) again to expand W_{qr} ,

$$W_{pqr} = P^p_{qr}\, (I_p \times P^q_r)\,(I_{pq} \times W_r)\,(I_p \times D^q_r)\,(I_p \times W_q \times I_r)\,D^p_{qr}\, (W_p \times I_{qr})$$

which can be written as

$$W_{pqr} = P_1\, P_2\, P_3\, T_3\, T_2\, T_1$$

with

$$T_1 = D^p_{qr}\, (W_p \times I_{qr})$$

$$T_2 = (I_p \times D^q_r)\; (I_p \times W_q \times I_r)$$

$$T_3 = (I_{pq} \times D^r_1)\; (I_{pq} \times W_r)$$

$$P_1 = P^p_{qr}$$

$$P_2 = I_p \times P^q_r$$

$$P_3 = I_{pq} \times P^r_1 \quad (=I_N)$$

As before, define $\ell_1 = 1$, $\ell_{i+1} = n_i \ell_i$ for $1 \leqslant i \leqslant k$, and $m_i = N/\ell_{i+1}$. Then the general case is given by Eq. (15) with

$$T_i = (I_{\ell_i} \times D_{m_i}^{n_i})(I_{\ell_i} \times W_{n_i} \times I_{m_i}) \tag{18}$$

$$P_i = (I_{\ell_i} \times P_{m_i}^{n_i}) \ .$$

This form corresponds to the Gentleman-Sande [9] version of the FFT algorithm, and is another form of decimation in frequency. The multiplications by T_i can be done in place, and in fact so can the permutations P_i, though the logic may then become rather complicated; see Singleton [20] for details. This form of the FFT has the advantage (rarely important on modern computers) that only a single array is required, but at the cost of having to perform an explicit unscrambling of the data at the end of the algorithm.

For a Fortran implementation we use the same routine as for self-sorting decimation in frequency, but within the subroutine PASS the inner loop is replaced by

$$\underset{\sim}{C}(J) = \Omega(K)*(W(IFAC)*\underset{\sim}{A}(J)).$$

The index I and the corresponding base addresses are no longer required, and the array $\underset{\sim}{C}$ can occupy the same locations as the array A.

The Cooley-Tukey [6] version of decimation in time is obtained by performing the permutations before the remainder of the algorithm. Again using Eq. (17) and the fact that $P_q^p D_q^p = D_p^q P_q^p$, Eq. (7) becomes

$$W_{pq} = (W_q \times I_p) D_p^q (I_q \times W_p) P_q^p \ .$$

The generalization to three factors is analogous to the Gentleman-Sande case, but using $W_{pqr} = W_{(pq)r}$. Skipping the details, we obtain (for k factors):

$$W_N = T_k T_{k-1} \cdots T_2 T_1 P_1 P_2 \cdots P_{k-1} P_k \tag{19}$$

with

$$T_i = (I_{m_i} \times W_{n_i} \times I_{\ell_i})(I_{m_i} \times D_{\ell_i}^{n_i})$$

$$P_i = (I_{m_i} \times P_{n_i}^{\ell_i}) .$$

As in the case of the two self-sorting algorithms, the Cooley-Tukey version can be obtained simply by transposing the Gentleman-Sande factorization. The appropriate Fortran routine is now that for decimation in time, but the inner loop of subroutine PASS becomes

$$\underset{\sim}{C}(J) = W(IFAC)*(\Omega(K)*\underset{\sim}{A}(J)).$$

Yet another distinct form of the algorithm using decimation in frequency is that due to Pease [18], which can be derived from the Gentleman-Sande form. In order to do so we need another useful identity:

$$P_r^{pq} \, P_p^{qr} = P_{pr}^{q} , \tag{20}$$

and a corollary of Eq. (16):

$$(A_p \times B_q) = P_p^q \, (B_q \times A_p) \, P_q^p . \tag{21}$$

Consider the factors T_i of the Gentleman-Sande algorithm, given by Eq. (18). Using Eqn. (21) we can rewrite each of them as

$$T_i = P_{\ell_i}^{m_i n_i} \, (D_{m_i}^{n_i} \times I_{\ell_i})(W_{n_i} \times I_{\ell_i m_i}) \, P_{m_i n_i}^{\ell_i}$$

