# An Algorithm for Computing the Mixed Radix Fast Fourier Transform

RICHARD C. SINGLETON, Senior Member, IEEE
Stanford Research Institute
Menlo Park, Calif. 94025

**Abstract**

**This paper presents an algorithm for computing the fast Fourier transform, based on a method proposed by Cooley and Tukey. As in their algorithm, the dimension $n$ of the transform is factored (if possible), and $n/p$ elementary transforms of dimension $p$ are computed for each factor $p$ of $n$. An improved method of computing a transform step corresponding to an odd factor of $n$ is given; with this method, the number of complex multiplications for an elementary transform of dimension $p$ is reduced from $(p-1)^2$ to $(p-1)^2/4$ for odd $p$. The fast Fourier transform, when computed in place, requires a final permutation step to arrange the results in normal order. This algorithm includes an efficient method for permuting the results in place. The algorithm is described mathematically and illustrated by a FORTRAN subroutine.**

Manuscript received December 2, 1968.

This work was supported by Stanford Research Institute, out of Research and Development funds.

## Introduction

The fast Fourier transform (FFT) algorithm is an efficient method for computing the transformation

$$\alpha_k = \sum_{j=0}^{n-1} x_j \exp(i2\pi jk/n) \tag{1}$$

for $k=0, 1, \cdots, n-1$, where $\{x_j\}$ and $\{\alpha_k\}$ are both complex-valued. The basic idea of the current form of the fast Fourier transform algorithm, that of factoring $n$,

$$n = \prod_{i=1}^{m} n_i,$$

and then decomposing the transform into $m$ steps with $n/n_i$ transformations of size $n_i$ within each step, is that of Cooley and Tukey [1]. Most subsequent authors have directed their attention to the special case of $n=2^m$. Explanation and programming are simpler for $n=2^m$ than for the general case, and the restricted choice of values of $n$ is adequate for a majority of applications. There are, however, some applications in which a wider choice of values of $n$ is needed. The author has encountered this need in spectral analysis of speech and economic time series data.

Gentleman and Sande [2] have extended the development of the general case and describe possible variations in organizing the algorithm. They mention the existence of a mixed radix FFT program written by Sande. Available mixed radix programs include one in ALGOL by Singleton [3] and another in FORTRAN by Brenner [4]. A FORTRAN program based on the algorithm discussed here is included in Appendix I; this program was compared with Brenner's on several computers (CDC 6400, CDC 6600, IBM 360/67, and Burroughs B5500) and found to be significantly faster.

## The Mixed Radix FFT

The complex Fourier transform (1) can be expressed as a matrix multiplication

$$\alpha = Tx,$$

where $T$ is an $n \times n$ matrix of complex exponentials

$$t_{jk} = \exp(i2\pi jk/n).$$

In decomposing the matrix $T$, we use the factoring of Sande [2], rather than the original factoring of Cooley [1]. However, if the data are first permuted to digit-reversed order and then transformed, Cooley's factoring leads to an equally efficient algorithm.

In computing the fast Fourier transform, we factor $T$ as

$$T = PF_mF_{m-1} \cdots F_2F_1,$$

where $F_i$ is the transform step corresponding to the factor $n_i$ of $n$ and $P$ is a permutation matrix. The matrix $F_i$ has only $n_i$ nonzero elements in each row and column and can be partitioned into $n/n_i$ square submatrices of dimension $n_i$; it is this partition and the resulting reduction in multiplications that is the basis for the FFT algorithm. The matrices $F_i$ can be further factored to yield

$$F_i = R_i T_i,$$

where $R_i$ is a diagonal matrix of rotation factors (called twiddle factors by Gentleman and Sande [2]) and $T_i$ can be partitioned into $n/n_i$ identical square submatrices, each the matrix of a complex Fourier transform of dimension $n_i$. Although it might appear that this step increases the number of complex multiplications, it in fact enables us to exploit trigonometric function symmetries and multipliers of simple form (e.g., $e^{i\pi}$, $e^{i\pi/2}$, and $e^{i\pi/4}$) in computing $T_i$ that more than compensate for the fewer than $n$ multiplications in applying the rotation $R_i$. This point will be discussed further in later sections.

The permutation $P$ is required because the transformed result is initially in digit-reversed order, i.e., the Fourier coefficient $\alpha_j$, with

$$j = j_m n_{m-1} n_{m-2} \cdots n_1 + \cdots + j_2 n_1 + j_1,$$

is found in location

$$j' = j_1 n_2 n_3 \cdots n_m + j_2 n_3 n_4 \cdots n_m + \cdots + j_m.$$

As mentioned previously by the author [5], the permutation may be performed in place by pair interchanges if $n$ is factored so that

$$n_i = n_{m-i}$$

for $i < n-i$. In this case, we can count $j$ in natural order and $j'$ in digit-reversed order, then exchange $\alpha_j$ and $\alpha_{j'}$ if $j < j'$. This method is a generalization of a well-known method for reordering the radix-2 FFT result.

Before computing the Fourier transform, we first decompose $n$ into its prime factors. The square factors are arranged symmetrically about the factors of the square-free portion of $n$. Thus $n = 270$ is factored as

$$3 \times 2 \times 3 \times 5 \times 3.$$

Then the permutation $P$ is factored into two steps,

$$P = P_2 P_1.$$

The permutation $P_1$ is associated with the square factors of $n$ and is done by pair interchanges as described above, except that the digits of $n$ corresponding to the square-free factors are held constant and the digits of the square factors are exchanged symmetrically. Thus if

$$n = n_1 n_2 n_3 n_4 n_5 n_6 n_7$$

with $n_1 = n_7$, $n_2 = n_6$, and $n_3$, $n_4$, and $n_5$ are relatively prime, we interchange

$$j = j_7 n_6 n_5 \cdots n_1 + j_6 n_5 n_4 \cdots n_1 + j_5 n_4 n_3 n_2 n_1 + j_4 n_3 n_2 n_1 + j_3 n_2 n_1 + j_2 n_1 + j_1$$

and

$$j' = j_1 n_6 n_5 \cdots n_1 + j_2 n_5 n_4 \cdots n_1 + j_5 n_4 n_3 n_2 n_1 + j_4 n_3 n_2 n_1 + j_3 n_2 n_1 + j_5 n_1 + j_6.$$

The permutation $P_1$ in this case leaves each result element in its correct segment of length $n/n_1 n_2$, grouped in subsequences of $n_1 n_2$ consecutive elements. The permutation $P_2$ then completes the reordering by permuting the $n_3 n_4 n_5$ subsequences within each segment of length $n/n_1 n_2$. In the FORTRAN subroutine given in Appendix I, $P_2$ is done by first determining the permutation cycles for digit reversal of the digits corresponding to the square-free factors, then permuting the data following these cycles. The permutation can be done using as few as two elements of temporary storage to hold a single complex result, but the program uses its available array space to permute the subsequences of length $n_1 n_2$ if possible.

## The Rotation Factor

In the previous section, we described the factoring of the transform step $F_i$, corresponding to a factor $n_i$ of $n$, into a product $R_i T_i$ of a matrix $T_i$ of $n/n_i$ identical Fourier transforms of dimension $n_i$ and a diagonal rotation factor matrix $R_i$. Here we specify the elements of $R_i$ for the Sande version of the FFT.