At the left-hand end of the sequence $T_k \, T_{k-1} \cdots T_2 \, T_1$ we now get

$$P_{\ell_k}^{m_k n_k} = P_{N/n_k}^{n_k}$$

while at the right-hand end we get

$$P_N^1 = I_N$$

Within the sequence we get pairs of permutation matrices of the form

$$P_{m_{i+1} \, n_{i+1}}^{\ell_{i+1}} \qquad P_{\ell_i}^{m_i n_i}$$

Since $\ell_{i+1} = \ell_i n_i$ and $m_{i+1}\, n_{i+1} = m_i$, we can write this as

$$P_{m_i}^{\ell_i n_i} P_{\ell_i}^{m_i n_i} = P_{\ell_i m_i}^{n_i} = P_{N/n_i}^{n_i}$$

using Eq. (20).

We now have a new factorization which is still of the form (15), but with

$$T_i = P_{N/n_i}^{n_i} (D_{m_i}^{n_i} \times I_{\ell_i})(W_{n_i} \times I_{N/n_i}).$$

The advantage of this form (especially for the case $n_i = 2$ for all i, with which Pease was primarily concerned) is that the structure remains the same for each pass; only the diagonal matrix $(D_{m_i}^{n_i} \times I_{\ell_i})$ changes from one pass to the next. This form is therefore rather suitable for hardware implementations, although an explicit permutation of the data is still required to complete the transform (in fact the same permutation as in the Gentleman-Sande algorithm). Incidentally, readers who refer to Pease's paper [18] should note that his definition of the Kronecker product is different from that used here.

The appropriate Fortran routine is that for self-sorting decimation in frequency, except that subroutine PASS is always called with LA=1; the nested loop structure then collapses to a single loop with K as the control variable. The two statements incrementing J can then be combined to give J = J+IFAC. However, the indexing of the TRIGS array is now correct only for the first pass. There are two ways of retrieving the situation. We can provide a different TRIGS array for each pass; this is expensive in storage, but to anticipate the results of Section 4 it may be the best solution on a vector computer. Alternatively we can retain a nested loop structure in order to index correctly a single TRIGS array.

As in previous cases there is an analogous form of decimation in time, obtained either directly by transposition or by manipulation of the Cooley-Tukey version along the same lines as above. The outcome is a factorization of the form given by Eq. (19) but with

$$T_i = (W_{n_i} \times I_{N/n_i})(D_{\ell_i}^{n_i} \times I_{m_i})\, P_{n_i}^{N/n_i} \quad .$$

This form was recommended by Singleton [21] for transforming sequences of data too large to be contained in core. The appropriate Fortran routine is now that for self-sorting decimation in time with LA=1, and with modifications similar to those given for the previous algorithm. Note that Pease's algorithm and its transpose, like the self-sorting variants, require a work array.

We now have six different 'canonical' forms of the FFT algorithm, with the structure of the factorization defined in each case. For multiple factors there is now almost no limit to the number of different algorithms which could be invented. For example, if N=pqrs we could use any of the six forms to decompose the transforms into factors containing W_{pq} and W_{rs}, and then use any of the six forms to decompose each of these factors further.

(f) Remarks

(1) It could be argued that there are really only two canonical forms, decimation in frequency and decimation in time; within each of these categories, the intermediate results for any of the three forms are the same apart from a permutation.

(2) All forms require exactly the same amount of floating-point arithmetic.

(3) In the original paper by Cooley and Tukey [6] and many subsequent descriptions, the vector $\underset{\sim}{A}$(I) in the skeleton Fortran routine for decimation in frequency is multiplied by the product matrix Ω(K)*W(IFAC), i.e. we could compute

$$(\Omega(K)*W(IFAC))*\underset{\sim}{A}(I)$$

rather than

$$\Omega(K)*(W(IFAC)*\underset{\sim}{A}(I)) ,$$

and similarly in the routine for decimation in time.

Assuming that the elements of the product matrix are already available (they could be picked out of the TRIGS array), it appears at first sight that this would decrease the operation count. We shall see in the next section why this is not in fact the case. The elements of $\Omega(K)$ are variously referred to as phase factors, rotation factors [20] or twiddle factors [9].

(4) Although neglected for a long time, the self-sorting variants were discovered soon after the Cooley-Tukey version; Cochran et al [5] attribute the idea to Stockham. In a sense it is obvious that such variants should exist; the problem is to find simple algorithms for them. Gentleman and Sande [9] refer to the idea as "FFT's using scratch storage". Uhrich [27] presented a compact self-sorting radix-2 program, but with no explanation of how it worked. Glassman [10] and Drubin [8] also developed a self-sorting algorithm via matrix factorization. For a recent derivation using a somewhat different approach, see De Boor [7].

(5) The self-sorting variants have recently attracted more attention in the context of implementation on vector computers, since the explicit permutation required by the other variants is essentially a scalar operation and thus relatively time-consuming.

(6) There is one situation in which the extra storage requirements of the self-sorting forms and the permutation step of other variants can both be avoided. Suppose for example that the computation consists of a transform from physical space to Fourier space, an operation on the Fourier coefficients, and a transfer back to physical space. For the first transform we can use decimation in frequency, but omitting the permutation. The operation on the Fourier coefficients can then be carried out, using the coefficients in scrambled order. For the final transform we use decimation in time, omitting the permutation step and <u>using the factors of N in reverse order</u>. This procedure depends on the fact that the permutation sequences of Eqs. (15) and (19) are then inverses of each other. The penalty now of course is the need to operate on the scrambled Fourier coefficients; whether this is acceptable will depend on the application. There is also a need for routines to implement decimation in both frequency and time.

(7) Everything in this paper carries through with minor modifications for the inverse transform,

$$Nz_k = \sum_{j=0}^{N-1} x_j \exp(-2ijk\pi/N)$$

or

$$N\underset{\sim}{z} = W_N^* \underset{\sim}{x} \ .$$

Some writers define the forward and inverse transforms the other way round.

3. OPERATION COUNTS

Two aspects of the mixed-radix FFT algorithm for composite N contribute to its speed. The first was demonstrated in the previous Section; the transform can be factorized into a sequence of operations based on transforms of length n_i, where n_i are the factors of N. The second, less often mentioned, is the speed with which these short transforms can themselves be computed. For small n, $\underset{\sim}{x} = W_n\underset{\sim}{z}$ can be obtained using many fewer operations than appears necessary at first sight. (This is of course the reason for separating out the multiplications by phase factors in the algorithms of Section 2).

In this Section we give the algorithms for computing $\underset{\sim}{x} = W_n\underset{\sim}{z}$ for $2 \leqslant n \leqslant 6$, as used in the Cray-1 and Cyber 205 FFT packages mentioned in the Introduction. Note that for n = 3,5,6 they are different from the algorithms given by Winograd [29]. They are slightly more robust with respect to rounding errors, and the algorithm given here for n=5 requires fewer additions (but more multiplications) than Winograd's. The algorithms are all expressed in terms of complex numbers, but it will be seen that all multiplications are by real numbers or by i.

For n=2 we have simply

$$x_o = z_o + z_1 \ ; \quad x_1 = z_o - z_1$$

requiring 4 real additions.

For n=3 we form three intermediate results:

$$t_1 = z_1 + z_2 ; \quad t_2 = z_o - \tfrac{1}{2} t_1 ; \quad t_3 = \sin60*(z_1 - z_2) ;$$

$$x_o = z_o + t_1 ; \quad x_1 = t_2 + it_3 ; \quad x_2 = t_2 - it_3$$

requiring 12 real additions, 4 real multiplications.

For n=4:

$$t_1 = z_o + z_2 ; \quad t_2 = z_1 + z_3 ; \quad t_3 = z_o - z_2 ; \quad t_4 = z_1 - z_3 ;$$

$$x_o = t_1 + t_2 ; \quad x_1 = t_3 + it_4 ; \quad x_2 = t_1 - t_2 ; \quad x_3 = t_3 - it_4$$

requiring 16 real additions.