The rotation factor $R_i$ following the transform step $T_i$ has diagonal elements

$$r_j = \exp\left\{ i \frac{2\pi}{kk} (j \bmod k) \left[ \frac{j \bmod kk}{k} \right] \right\}$$

for $j = 0, 1, \cdots, n-1$ where

$$k = n/n_1 n_2 \cdots n_i \quad \text{and} \quad kk = n_i k,$$

and the square brackets [ ] denote the greatest integer $\leq$ the enclosed quantity. The rotation factors multiplying each transform of dimension $n_i$ within $T_i$ have angles

$$0, \theta, 2\theta, \cdots (n_i - 1)\theta,$$

where $\theta$ may differ from one transform to another. No multiplication is needed for the zero angle, thus there are at most

$$n(n_i - 1)/n_i$$

complex multiplications to apply the rotation factor following the transform step $T_i$. In addition, $\theta = 0$ for $(j \bmod k) = 0$ or $(j \bmod kk) = 0$, allowing the number of complex multiplications to be reduced by

$$(n_1 - 1) + n_1(n_2 - 1) + n_1 n_2 (n_3 - 1) + \cdots + n_1 n_2 \cdots n_{m-1}(n_m - 1) = n - 1.$$

We note that the number of rotation factor multiplications is independent of the order of arrangement of the factors of $n$. The final rotation factor $R_m$ has $\theta = 0$ for all elements, and thus is omitted.

## Counting Complex Multiplications

A complex multiplication, requiring four real multiplications and two real additions, is a relatively slow operation on most computers.[1] To a first approximation, the speed of an FFT algorithm is proportional to the number of complex multiplications used. The number of times we index through the data array is, however, an important secondary factor.

Using the results of the previous section, the number of complex multiplications for the rotation factors $R_i$ is

$$\sum_{i=1}^{m} \frac{n(n_i - 1)}{n_i} - (n - 1),$$

assuming we avoid multiplication for all rotations of zero angle. To this number we must add the multiplications for the transform steps $T_i$.

For $n$ a power of 2, we note that a complex Fourier transform of dimension 2 or 4 can be computed without multiplication and that a transform of dimension 8 requires only two real multiplications, equivalent to one-half a complex multiplication. Going one step further, a transform of dimension 16, computed as two factors of 4, requires the equivalent of six complex multiplications. Combining these results with the number of rotation factor multiplications and assuming that $n = 2^m$ is a power of the radix, the total number of complex multiplications is as follows:

| Radix | Number of Complex Multiplications |
|---|---|
| 2 | $mn/2-(n-1)$ |
| 4 | $3mn/8-(n-1)$ |
| 8 | $mn/3-(n-1)$ |
| 16 | $21mn/64-(n-1)$ |

These results have been given previously by Bergland [6]. The savings for 16 over 8 is small, considering the added complexity of the algorithm. As Bergland points out, radix 8, with provision for an additional factor of 4 or 2, is a good choice for an efficient FFT program for powers of 2. For the mixed radix FFT, we transform with factors of 4 whenever possible, but also provide for factors of 2.

We now consider the number of complex multiplications for a radix-$p$ transform of $n = p^m$ complex data values, where $p$ is an odd prime. While at first it might appear that an elementary transform of dimension $p$ requires $(p-1)^2$ complex multiplications, we show in the next section that $(p-1)^2$ real multiplications suffice, equivalent to $(p-1)^2/4$ complex multiplications. This result holds, in fact, for any odd value of $p$. Thus the transform steps for $n = p^m$ require the equivalent of

$$\frac{mn(p-1)^2}{4p}$$

complex multiplications. Adding the

$$\frac{mn(p-1)}{p} - (n-1)$$

multiplications for rotation factors, we obtain a total of

$$\frac{mn(p-1)(p+3)}{4p} - (n-1)$$

complex multiplications for a radix-$p$ transform. Since $p$ is assumed here to be an odd prime, we have no rotations with $\theta$ an integer multiple of $\pi/4$ to reduce further the number of complex multiplications.

The ratio of the number of complex multiplications to $n \log_2 n$ can serve as a measure of relative efficiency for the mixed radix FFT. The results of this section, neglecting the reduction by $n-1$ for $\theta = 0$, yield the following comparison:

| Radix | Relative Efficiency |
|---|---|
| 2 | 0.500 |
| 4 | 0.375 |
| 8 | 0.333 |
| 16 | 0.328 |
| 3 | 0.631 |
| 5 | 0.689 |
| 7 | 0.763 |
| 11 | 0.920 |
| 13 | 0.998 |
| 17 | 1.151 |
| 19 | 1.227 |
| 23 | 1.374 |

The general term for an odd prime $p$ is

$$\frac{(p-1)(p+3)}{4p \log_2 (p)}.$$

## Decomposition of a Complex Fourier Transform

In the previous section, we promised to show that a complex transform of dimension $p$, for $p$ odd, can be computed with $(p-1)^2$ real multiplications. Consider the complex transform

$$\begin{aligned} a_k + ib_k &= \sum_{j=0}^{p-1} (x_j + iy_j) \left\{ \cos\left(\frac{2\pi jk}{p}\right) + i \sin\left(\frac{2\pi jk}{p}\right) \right\} \\ &= x_0 + \sum_{j=1}^{p-1} x_j \cos\left(\frac{2\pi jk}{p}\right) - \sum_{j=1}^{p-1} y_j \sin\left(\frac{2\pi jk}{p}\right) \\ &\quad + i \left\{ y_0 + \sum_{j=1}^{p-1} y_j \cos\left(\frac{2\pi jk}{p}\right) + \sum_{j=1}^{p-1} x_j \sin\left(\frac{2\pi jk}{p}\right) \right\} \end{aligned}$$

[1] G. Golub (private communication) has pointed out that a complex multiplication can alternatively be done with three real multiplications and five real additions, as indicated by the following:

$$(a+ib)\cdot(c+id) = [(a+b)\cdot(c-d)+a\cdot d-b\cdot c]+i[a\cdot d+b\cdot c].$$

This method does not appear advantageous for FORTRAN coding, as the number of statements is increased from two to four.

$$
\begin{aligned}
&= x_0 + \sum_{j=1}^{(p-1)/2} (x_j + x_{p-j}) \cos\left(\frac{2\pi jk}{p}\right) \\
&\quad - \sum_{j=1}^{(p-1)/2} (y_j - y_{p-j}) \sin\left(\frac{2\pi jk}{p}\right) \\
&\quad + i\left\{y_0 + \sum_{j=1}^{(p-1)/2} (y_j + y_{p-j}) \cos\left(\frac{2\pi jk}{p}\right)\right. \\
&\quad \left. + \sum_{j=1}^{(p-1)/2} (x_j - x_{p-j}) \sin\left(\frac{2\pi jk}{p}\right)\right\}
\end{aligned}
$$

for $k=0, 1, \cdots, p-1$. We note first that

$$
a_0 + ib_0 = \sum_{j=0}^{p-1} (x_j + iy_j)
$$

is computed without multiplications. The remaining Fourier coefficients can be expressed as

$$
\begin{aligned}
a_k &= a_k^+ - a_k^- \\
a_{p-k} &= a_k^+ + a_k^- \\
b_k &= b_k^+ + b_k^- \\
b_{p-k} &= b_k^+ - b_k^-
\end{aligned}
$$

for $k=1, 2, \cdots, (p-1)/2$, where

$$
\begin{aligned}
a_k^+ &= x_0 + \sum_{j=1}^{(p-1)/2} (x_j + x_{p-j}) \cos\left(\frac{2\pi jk}{p}\right) \\
a_k^- &= \sum_{j=1}^{(p-1)/2} (y_j - y_{p-j}) \sin\left(\frac{2\pi jk}{p}\right) \\
b_k^+ &= y_0 + \sum_{j=1}^{(p-1)/2} (y_j + y_{p-j}) \cos\left(\frac{2\pi jk}{p}\right) \\
b_k^- &= \sum_{j=1}^{(p-1)/2} (x_j - x_{p-j}) \sin\left(\frac{2\pi jk}{p}\right).
\end{aligned}
$$

Altogether there are $2(p-1)$ series to sum, each with $(p-1)/2$ multiplications, for a total of $(p-1)^2$ real multiplications.