Singleton [20] gives an algorithm for arbitrary odd n which requires $(n-1)(n+3)$ real additions and $(n-1)^2$ real multiplications. For n=5 this would give 32 real additions and 16 real multiplications. However, in the same paper a modification due to Rader is given for n=5 which saves 4 multiplications. (Winograd's algorithm saves a further two multiplications at the expense of 2 extra additions).

$$t_1 = z_1 + z_4 ; \quad t_2 = z_2 + z_3 ; \quad t_3 = z_1 - z_4 ; \quad t_4 = z_2 - z_3 ; \quad t_5 = t_1 + t_2 ;$$

$$t_6 = (\sqrt{5}/4)(t_1 - t_2) ; \quad t_7 = z_o - \frac{1}{4} t_5 ; \quad t_8 = t_7 + t_6 : t_9 = t_7 - t_6 ;$$

$$t_{10} = \sin72*t_3 + \sin36*t_4 ; \quad t_{11} = \sin36*t_3 - \sin72*t_4 ;$$

$$x_o = z_o + t_5 ; \quad x_1 = t_8 + it_{10} ; x_2 = t_9 + it_{11} ; x_3 = t_9 - it_{11} ; x_4 = t_8 - it_{10}$$

requiring 32 real additions and 12 real multiplications.

A useful variation, which reduces all operations to triadic form, is

$$t_3 = \sin72* (z_1 - z_4) ; \quad t_4 = \sin72* (z_2 - z_3) ;$$

$$t_{10} = t_3 + (\sin36/\sin72)* t_4 ; \quad t_{11} = (\sin36/\sin72)*t_3 - t_4$$

with no change in the operation count.

The inclusion of n=6 in an FFT package is unconventional, but as will be shown later it leads to a significant decrease in the overall operation count. The algorithm is the smallest example of Good's theorem [12] that, by suitable permutations of the input and output data, the phase factors in the factorization of W_{pq} can be eliminated if p and q are mutually prime (here p=2, q=3).

We can rewrite $\underset{\sim}{x} = W_6\underset{\sim}{z}$ as:

$$\begin{bmatrix} x_0 \\ x_4 \\ x_2 \\ \hdashline x_3 \\ x_1 \\ x_5 \end{bmatrix} = \left[\begin{array}{c:c} W_3 & W_3 \\ \hdashline W_3 & -W_3 \end{array}\right] \begin{bmatrix} z_0 \\ z_2 \\ z_4 \\ \hdashline z_3 \\ z_5 \\ z_1 \end{bmatrix}$$

Thus we can use the algorithm for n=3 on the vectors (z_0, z_2, z_4) and (z_3, z_5, z_1), and then take sums and differences; or we can take the sums and differences first, and then apply the algorithm for n=3 to the results. In either case, the operation count is 36 real additions and 8 real multiplications. Including the n=6 algorithm within a conventional FFT enables some advantage to be taken of mutually prime factors, without going to the complexity of Good's algorithm.

Table I summarizes the operation counts for these fast "small transform" algorithms, and compares them with the corresponding basic algorithms using direct matrix multiplication (but avoiding redundant multiplications by ± 1 and $\pm i$).

To emphasize the point about separating out the multiplication by phase factors, consider the computation $\underset{\sim}{x} = \Omega W_6\underset{\sim}{z}$ where Ω is a diagonal matrix of complex numbers, the first of which is 1. Applying the algorithm above for n=6 and then multiplying the result by Ω takes a total of 46 real additions and 28 real multiplications. Direct multiplication of $\underset{\sim}{z}$ by the product matrix ΩW_6, on the other hand, would require 120 real additions and 120 real multiplications.

We are now ready to establish the operation count for the whole FFT algorithm. First, it is useful to note the operation count per real point for the computation $\underset{\sim}{x} = \Omega W_n \underset{\sim}{z}$. This is found by adding the operations for (n-1) complex multiplications to the "fast" entries of Table I, and dividing by 2n. The results are summarized in Table II.

So, for example, if n=2 is a factor of N used in the complex transform algorithm, then the pass corresponding to that factor will contribute 3N real additions and 2N real multiplications to the operation count. There is one saving which we have not yet taken into account; in the skeleton Fortran routine of Section 2, $\Omega(K)$ is the identity matrix for K=0, and the corresponding (n-1) complex multiplications can be omitted. The number of times this happens at each stage is given by the variable LA. If $N=n_1 n_2 n_3 \ldots n_k$, then at successive stages LA=1, n_1, $n_1 n_2$,... and the total number of complex multiplications saved is

$$(n_1 - 1) + n_1 (n_2 - 1) + n_1 n_2 (n_3 - 1) + \ldots + n_1 n_2 \ldots n_{k-1} (n_k - 1)$$

which is just (N-1). (Singleton [20] gives a similar demonstration).

Suppose we have factorized N in the form $N = 2^p 3^q 4^r 5^s 6^t$. Using the results of Table II, and including the above saving, the total operation count for the complex FFT algorithm is:
number of real additions:

$$2N(1.5p + 2.67q + 2.75r + 4s + 3.83t - 1) + 2 \; ;$$

number of real multiplications:

$$2N(p + 2q + 1.5r + 2.8s + 2.33t - 2) + 4 \; .$$

Notice, incidentally, that the operation count is independent of the order in which the factors are used.

Table III lists the number of operations required for a range of values of N, both for a basic FFT algorithm allowing only the prime factors 2, 3 and 5, and for a "de luxe" version including the factors 4 and 6. (In ambiguous cases, factors of 6 are extracted first).

Several observations are in order here. A direct calculation of the DFT would require almost $4N^2$ real additions and $4N^2$ multiplications; the counts in Table III are smaller by almost two orders of magnitude. Inclusion of the factors 4 and 6 gives a significant improvement. Finally, there is nothing very special (except program simplicity) about the case $N = 2^p$.

4. NOTES ON IMPLEMENTATION

Thus far we have considered the efficiency of the FFT algorithm only in terms of the number of arithmetic operations required. Just as important is the efficiency in terms of the utilization of the available computer hardware, and for current large-scale scientific applications this raises the question of vectorization.

The problem of vectorizing the FFT algorithm on the Cray-1 and Cyber 205 has been discussed in detail elsewhere [24, 25], and here only a summary will be given.

Basically we are concerned with the inner loop of the subroutine PASS outlined in Section 2. This loop vectorizes naturally with an increment of 1 between successive elements of vectors (assuming here that the real and imaginary parts of complex numbers have been stored in separate arrays). Note that the vectors we are discussing now should not be confused with the <u>mathematical</u> vectors $\underset{\sim}{A}(I)$ and $\underset{\sim}{C}(J)$, which were used only for compactness of notation; for clarification, consider the example of IFAC=2 given in Section 2. The vector length is LA, which unfortunately never becomes very long for the modest values of N encountered in most applications. For example if N=180 = 5 x 6 x 6 the vector length on successive passes is 1, 5, 30, using decimation in frequency.

On the Cray-1 (unlike the Cyber 205), the elements of a vector do not have to be contiguously stored, but can be spaced at any constant increment. This allows us to turn the nested loop structure of subroutine PASS "inside out", giving instead a vector length of N/(IFAC*LA). The case N = 180 = 5 x 6 x 6 can then be implemented with vector lengths 36, 6, 30 on successive passes, by choosing the most appropriate structure at each stage. Even on the Cray-1, which reaches half maximum speed with vector lengths as short as 10 or thereabouts, this is still some way from being optimal. Also the inside-out loop structure does have certain technical drawbacks as it involves additional memory references, redundant multiplications and a risk of bank conflicts [24].

As hinted in Section 2, the FFT variant due to Pease [18] appears to be more suitable for vectorization, since it can be implemented with a vector length of N/IFAC throughout. But as shown in [24] this is true only if the permutation step can be dispensed with; otherwise the advantage gained from the longer vectors is outweighed by the time required for reordering.

The solution adopted on the Cray-1 arises from the fact that we usually need to perform many transforms simultaneously. It is then easy to arrange the data so that the transforms are performed in parallel, with each vector containing one element from each transform, and the vector length equal to the number of transforms being performed. The details of the indexing are transferred to outer loops, and become irrelevant from the standpoint of vectorization. As the <u>optimum</u> vector length is 64, and we can usually arrange for the number of simultaneous transforms to be of that order or even greater, the problem was considered solved for that machine.

Table IV lists the times per transform for complex FFT's on the Cray-1 for various values of N, comparing the speed of single transforms (vectorized as efficiently as possible) with multiple transforms (64 in parallel). All routines were written in Fortran. As expected, the advantage gained by performing many transforms in parallel decreases as N increases, but even at N = 1024 it is still significant.