For $p$ an odd prime and for fixed $j$, the multipliers

$$
\cos(2\pi jk/p) \qquad \text{for } k = 1, 2, \cdots (p-1)/2
$$

have no duplications of magnitude, thus no further reduction in multiplications appears possible.[2] The same condition holds for the multipliers

$$
\sin(2\pi jk/p) \qquad \text{for } k = 1, 2, \cdots (p-1)/2.
$$

[2] C. M. Rader (private communication) has proposed an alternative decomposition of a Fourier transform of dimension 5, using the equivalent of 3 complex multiplications (12 real multiplications) instead of the 4 complex multiplications used in the algorithm described in this paper. In Appendix III we give a FORTRAN coding of Rader's method. When substituted in subroutine FFT (Appendix I), times were unchanged on the CDC 6600 computer and improved by about 5 percent for radix-5 transforms on the CDC 6400 computer (the 6400 has a relatively slower multiply operation). Rader's method looks advantageous for coding in machine language on a computer having multiple arithmetic registers available for temporary storage of intermediate results.

For even values of $p$, a decomposition similar to the above yields $4(p/2-1)$ series to sum, each with $(p/2-1)$ multiplications. Thus a complex Fourier transform for $p$ even can be computed with at most $(p-2)^2$ real multiplications. For $p>2$, we know that this result can be improved. Combining results for the odd and even cases, we can state that a Fourier transform of dimension $p$ can be computed with the equivalent of

$$
\left[\frac{p-1}{2}\right]^2
$$

or fewer complex multiplications, where the square brackets [ ] denote the largest integer value $\leq$ the enclosed quantity.

## A Method for Computing Trigonometric Function Values

The trigonometric function values used in the fast Fourier transform can all be represented in terms of integer powers of

$$
\exp(i2\pi/n),
$$

the $n$th root of unity. Since we often use a sequence of equally spaced values on the unit circle, it is useful to have accurate methods of generating them by complex multiplication, rather than by repeated use of the library sine and cosine functions. For very short sequences, we use the simple method

$$
\xi_{k+1} = \xi_k \exp(i\theta),
$$

where

$$
\xi_0 = 1
$$

and $\{\xi_k\}$ is the sequence of computed values $\exp(ik\theta)$. This method suffers, however, from rapid accumulation of round-off errors. A better method, proposed by the author in an earlier paper [5], is to use the difference equation

$$
\xi_{k+1} = \xi_k + \eta\xi_k,
$$

where the multiplier

$$
\begin{aligned}
\eta &= \exp(i\theta) - 1 \\
&= 2i \sin(\theta/2) \exp(i\theta/2) \\
&= -2\sin^2(\theta/2) + i \sin(\theta)
\end{aligned}
$$

decreases in magnitude with decreasing $\theta$. This method gives good accurcay on a computer using rounded floating-point arithmetic (e.g., the Burroughs B5500). However, with truncated arithmetic (as on the IBM 360/67), the value of $\xi_k$ tends to spiral inward from the unit circle with increasing $k$.

In Table I, we show the accumulated errors from extrapolating to $\pi/2$ in $2^k$ increments, using rounded arithmetic (machine language) and truncated arithmetic (FORTRAN) on a CDC 6400 computer; identical initial values, from the library sine and cosine functions, were used in computing the results in each of the three pairs of columns. In examining the second pair of columns, we find that the angle after $2^k$ extrapolation steps is very close to $\pi/2$, but that the magnitude has shrunk through truncation. To

TABLE I

Extrapolated Values of cos $\pi/2$ and sin $\pi/2-1$ on a CDC 6400 Computer, Using Rounded and Truncated Arithmetic Operations (Values in Units of $10^{-14}$)

| Number of Extrapolations | Rounded Arithmetic Without Correction | | Truncated Arithmetic Without Correction | | Truncated Arithmetic With Correction | |
|---|---|---|---|---|---|---|
| | $\cos \pi/2$ | $\sin \pi/2-1$ | $\cos \pi/2$ | $\sin \pi/2-1$ | $\cos \pi/2$ | $\sin \pi/2-1$ |
| $2^4$ | 2.6 | 0.0 | 2.9 | −3.6 | 2.8 | −0.4 |
| $2^5$ | 3.0 | −0.7 | 4.2 | −8.2 | 3.9 | −0.4 |
| $2^6$ | 2.5 | 0.0 | 3.8 | −13.1 | 4.1 | −0.4 |
| $2^7$ | 5.1 | −0.7 | 4.0 | −26.3 | 3.6 | −0.4 |
| $2^8$ | 2.7 | −0.7 | 4.5 | −51.9 | 4.5 | −0.4 |
| $2^9$ | 4.9 | −1.1 | 5.6 | −104.4 | 5.2 | −0.4 |
| $2^{10}$ | 8.8 | −1.1 | 5.2 | −213.2 | 4.7 | −0.4 |
| $2^{11}$ | 14.2 | 2.1 | 2.8 | −426.0 | 0.0 | −0.4 |
| $2^{12}$ | 13.2 | −0.4 | −1.1 | −866.2 | −5.4 | −0.4 |
| $2^{13}$ | −0.2 | −0.4 | 4.7 | −1705.7 | 4.2 | −0.4 |
| $2^{14}$ | −7.4 | 1.4 | −9.6 | −3440.1 | 1.2 | −0.4 |
| $2^{15}$ | −10.3 | −1.8 | −9.4 | −6841.5 | 36.9 | −0.4 |
| $2^{16}$ | 19.8 | 2.8 | 4.4 | −13707.1 | −33.3 | −0.4 |
| $2^{17}$ | 18.7 | 5.7 | −8.9 | −27416.7 | 3.5 | −0.4 |

compensate for this shrinkage, we modify the above method to restore the extrapolated value to unit magnitude. We first compute a trial value

$$\gamma_k = \xi_k + \eta\xi_k$$

where

$$\eta = -2\sin^2(\theta/2) + i\sin(\theta)$$

and

$$\xi_0 = 1,$$

then multiply by a scalar

$$\xi_{k+1} = \delta_k\gamma_k,$$

where

$$\delta_k \approx \frac{1}{\sqrt{\gamma_k\gamma_k^*}},$$

to obtain the new value. Since $\gamma_k\gamma_k^*$ is very close to 1, we can avoid the library square-root function and use the approximation

$$\delta_k = \frac{1}{2}\left(\frac{1}{\gamma_k\gamma_k^*} + 1\right).$$

Or if division is more costly than multiplication, we can alternatively use the approximation

$$\delta_k = \tfrac{1}{2}(3 - \gamma_k\gamma_k^*).$$

On the CDC 6400 computer, both approximations give the results shown in the third pair of columns of Table I. This rescaling of magnitude uses four real multiplications and a divide (or five real multiplications) in addition to the four real multiplications to compute the trial value of $\gamma_k$. However, on most computers, these calculations will take less time than computing the values using the library trigonometric function.

The added step of rescaling the extrapolated trigonometric function values to the unit circle can also be used when computing with rounded arithmetic, but the gain in accuracy is small. The subroutines in Appendixes I and II include comment cards indicating the changes to remove the rescaling. On the other hand, the number of multiplications may be reduced by one when using truncated arithmetic, through using the overcorrection multiplier

$$\delta_k = 2 - \gamma_k\gamma_k^*.$$

In this case, the truncation bias stabilizes a method that mathematically borders on instability. On the CDC 6400 computer, this multiplier gives comparable accuracy to the multiplier suggested above.

## A FORTRAN Subroutine for the Mixed Radix FFT

In Appendix I, we list a FORTRAN subroutine for computing the mixed radix FFT or its inverse, using the algorithm described above. This subroutine computes either a single-variate complex Fourier transform or the calculation for one variate of a multivariate transform.