An interesting result emerges if the same programs are run on a serial computer; timings obtained on such a machine, the Cyber 175, are included in Table IV. The advantage of performing transforms in parallel is still quite marked, due to the reduction in overheads for loop control and subroutine calls. Somewhat similar results were presented in [16]; the lesson is that vector techniques can pay significant dividends even on non-vector machines.

On the Cyber 205, which requires vector lengths of order 100 to reach half maximum speed, there is no way in which a single transform of modest length could be vectorized very efficiently. Unfortunately, even the multiple vectorization scheme adopted on the Cray-1 proves to be inadequate on the Cyber 205. First, the requirement that vector elements be contiguous reduces our freedom to specify the layout of the data; the multiple approach only works if the simultaneous transforms are interleaved (i.e., the data vectors to be transformed are given as rows of a columnwise matrix). More seriously, the efficiency continues to improve as vector lengths increase (up to a maximum of 64K-1), and vector lengths should preferably be at least of order several hundred, typically rather more than the number of transforms which can conveniently be performed in parallel. Fortunately, interleaving the simultaneous transforms provides a solution. Recall from Section 2(d) that, for the self-sorting decimation-in-frequency algorithm for a single transform of length $N = pqr$, the <u>second</u> stage is equivalent to the <u>first</u> stage of p interleaved transforms each of length qr, and the <u>third</u> stage is equivalent to the <u>first</u> (and only) stage of pq interleaved transforms each of length r. Suppose now that we have M interleaved transforms each of length $N = pqr$. Applying the simple multiple vectorization technique gives a vector length of M throughout. However, from the remarks above we see that after the first pass the remainder of the process is equivalent to Mp interleaved transforms of length qr, and so the second pass can be done with a vector length Mp. The third pass is then equivalent to Mpq interleaved transforms of length r, and can be done with a vector length Mpq. Thus interleaving the transforms, which had to be done in order to implement the multiple approach at all, in fact results in significantly longer vectors after the first pass. This procedure could of course also be applied on the Cray-1, but there would be little gain unless the number of transforms was much less than 64.

An analogous result exists for the decimation-in-time algorithm; it was used by Korn and Lambiotte [16], who elevate it to the status of a theorem.

Finally, a note on assembly language programs. Implementing a multiple self-sorting mixed-radix FFT package in assembly language is a task not to be undertaken lightly, as the author will wearily testify; however, depending on the computer, the compiler and the application, it may be worthwhile. On the Cray-1, for example, a package written in CAL performs FFT's at around 100 megaflops, approximately twice the rate obtained in Fortran. The provision of vector registers, and the register-to-register arithmetic, offer ample scope for the CAL programmer to overlap arithmetic with memory references and to organize the scheduling of instructions at a level beyond the capabilities of the Fortran compiler. On the Cyber 205, on the other hand, the memory-to-memory arithmetic makes such considerations irrelevant, and very little is gained by the use of assembly language.

5. TWO-DIMENSIONAL TRANSFORMS

In some circumstances we need to perform a two-dimensional complex discrete Fourier transform, defined by

$$x_{j,k} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} z_{m,n} \exp(2ijm\pi/M) \exp(2ikn\pi/N). \qquad (22)$$

In the conventional approach, we first compute N independent DFT's given by

$$y_{j,n} = \sum_{m=0}^{M-1} z_{m,n} \exp(2ijm\pi/M)$$

and then compute M independent DFT's given by

$$x_{j,k} = \sum_{n=0}^{N-1} y_{j,n} \exp(2ikn\pi/N).$$

In other words, we first compute DFT's along rows of the grid, and then compute DFT's along columns of the grid (or, of course, vice versa).

It does not seem to be generally known that there is a faster way of implementing Eq. (22). Henrici [13] mentions an alternative approach, but states that it is only faster for the case $N=2^p$. However, this is because he uses a version of the FFT in which the rotation factors are not separated out. We will show here that the alternative approach can be faster for any composite N.