To compute a single-variate transform (1) of $n$ data values,

$$\text{CALL FFT}(A, B, n, n, n, 1).$$

The "inverse" transform

$$x_j = \sum_{k=0}^{n-1} \alpha_k \exp(-i2\pi jk/n)$$

is computed by

$$\text{CALL FFT}(A, B, n, n, n, -1).$$

Scaling is left to the user. The two calls in succession give the transformation

$$T^*Tx = nx,$$

i.e., $n$ times the original values, except for round-off errors. The arrays $A$ and $B$ originally hold the real and imaginary components of the data, indexed from 1 to $n$;

the data values are replaced by the complex Fourier coefficients. Thus the real component of $\alpha_k$ is found in $A(k+1)$, and the imaginary component in $B(k+1)$, for $k=0, 1, \cdots, n-1$.

The difference between the transform and inverse calculation is primarily one of changing the sign of a variable holding the value $2\pi$. The one additional change is to follow an alternative path within the radix-4 section of the program, using the angle $-\pi/2$ rather than $\pi/2$.

The use of the subroutine for multivariate transforms is described in the comments at the beginning of the program. To compute a bivariate transform on data stored in rectangular arrays $A$ and $B$, the subroutine is called once to transform the columns and again to transform the rows. A multivariate transform is essentially a single-variate transform with modified indexing.

The subroutine as listed permits a maximum prime factor of 23, using four arrays of this dimension. The dimension of these arrays may be reduced to 1 if $n$ contains no prime factors greater than 5. An array NP(209) is used in permuting the results to normal order; the present value permits a maximum of 210 for the product of the square-free factors of $n$. If $n$ contains at most one square-free factor, the dimension of this array can be reduced to $j+1$, where $j$ is the maximum number of prime factors of $n$. A sixth arrray NFAC(11) holds the factors of $n$. This is ample for any transform that can be done on a computer with core storage for $2^{17}$ real values ($2^{16}$ complex values);

$$52\,488 = 2 \times 3^4 \times 2 \times 3^4 \times 2$$

is the only number $<2^{16}$ with as many as 11 factors, given the factoring used in this algorithm. The existing array dimensions do not permit unrestricted choice of $n$, but they rule out only a very small percentage of the possible values.

The transform portion of the subroutine includes sections for factors of 2, 3, 4, and 5, as well as a general section for odd factors. The sections for 2 and 4 include multiplication of each result value by the rotation factor; combining the two steps gives about a 10 percent speed improvement over using the general rotation factor section in the program, due to reduced indexing. The sections for 3 and 5 are similar to the general odd factors section, and they improve speed substantially for these factors by reducing indexing operations. The odd factors section is used for odd primes $>5$, but can handle any odd factor. The rotation factor section works for any factor but is used only for odd factors.

The permutation for square factors of $n$ contains special code for single-variate transforms, since less indexing is required. However, the permutation for multivariate transforms also works on single-variate transforms.

The author has previously published an ALGOL procedure [3] of the same name and with a similar function. One significant difference between the two algorithms is that the ALGOL one is organized for computing large transforms on a virtual core memory system (e.g., the Burroughs B5500 computer). This constraint leads to a small loss in efficiency compared with the present algorithm. In the ALGOL algorithm, the Cooley version of the FFT algorithm is used, with a simulated recursive structure; rotation factor multiplication is included within the transform phase, requiring two additional arrays with dimension equal to the largest prime factor in $n$. The transform method for odd factors is like that used here. The permutation for square factors of $n$ also has a simulated recursive structure, with one level of "recursion" for each square factor in $n$; in the present algorithm, this permutation is consolidated into a single step. The permutation for square-free factors is identical in both algorithms. The ALGOL algorithm contains a number of dynamic arrays, which is an obstacle to translation to FORTRAN. On the other hand, the FORTRAN subroutine given here can easily be translated to ALGOL, with the addition of dynamic upper bounds on all arrays other than NFAC; in making this translation, it would be desirable to modify the data indexing to go from 0 to $n-1$ to correspond with the mathematical notation.

## Timing and Accuracy

The subroutine FFT was tested for time and accuracy on a CDC 6400 computer at Stanford Research Institute. The results are shown in Table II. The times are central processor times, which are measured with 0.002 second resolution; the times measured on successive runs rarely differ by more than 0.002 to 0.004 second. Furthermore, calling the subroutine with $n=2$ yields a timing result of 0 or 0.002 second; thus the time is apparently measured with negligible bias.

The data used in the trials were random normal deviates with a mean of zero and a standard deviation of one (i.e., an expected rms value of one). The subroutine was called twice:

$$\text{CALL FFT}(A, B, n, n, n, 1)$$

$$\text{CALL FFT}(A, B, n, n, n, -1);$$

then the result was scaled by $1/n$. The squared deviations from the original data values were summed, the real and imaginary quantities separately, then divided by $n$ and square roots taken to yield an rms error value. The two values were in all cases comparable in magnitude, and an average is reported in Table II.

The measured times were normalized in two ways, first by dividing by

$$n \sum_{i=1}^{m} n_i,$$

and second by dividing by

$$n \log_2(n).$$

To a first approximation, computing time for the mixed radix FFT is proportional to $n$ times the sum of the factors of $n$, and we observe in the present case that a proportionality constant of 25 $\mu$s gives a fair fit to this model.

**TABLE II**

**Timing and Accuracy Tests of Subroutine FFT on a CDC 6400 Computer**

| Factoring of $n$ | Time (seconds) | Time / $n\sum n_i$ ($\mu$s) | Time / $n \log_2 n$ ($\mu$s) | rms Error ($\times 10^{-13}$) |
|---|---|---|---|---|
| $512 = 4^2 \times 2 \times 4^2$ | 0.188 | 20.4 | 40.8 | 1.1 |
| $1024 = 4^2 \times 4 \times 4^2$ | 0.398 | 19.4 | 38.9 | 1.2 |
| $2048 = 4^2 \times 2 \times 2 \times 2 \times 4^2$ | 0.928 | 20.6 | 41.2 | 1.4 |
| $4096 = 4^3 \times 4^3$ | 1.864 | 19.0 | 37.9 | 1.5 |
| $2187 = 3^3 \times 3 \times 3^3$ | 1.494 | 32.5 | 61.6 | 1.6 |
| $3125 = 5^2 \times 5 \times 5^2$ | 1.898 | 24.3 | 52.3 | 2.3 |
| $2401 = 7^2 \times 7^2$ | 2.310 | 34.4 | 85.7 | 2.6 |
| $1331 = 11 \times 11 \times 11$ | 1.324 | 30.1 | 95.9 | 2.5 |
| $2197 = 13 \times 13 \times 13$ | 2.478 | 28.9 | 101.6 | 3.5 |
| $289 = 17 \times 17$ | 0.272 | 27.7 | 115.1 | 2.5 |
| $361 = 19 \times 19$ | 0.372 | 27.1 | 121.3 | 3.2 |
| $529 = 23 \times 23$ | 0.636 | 26.1 | 132.9 | 3.5 |
| $1000 = 2 \times 5 \times 2 \times 5 \times 2 \times 5$ | 0.546 | 26.0 | 54.8 | 1.6 |
| $2000 = 4 \times 5 \times 5 \times 5 \times 4$ | 1.042 | 22.6 | 47.5 | 1.7 |

On the basis of counting complex multiplications, we would expect a decline in this proportionality constant with increasing radix; a decline is observed for odd primes$>5$. Factors of 5 or less are of course favored by special coding in the program. The second normalized time value places all times on a comparable scale, allowing one to assess the relative efficiency of using values of $n$ other than powers of 2; these results follow closely the relative efficiency values derived in an earlier section by counting complex multiplications, except that radix 5 is substantially better than predicted.

In Table III, we list the numbers up to 100 000 containing no prime factor greater than 5 to aid the user in selecting efficient values of $n$.