The two-dimensional FFT algorithm is most easily derived using matrix algebra. Eq. (22) may be written in the form

$$\tilde{x} = (W_N \times W_M)\, \tilde{z} \;.$$

For simplicity we will consider the case M = N = pq and choose the self-sorting variant of the one-dimensional algorithm, though any other variant would do equally well. From Eq. (7) we have

$$(W_{pq} \times W_{pq}) = [\,(W_q \times I_p)\, P_q^p\, D_q^p\, (W_p \times I_q)] \times [\,(W_q \times I_p)\, P_q^p\, D_q^p\, (W_p \times I_q)] \;.$$

Using the rules of matrix algebra for Kronecker products, we can rewrite this as

$$(W_{pq} \times W_{pq}) = (W_q \times I_p \times W_q \times I_p)(P_q^p \times P_q^p)(D_q^p \times D_q^p)(W_p \times I_q \times W_p \times I_q) \;. \qquad (23)$$

Eq. (23) is a direct two-dimensional analogue of Eq. (7). Reading the factors from right to left, we have first a set of q^2 (p x p) two-dimensional transforms, then a diagonal matrix of rotation factors, then a permutation matrix, and finally a set of p^2 (q x q) two-dimensional transforms. The important point is that the rotation factors have been multiplied together, and there are now only N^2 of them; in the conventional approach the N transforms along rows contribute N^2 rotation factors, and the N transforms along columns contribute a further N^2, so there are twice as many altogether. If redundant multiplications by 1 are avoided in each case, then the ratio becomes slightly less favourable, but Eq. (23) still represents a significant improvement. The extension to more factors is straightforward, as is the extension to the case $M \neq N$. By way of illustration, for $M=N=64=4^3$ the three-factor analogue of Eq. (23) includes 7047 non-redundant multiplications by rotation factors, while the conventional approach requires 10368.

The operation count for the rest of the algorithm, excluding the multiplications by rotation factors, is in general the same whichever approach is used. However, for certain factors (e.g. 3x3 and 5x5), the small two-dimensional transforms can themselves be implemented using fewer operations than the conventional approach, thus affording further savings; the tricks required here are analogous to those used by Winograd [29] for one-dimensional transforms.

Also straightforward is the extension to three or more dimensions; the higher the dimensionality of the transform, the greater the savings which can be made.

6. CONCLUDING REMARKS

The aim of this paper has been to present some of the ideas which have contributed towards the development of FFT packages used in numerical weather prediction models at ECMWF and the UK Meteorological Office. In particular the matrix algebra approach of Section 2 may help to spread understanding of how the many variants of the FFT algorithm can be derived. The skeleton Fortran routines, together with the small transform algorithms of Section 3, should provide a basis from which interested readers can construct FFT programs to suit their own requirements. An effort has been made to dispel some lingering myths about the FFT, to point out the advantages of the self-sorting variants and of vector techniques, and briefly to indicate how savings can be made in multi-dimensional transforms. Finally, it is hoped that something of the intriguing character of the algorithm has been conveyed.

TABLE I

Operation counts for transforms of length n

	BASIC		FAST	
n	adds	mults	adds	mults
2	4	0	4	0
3	20	16	12	4
4	24	0	16	0
5	72	64	32	12
6	92	64	36	8

TABLE II

Operations per real point for $\underset{\sim}{x} = \Omega W_n \underset{\sim}{z}$

n	adds	mults
2	1.5	1
3	2.67	2
4	2.75	1.5
5	4	2.8
6	3.83	2.33

TABLE III

Real operation counts for complex FFT's of length N

N	n=2,3,5 adds	n=2,3,5 mults	n=2,3,4,5,6 adds	n=2,3,4,5,6 mults
$180 = 2^2 . 3^2 . 5$	4082	2452	3842	1972
$192 = 2^6 . 3$	4098	2308	3778	1668
$200 = 2^3 . 5^2$	4602	2644	4502	2444
$216 = 2^3 . 3^3$	4970	3028	4538	2164
$240 = 2^4 . 3 . 5$	5602	3268	5322	2708
$256 = 2^8$	5634	3076	5122	2052

TABLE IV

Times per transform (ms) for complex FFT's

N	Cray-1 single	Cray-1 multiple	Cyber 175 single	Cyber 175 multiple
32	.074	.012	.750	.280
36	.086	.016	.780	.330
96	.150	.052	1.940	1.110
100	.160	.064	2.090	1.550
1024	1.110	.760	21.140	19.000

REFERENCES.