When compared with Brenner's FORTRAN subroutine [6] on the CDC 6400 computer, FFT was about 8 percent faster for radix 2, about 50 percent faster for radix 3 and 5, and about 22 percent faster for odd prime radix$\geq 7$. Brenner's subroutine also requires working storage array space equal to that used for data when computing other than radix-2 transforms.

The FORTRAN style in the subroutine FFT was designed to simplify hand compiling into assembly language for the CDC 6600 to gain improved efficiency. Times on the CDC 6600 for the assembly language version are approximately 1/10 of those shown in Table II. The register arrangement of the CDC 6600 is well suited to the radix-2 FFT; the author has written a subroutine occupying 59 words of storage on this machine, including the constants used to generate all needed trigonometric function values, that computes a complex FFT for $n = 1024$ in 42 ms.

## Transforming Real Data

As others have pointed out previously, a single-variate Fourier transform of $2n$ real data values can be computed by use of a complex Fourier transform of dimension $n$. In Appendix II, we include a FORTRAN subroutine REALTR,

**TABLE III**

**Numbers $\leq$ 100 000 Containing No Prime Factor Greater Than 5**

| | | | | | |
|---|---|---|---|---|---|
| 2 | 288 | 1920 | 6750 | 19440 | 48000 |
| 3 | 300 | 1944 | 6912 | 19683 | 48600 |
| 4 | 320 | 2000 | 7200 | 20000 | 49152 |
| 5 | 324 | 2025 | 7290 | 20250 | 50000 |
| 6 | 360 | 2048 | 7500 | 20480 | 50625 |
| 8 | 375 | 2160 | 7680 | 20736 | 51200 |
| 9 | 384 | 2187 | 7776 | 21600 | 51840 |
| 10 | 400 | 2250 | 8000 | 21870 | 52488 |
| 12 | 405 | 2304 | 8100 | 22500 | 54000 |
| 15 | 432 | 2400 | 8192 | 23040 | 54675 |
| 16 | 450 | 2430 | 8640 | 23328 | 55296 |
| 18 | 480 | 2500 | 8748 | 24000 | 56250 |
| 20 | 486 | 2560 | 9000 | 24300 | 57600 |
| 24 | 500 | 2592 | 9216 | 24576 | 58320 |
| 25 | 512 | 2700 | 9375 | 25000 | 59049 |
| 27 | 540 | 2880 | 9600 | 25600 | 60000 |
| 30 | 576 | 2916 | 9720 | 25920 | 60750 |
| 32 | 600 | 3000 | 10000 | 26244 | 61440 |
| 36 | 625 | 3072 | 10125 | 27000 | 62208 |
| 40 | 640 | 3125 | 10240 | 27648 | 62500 |
| 45 | 648 | 3200 | 10368 | 28125 | 64000 |
| 48 | 675 | 3240 | 10800 | 28800 | 64800 |
| 50 | 720 | 3375 | 10935 | 29160 | 65536 |
| 54 | 729 | 3456 | 11250 | 30000 | 65610 |
| 60 | 750 | 3600 | 11520 | 30375 | 67500 |
| 64 | 768 | 3645 | 11664 | 30720 | 69120 |
| 72 | 800 | 3750 | 12000 | 31104 | 69984 |
| 75 | 810 | 3840 | 12150 | 31250 | 72000 |
| 80 | 864 | 3888 | 12288 | 32000 | 72900 |
| 81 | 900 | 4000 | 12500 | 32400 | 73728 |
| 90 | 960 | 4050 | 12800 | 32768 | 75000 |
| 96 | 972 | 4096 | 12960 | 32805 | 76800 |
| 100 | 1000 | 4320 | 13122 | 33750 | 77760 |
| 108 | 1024 | 4374 | 13500 | 34560 | 78125 |
| 120 | 1080 | 4500 | 13824 | 34992 | 78732 |
| 125 | 1125 | 4608 | 14400 | 36000 | 80000 |
| 128 | 1152 | 4800 | 14580 | 36450 | 81000 |
| 135 | 1200 | 4860 | 15000 | 36864 | 81920 |
| 144 | 1215 | 5000 | 15360 | 37500 | 82944 |
| 150 | 1250 | 5120 | 15552 | 38400 | 84375 |
| 160 | 1280 | 5184 | 15625 | 38880 | 86400 |
| 162 | 1296 | 5400 | 16000 | 39366 | 87480 |
| 180 | 1350 | 5625 | 16200 | 40000 | 90000 |
| 192 | 1440 | 5760 | 16384 | 40500 | 91125 |
| 200 | 1458 | 5832 | 16875 | 40960 | 92160 |
| 216 | 1500 | 6000 | 17280 | 41472 | 93312 |
| 225 | 1536 | 6075 | 17496 | 43200 | 93750 |
| 240 | 1600 | 6144 | 18000 | 43740 | 96000 |
| 243 | 1620 | 6250 | 18225 | 45000 | 97200 |
| 250 | 1728 | 6400 | 18432 | 46080 | 98304 |
| 256 | 1800 | 6480 | 18750 | 46656 | 98415 |
| 270 | 1875 | 6561 | 19200 | 46875 | 100000 |

similar to an ALGOL procedure REALTRAN given elsewhere [7] by the author.

The real data values are stored alternately in the arrays $A$ and $B$,

$$A(1), B(1), A(2), B(2), \cdots A(n), B(n),$$

then we

$$\text{CALL FFT}(A, B, n, n, n, 1)$$

$$\text{CALL REALTR}\ (A, B, n, 1).$$

After scaling by $0.5/n$, the results in $A$ and $B$ are the Fourier cosine and sine coefficients, i.e.,

$$a_k = A(k+1)$$

$$b_k = B(k+1)$$

for $k = 0, 1, \cdots, n$, with $b = b_n = 0$. The inverse operation,

$$\text{CALL REALTR}\ (A, B, n, -1)$$

$$\text{CALL FFT}(A, B, n, n, n, -1),$$

after scaling by 1/2, evaluates the Fourier series and leaves the time domain values stored

$$A(1), B(1), A(2), B(2) \cdots A(n), B(n)$$

as originally.

The subroutine REALTR, called with ISN = 1, separates the complex transforms of the even- and odd-numbered data values, using the fact that the transform of real data has the complex conjugate symmetry

$$\alpha_{n-k} = \alpha_k{}^*$$

for $k = 1, 2, \cdots, n-1$, then performs a final radix-2 step to complete the transform for the $2n$ real values. If called with ISN $= -1$, the inverse operation is performed. The pair of calls

$$\text{CALL REALTR } (A, B, n, 1)$$
$$\text{CALL REALTR } (A, B, n, -1)$$

return the original values multiplied by 4, except for round-off errors.

Time on the CDC 6400 for $n = 1000$ is 0.100 second, and for $n = 2000$, 0.200 second. Time for REALTR is a linear function of $n$ for other numbers of data values. The rms error for the above pair of calls of REALTR was $1.6 \times 10^{-14}$ for both $n = 1000$ and $n = 2000$.

## Conclusion

We have described an efficient algorithm for computing the mixed radix fast Fourier transform and have illustrated this algorithm by a FORTRAN subroutine FFT for computing multivariate transforms. The principal means of improving efficiency is the reduction in the number of complex multiplications for an odd prime factor $p$ of $n$ to approximately

$$n(p-1)(p+3)/4p.$$

The algorithm also permutes the result in place by pair interchanges for the square factors of $n$, using additional temporary storage during permutation only when $n$ has two or more square-free factors.

A second subroutine REALTR for completing the transform of $2n$ real values is given, allowing efficient use of a complex transform of dimension $n$ for the major portion of the computing in this case.

By use of these two subroutines, Fourier transforms can be computed for many possible values of $n$, with nearly as good efficiency as for $n$ a power of 2. This expanded range of values has been found useful by the author in speech and economic time series analysis work.

Before Cooley and Tukey's paper [1], Good [8] presented a fast Fourier transform method based on decomposing $n$ into mutually prime factors. This algorithm uses a more complicated indexing scheme than the Cooley-Tukey algorithm, but avoids the rotation factor multiplications. While the restriction to mutually prime factors is an obstacle to general use of Good's algorithm, we note that it could have been used here to transform the square-free factors of $n$. This alternative has not been tried, but the potential gain, if any, appears small.


### References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.*, vol. 19, pp. 297–301, April 1965.

[2] W. M. Gentleman and G. Sande, "Fast Fourier transforms for fun and profit," *1966 Fall Joint Computer Conf., AFIPS Proc.*, vol. 29. Washington, D. C.: Spartan, 1966, pp. 563–578.

[3] R. C. Singleton, "An ALGOL procedure for the fast Fourier transform with arbitrary factors," *Commun ACM*, vol. 11, pp. 776–779, Algorithm 339, November 1968.

[4] N. M. Brenner, "Three FORTRAN programs that perform the Cooley–Tukey Fourier transform," M.I.T. Lincoln Lab., Lexington, Mass., Tech. Note 1967-2, July 1967.

[5] R. C. Singleton, "On computing the fast Fourier transform," *Commun. ACM*, vol. 10, pp. 647–654, October 1967.

[6] G. D. Bergland, "A fast Fourier transform algorithm using base 8 iterations," *Math. Comp.*, vol. 22, pp. 275–279, April 1968.

[7] R. C. Singleton, "ALGOL procedures for the fast Fourier transform," *Commun. ACM*, vol. 11, pp. 773–776, Algorithm 338, November 1968.

[8] I. J. Good, "The interaction algorithm and practical Fourier series," *J. Roy. Stat. Soc.*, ser. B, vol. 20, pp. 361–372, 1958; Addendum, vol. 22, pp. 372–375, 1960.