1. R.BELLMAN, "Introduction to Matrix Analysis", McGraw-Hill, 1970.

2. E.O. BRIGHAM, "The Fast Fourier Transform", Prentice-Hall, Englewood Cliffs, N.J., 1974.

3. W. BOURKE, An efficient, one-level, primitive-equation spectral model, Mon.Wea.Rev. 100 (1972) 683-689.

4. D.M. BURRIDGE and C. TEMPERTON, A fast Poisson-solver for large grids, J.Computational Phys. 30 (1979), 145-148.

5. W.T. COCHRAN ET AL., What is the Fast Fourier Transform? IEEE Trans.Audio and Electroacoustics 15 (1967), 45-55.

6. J.W. COOLEY AND J.W. TUKEY, An algorithm for the machine calculation of complex Fourier series, Math.Comp. 19 (1965), 297-301.

7. C. DE BOOR, FFT as nested multiplication, with a twist, SIAM J. Sci. Stat. Comput. 1 (1980), 173-178.

8. M. DRUBIN, Kronecker product factorization of the FFT matrix, IEEE Trans. Computers 20 (1971), 590-593.

9. W.M. GENTLEMAN AND G. SANDE, Fast Fourier Transforms - for fun and profit, Proc. AFIPS Joint Computer Conference 29 (1966), 563-578.

10. J.A. GLASSMAN, A generalization of the fast Fourier transform, IEEE Trans. Computers 19 (1970), 105-116.

11. B. GOLD AND C.M. RADER, "Digital Processing of Signals", McGraw-Hill, 1969.

12. I.J. GOOD, The interaction algorithm and practical Fourier analysis, J.Roy.Statist.Soc. Ser.B 20 (1958), 361-372.

13. P. HENRICI, Fast Fourier methods in computational complex analysis, SIAM Rev. 21 (1979), 481-527.

14. J.L. HOLLOWAY, M.J. SPELMAN AND S. MANABE, Latitude-longitude grid suitable for numerical time integration of a global atmospheric model, Mon.Wea.Rev. 101 (1973), 69-78.

15. D. KAHANER, Matrix description of the Fast Fourier Transform, IEEE Trans. Audio and Electroacoustics 18 (1970), 442-450.

16. D.G. KORN AND J.J. LAMBIOTTE, Computing the Fast Fourier Transform on a vector computer, Math. Comp. 33 (1979), 977-992.

17. S.A. ORSZAG, Numerical simulation of incompressible flows within simple boundaries, I: Galerkin (spectral) representations, Stud.Appl.Math. 50 (1971), 293-327.

18. M.C. PEASE, An adaptation of the Fast Fourier Transform for parallel processing, J.ACM 15 (1968), 252-264.

19. A. ROBERT, J. HENDERSON AND C. TURNBULL, An implicit time integration scheme for baroclinic models of the atmosphere, Mon.Wea.Rev. 100 (1972), 329-335.

20. R.C. SINGLETON, An algorithm for computing the mixed radix Fast Fourier Transform, IEEE Trans. Audio and Electroacoustics 17 (1969), 93-103.

21. R.C. SINGLETON, On computing the Fast Fourier Transform, C. ACM 10 (1967), 647-654.

22. G. STRANG, "Linear Algebra and its Applications", Academic Press, 1980 (2nd edition).

23. P. N. SWARTZTRAUBER, The direct solution of the discrete Poisson equation on the surface of a sphere, J.Computational Phys. 15 (1974), 46-54.

24. C. TEMPERTON, Fast Fourier Transforms and Poisson-solvers on Cray-1, in "Supercomputers", Infotech State of the Art Report, Infotech International Ltd., Maidenhead, U.K., 1979.

25. C. TEMPERTON, Fast real Fourier Transforms on the Cyber 205, Met. O.11 Technical Note No.155, Meteorological Office, U.K., 1982.

26. F. THEILHEIMER, A matrix version of the Fast Fourier Transform, IEEE Trans. Audio and Electroacoustics 17 (1969), 158-161.

27. M.J. UHRICH, Fast Fourier Transforms without sorting, IEEE Trans. Audio and Electroacoustics 17 (1969), 170-172.

28. D.L. WILLIAMSON, Linear stability of finite-difference approximations on a uniform latitude-longitude grid with Fourier filtering, Mon.Wea.Rev. 104 (1976), 31-41.

29. S. WINOGRAD, On computing the discrete Fourier transform, Math. Comp. 32 (1978), 175-199.