## Appendix I

### FORTRAN Subroutine FFT, for Computing the Mixed Radix Fourier Transform

```
      SUBROUTINE FFT(A,B,NTOT,N,NSPAN,ISN)
C  MULTIVARIATE COMPLEX FOURIER TRANSFORM, COMPUTED IN PLACE
C    USING MIXED-RADIX FAST FOURIER TRANSFORM ALGORITHM.
C  BY R. C. SINGLETON, STANFORD RESEARCH INSTITUTE, OCT. 1968
C  ARRAYS A AND B ORIGINALLY HOLD THE REAL AND IMAGINARY
C    COMPONENTS OF THE DATA, AND RETURN THE REAL AND
C    IMAGINARY COMPONENTS OF THE RESULTING FOURIER COEFFICIENTS.
C  MULTIVARIATE DATA IS INDEXED ACCORDING TO THE FORTRAN
C    ARRAY ELEMENT SUCCESSOR FUNCTION, WITHOUT LIMIT
C    ON THE NUMBER OF IMPLIED MULTIPLE SUBSCRIPTS.
C    THE SUBROUTINE IS CALLED ONCE FOR EACH VARIATE.
C    THE CALLS FOR A MULTIVARIATE TRANSFORM MAY BE IN ANY ORDER.
C  NTOT IS THE TOTAL NUMBER OF COMPLEX DATA VALUES.
C  N IS THE DIMENSION OF THE CURRENT VARIABLE.
C  NSPAN/N IS THE SPACING OF CONSECUTIVE DATA VALUES
C    WHILE INDEXING THE CURRENT VARIABLE.
C  THE SIGN OF ISN DETERMINES THE SIGN OF THE COMPLEX
C    EXPONENTIAL, AND THE MAGNITUDE OF ISN IS NORMALLY ONE.
C  A TRI-VARIATE TRANSFORM WITH A(N1,N2,N3), B(N1,N2,N3)
C    IS COMPUTED BY
C      CALL FFT(A,B,N1*N2*N3,N1,N1,1)
C      CALL FFT(A,B,N1*N2*N3,N2,N1*N2,1)
C      CALL FFT(A,B,N1*N2*N3,N3,N1*N2*N3,1)
C  FOR A SINGLE-VARIATE TRANSFORM,
C    NTOT = N = NSPAN = (NUMBER OF COMPLEX DATA VALUES), F.G.
C      CALL FFT(A,B,N,N,N,1)
C  THE DATA MAY ALTERNATIVELY BE STORED IN A SINGLE COMPLEX
C    ARRAY A, THEN THE MAGNITUDE OF ISN CHANGED TO TWO TO
C    GIVE THE CORRECT INDEXING INCREMENT AND A(2) USED TO
C    PASS THE INITIAL ADDRESS FOR THE SEQUENCE OF IMAGINARY
C    VALUES, E.G.
C      CALL FFT(A,A(2),NTOT,N,NSPAN,2)
C  ARRAYS AT(MAXF), CK(MAXF), BT(MAXF), SK(MAXF), AND NP(MAXP)
C    ARE USED FOR TEMPORARY STORAGE.  IF THE AVAILABLE STORAGE
C    IS INSUFFICIENT, THE PROGRAM IS TERMINATED BY A STOP.
C    MAXF MUST BE .GE. THE MAXIMUM PRIME FACTOR OF N.
C    MAXP MUST BE .GT. THE NUMBER OF PRIME FACTORS OF N.
C    IN ADDITION, IF THE SQUARE-FREE PORTION K OF N HAS TWO OR
C    MORE PRIME FACTORS, THEN MAXP MUST BE .GE. K-1.
      DIMENSION A(1),B(1)
C  ARRAY STORAGE IN NFAC FOR A MAXIMUM OF 11 FACTORS OF N.
C  IF N HAS MORE THAN ONE SQUARE-FREE FACTOR, THE PRODUCT OF THE
C    SQUARE-FREE FACTORS MUST BE .LE. 210
      DIMENSION NFAC(11),NP(209)
C  ARRAY STORAGE FOR MAXIMUM PRIME FACTOR OF 23
      DIMENSION AT(23),CK(23),BT(23),SK(23)
      EQUIVALENCE (I,II)
C  THE FOLLOWING TWO CONSTANTS SHOULD AGREE WITH THE ARRAY DIMENSIONS.
      MAXF=23
      MAXP=209
      IF(N .LT. 2) RETURN
      INC=ISN
      RAD=8.0*ATAN(1.0)
      S72=RAD/5.0
      C72=COS(S72)
      S72=SIN(S72)
      S120=SQRT(0.75)
      IF(ISN .GE. 0) GO TO 10
      S72=-S72
      S120=-S120
      RAD=-RAD
      INC=-INC
   10 NT=INC*NTOT
      KS=INC*NSPAN
      KSPAN=KS
      NN=NT-INC
```

```
      JC=KS/N
      RADF=RAD*FLOAT(JC)*0.5
      I=0
      JF=0
C  DETERMINE THE FACTORS OF N
      M=0
      K=N
      GO TO 20
   15 M=M+1
      NFAC(M)=4
      K=K/16
   20 IF(K-(K/16)*16 .EQ. 0) GO TO 15
      J=3
      JJ=9
      GO TO 30
   25 M=M+1
      NFAC(M)=J
      K=K/JJ
   30 IF(MOD(K,JJ) .EQ. 0) GO TO 25
      J=J+2
      JJ=J**2
      IF(JJ .LE. K) GO TO 30
      IF(K .GT. 4) GO TO 40
      KT=M
      NFAC(M+1)=K
      IF(K .NE. 1) M=M+1
      GO TO 80
   40 IF(K-(K/4)*4 .NE. 0) GO TO 50
      M=M+1
      NFAC(M)=2
      K=K/4
   50 KT=M
      J=2
   60 IF(MOD(K,J) .NE. 0) GO TO 70
      M=M+1
      NFAC(M)=J
      K=K/J
   70 J=((J+1)/2)*2+1
      IF(J .LE. K) GO TO 60
   80 IF(KT .EQ. 0) GO TO 100
      J=KT
   90 M=M+1
      NFAC(M)=NFAC(J)
      J=J-1
      IF(J .NE. 0) GO TO 90
C  COMPUTE FOURIER TRANSFORM
  100 SD=RADF/FLOAT(KSPAN)
      CD=2.0*SIN(SD)**2
      SD=SIN(SD+SD)
      KK=1
      I=I+1
      IF(NFAC(I) .NE. 2) GO TO 400
C  TRANSFORM FOR FACTOR OF 2 (INCLUDING ROTATION FACTOR)
      KSPAN=KSPAN/2
      K1=KSPAN+2
  210 K2=KK+KSPAN
      AK=A(K2)
      BK=B(K2)
      A(K2)=A(KK)-AK
      B(K2)=B(KK)-BK
      A(KK)=A(KK)+AK
      B(KK)=B(KK)+BK
      KK=K2+KSPAN
      IF(KK .LE. NN) GO TO 210
      KK=KK-NN
      IF(KK .LE. JC) GO TO 210
      IF(KK .GT. KSPAN) GO TO 800
  220 C1=1.0-CD
      S1=SD
  230 K2=KK+KSPAN
      AK=A(KK)-A(K2)
      BK=B(KK)-B(K2)
      A(KK)=A(KK)+A(K2)
      B(KK)=B(KK)+B(K2)
      A(K2)=C1*AK-S1*BK
      B(K2)=S1*AK+C1*BK
      KK=K2+KSPAN
      IF(KK .LT. NT) GO TO 230
      K2=KK-NT
      C1=-C1
      KK=K1-K2
      IF(KK .GT. K2) GO TO 230
      AK=C1-(CD*C1+SD*S1)
      S1=(SD*C1-CD*S1)+S1
C  THE FOLLOWING THREE STATEMENTS COMPENSATE FOR TRUNCATION
C    ERROR.  IF ROUNDED ARITHMETIC IS USED, SUBSTITUTE
C     C1=AK
      C1=0.5/(AK**2+S1**2)+0.5
      S1=C1*S1
      C1=C1*AK
      KK=KK+JC
      IF(KK .LT. K2) GO TO 230
      K1=K1+INC+INC
      KK=(K1-KSPAN)/2+JC
      IF(KK .LE. JC+JC) GO TO 220
      GO TO 100
C  TRANSFORM FOR FACTOR OF 3 (OPTIONAL CODE)
  320 K1=KK+KSPAN
      K2=K1+KSPAN
      AK=A(KK)
      BK=B(KK)
      AJ=A(K1)+A(K2)
      BJ=B(K1)+B(K2)
      A(KK)=AK+AJ
      B(KK)=BK+BJ
      AK=-0.5*AJ+AK
      BK=-0.5*BJ+BK
      AJ=(A(K1)-A(K2))*S120
      BJ=(B(K1)-B(K2))*S120
      A(K1)=AK-BJ
      B(K1)=BK+AJ
      A(K2)=AK+BJ
      B(K2)=BK-AJ
      KK=K2+KSPAN
      IF(KK .LT. NN) GO TO 320
      KK=KK-NN
      IF(KK .LE. KSPAN) GO TO 320
      GO TO 700
C  TRANSFORM FOR FACTOR OF 4
  400 IF(NFAC(I) .NE. 4) GO TO 600
      KSPNN=KSPAN
      KSPAN=KSPAN/4
  410 C1=1.0
      S1=0
  420 K1=KK+KSPAN
      K2=K1+KSPAN
      K3=K2+KSPAN
      AKP=A(KK)+A(K2)
      AKM=A(KK)-A(K2)
      AJP=A(K1)+A(K3)
      AJM=A(K1)-A(K3)
      A(KK)=AKP+AJP
      AJP=AKP-AJP
      BKP=B(KK)+B(K2)
      BKM=B(KK)-B(K2)
      BJP=B(K1)+B(K3)
      BJM=B(K1)-B(K3)
      B(KK)=BKP+BJP
      BJP=BKP-BJP
      IF(ISN .LT. 0) GO TO 450
      AKP=AKM-BJM
      AKM=AKM+BJM
      BKP=BKM+AJM
      BKM=BKM-AJM
      IF(S1 .EQ. 0.0) GO TO 460
  430 A(K1)=AKP*C1-BKP*S1
      B(K1)=AKP*S1+BKP*C1
      A(K2)=AJP*C2-BJP*S2
      B(K2)=AJP*S2+BJP*C2
      A(K3)=AKM*C3-BKM*S3
      B(K3)=AKM*S3+BKM*C3
      KK=K3+KSPAN
      IF(KK .LE. NT) GO TO 420
  440 C2=C1-(CD*C1+SD*S1)
      S1=(SD*C1-CD*S1)+S1
C  THE FOLLOWING THREE STATEMENTS COMPENSATE FOR TRUNCATION
C    ERROR.  IF ROUNDED ARITHMETIC IS USED, SUBSTITUTE
C     C1=C2
      C1=0.5/(C2**2+S1**2)+0.5
      S1=C1*S1
      C1=C1*C2
      C2=C1**2-S1**2
      S2=2.0*C1*S1
      C3=C2*C1-S2*S1
      S3=C2*S1+S2*C1
      KK=KK-NT+JC
      IF(KK .LE. KSPAN) GO TO 420
      KK=KK-KSPAN+INC
      IF(KK .LE. JC) GO TO 410
      IF(KSPAN .EQ. JC) GO TO 800
      GO TO 100
  450 AKP=AKM+BJM
      AKM=AKM-BJM
      BKP=BKM-AJM
      BKM=BKM+AJM
      IF(S1 .NE. 0.0) GO TO 430
  460 A(K1)=AKP
      B(K1)=BKP
      A(K2)=AJP
      B(K2)=BJP
      A(K3)=AKM
      B(K3)=BKM
      KK=K3+KSPAN
      IF(KK .LE. NT) GO TO 420
      GO TO 440
C  TRANSFORM FOR FACTOR OF 5 (OPTIONAL CODE)
  510 C2=C72**2-S72**2
      S2=2.0*C72*S72
  520 K1=KK+KSPAN
      K2=K1+KSPAN
      K3=K2+KSPAN
      K4=K3+KSPAN
      AKP=A(K1)+A(K4)
      AKM=A(K1)-A(K4)
      BKP=B(K1)+B(K4)
      BKM=B(K1)-B(K4)
      AJP=A(K2)+A(K3)
      AJM=A(K2)-A(K3)
      BJP=B(K2)+B(K3)
      BJM=B(K2)-B(K3)
      AA=A(KK)
      BB=B(KK)
      A(KK)=AA+AKP+AJP
      B(KK)=BB+BKP+BJP
      AK=AKP*C72+AJP*C2+AA
      BK=BKP*C72+BJP*C2+BB
      AJ=AKM*S72+AJM*S2
      BJ=BKM*S72+BJM*S2
      A(K1)=AK-BJ
      A(K4)=AK+BJ
      B(K1)=BK+AJ
      B(K4)=BK-AJ
      AK=AKP*C2+AJP*C72+AA
      BK=BKP*C2+BJP*C72+BB
      AJ=AKM*S2-AJM*S72
      BJ=BKM*S2-BJM*S72
      A(K2)=AK-BJ
      A(K3)=AK+BJ
      B(K2)=BK+AJ
      B(K3)=BK-AJ
      KK=K4+KSPAN
      IF(KK .LT. NN) GO TO 520
      KK=KK-NN
      IF(KK .LE. KSPAN) GO TO 520
      GO TO 700
C  TRANSFORM FOR ODD FACTORS
  600 K=NFAC(I)
      KSPNN=KSPAN
      KSPAN=KSPAN/K
      IF(K .EQ. 3) GO TO 320
      IF(K .EQ. 5) GO TO 510
      IF(K .EQ. JF) GO TO 640
      JF=K
      S1=RAD/FLOAT(K)
      C1=COS(S1)
      S1=SIN(S1)
      IF(JF .GT. MAXF) GO TO 998
      CK(JF)=1.0
      SK(JF)=0.0
      J=1
  630 CK(J)=CK(K)*C1+SK(K)*S1
      SK(J)=CK(K)*S1-SK(K)*C1
      K=K-1
      CK(K)=CK(J)
      SK(K)=-SK(J)
      J=J+1
      IF(J .LT. K) GO TO 630
```

```
  640 K1=KK
      K2=KK+KSPNN
      AA=A(KK)
      BB=B(KK)
      AK=AA
      BK=BB
      J=1
      K1=K1+KSPAN
  650 K2=K2-KSPAN
      J=J+1
      AT(J)=A(K1)+A(K2)
      AK=AT(J)+AK
      BT(J)=B(K1)+B(K2)
      BK=BT(J)+BK
      J=J+1
      AT(J)=A(K1)-A(K2)
      BT(J)=B(K1)-B(K2)
      K1=K1+KSPAN
      IF(K1 .LT. K2) GO TO 650
      A(KK)=AK
      B(KK)=BK
      K1=KK
      K2=KK+KSPNN
      J=1
  660 K1=K1+KSPAN
      K2=K2-KSPAN
      JJ=J
      AK=AA
      BK=BB
      AJ=0.0
      BJ=0.0
      K=1
  670 K=K+1
      AK=AT(K)*CK(JJ)+AK
      BK=BT(K)*CK(JJ)+BK
      K=K+1
      AJ=AT(K)*SK(JJ)+AJ
      BJ=BT(K)*SK(JJ)+BJ
      JJ=JJ+J
      IF(JJ .GT. JF) JJ=JJ-JF
      IF(K .LT. JF) GO TO 670
      K=JF-J
      A(K1)=AK-BJ
      B(K1)=BK+AJ
      A(K2)=AK+BJ
      B(K2)=BK-AJ
      J=J+1
      IF(J .LT. K) GO TO 660
      KK=KK+KSPNN
      IF(KK .LE. NN) GO TO 640
      KK=KK-NN
      IF(KK .LE. KSPAN) GO TO 640
C  MULTIPLY BY ROTATION FACTOR (EXCEPT FOR FACTORS OF 2 AND 4)
  700 IF(I .EQ. M) GO TO 800
      KK=JC+1
  710 C2=1.0-CD
      S1=SD
  720 C1=C2
      S2=S1
      KK=KK+KSPAN
  730 AK=A(KK)
      A(KK)=C2*AK-S2*B(KK)
      B(KK)=S2*AK+C2*B(KK)
      KK=KK+KSPNN
      IF(KK .LE. NT) GO TO 730
      AK=S1*S2
      S2=S1*C2+C1*S2
      C2=C1*C2-AK
      KK=KK-NT+KSPAN
      IF(KK .LE. KSPNN) GO TO 730
      C2=C1-(CD*C1+SD*S1)
      S1=S1+(SD*C1-CD*S1)
C  THE FOLLOWING THREE STATEMENTS COMPENSATE FOR TRUNCATION
C    ERROR.  IF ROUNDED ARITHMETIC IS USED, THEY MAY
C    BE DELETED.
      C1=0.5/(C2**2+S1**2)+0.5
      S1=C1*S1
      C2=C1*C2
      KK=KK-KSPNN+JC
      IF(KK .LE. KSPAN) GO TO 720
      KK=KK-KSPAN+JC+INC
      IF(KK .LE. JC+JC) GO TO 710
      GO TO 100
C  PERMUTE THE RESULTS TO NORMAL ORDER---DONE IN TWO STAGES
C  PERMUTATION FOR SQUARE FACTORS OF N
  800 NP(1)=KS
      IF(KT .EQ. 0) GO TO 890
      K=KT+KT+1
      IF(M .LT. K) K=K-1
      J=1
      NP(K+1)=JC
  810 NP(J+1)=NP(J)/NFAC(J)
      NP(K)=NP(K+1)*NFAC(J)
      J=J+1
      K=K-1
      IF(J .LT. K) GO TO 810
      K3=NP(K+1)
      KSPAN=NP(2)
      KK=JC+1
      K2=KSPAN+1
      J=1
      IF(N .NE. NTOT) GO TO 850
C  PERMUTATION FOR SINGLE-VARIATE TRANSFORM (OPTIONAL CODE)
  820 AK=A(KK)
      A(KK)=A(K2)
      A(K2)=AK
      BK=B(KK)
      B(KK)=B(K2)
      B(K2)=BK
      KK=KK+INC
      K2=KSPAN+K2
      IF(K2 .LT. KS) GO TO 820
  830 K2=K2-NP(J)
      J=J+1
      K2=NP(J+1)+K2
      IF(K2 .GT. NP(J)) GO TO 830
      J=1
  840 IF(KK .LT. K2) GO TO 820
      KK=KK+INC
      K2=KSPAN+K2
      IF(K2 .LT. KS) GO TO 840
      IF(KK .LT. KS) GO TO 830
      JC=K3
      GO TO 890
C  PERMUTATION FOR MULTIVARIATE TRANSFORM
  850 K=KK+JC
  860 AK=A(KK)
      A(KK)=A(K2)
      A(K2)=AK
      BK=B(KK)
      B(KK)=B(K2)
      B(K2)=BK
      KK=KK+INC
      K2=K2+INC
      IF(KK .LT. K) GO TO 860
      KK=KK+KS-JC
      K2=K2+KS-JC
      IF(KK .LT. NT) GO TO 850
      K2=K2-NT+KSPAN
      KK=KK-NT+JC
      IF(K2 .LT. KS) GO TO 850
  870 K2=K2-NP(J)
      J=J+1
      K2=NP(J+1)+K2
      IF(K2 .GT. NP(J)) GO TO 870
      J=1
  880 IF(KK .LT. K2) GO TO 850
      KK=KK+JC
      K2=KSPAN+K2
      IF(K2 .LT. KS) GO TO 880
      IF(KK .LT. KS) GO TO 870
      JC=K3
  890 IF(2*KT+1 .GE. M) RETURN
      KSPNN=NP(KT+1)
C  PERMUTATION FOR SQUARE-FREE FACTORS OF N
      J=M-KT
      NFAC(J+1)=1
  900 NFAC(J)=NFAC(J)*NFAC(J+1)
      J=J-1
      IF(J .NE. KT) GO TO 900
      KT=KT+1
      NN=NFAC(KT)-1
      IF(NN .GT. MAXP) GO TO 998
      JJ=0
      J=0
      GO TO 906
  902 JJ=JJ-K2
      K2=KK
      K=K+1
      KK=NFAC(K)
  904 JJ=KK+JJ
      IF(JJ .GE. K2) GO TO 902
      NP(J)=JJ
  906 K2=NFAC(KT)
      K=KT+1
      KK=NFAC(K)
      J=J+1
      IF(J .LE. NN) GO TO 904
C  DETERMINE THE PERMUTATION CYCLES OF LENGTH GREATER THAN 1
      J=0
      GO TO 914
  910 K=KK
      KK=NP(K)
      NP(K)=-KK
      IF(KK .NE. J) GO TO 910
      K3=KK
  914 J=J+1
      KK=NP(J)
      IF(KK .LT. 0) GO TO 914
      IF(KK .NE. J) GO TO 910
      NP(J)=-J
      IF(J .NE. NN) GO TO 914
      MAXF=INC*MAXF
C  REORDER A AND B, FOLLOWING THE PERMUTATION CYCLES
      GO TO 950
  924 J=J-1
      IF(NP(J) .LT. 0) GO TO 924
      JJ=JC
  926 KSPAN=JJ
      IF(JJ .GT. MAXF) KSPAN=MAXF
      JJ=JJ-KSPAN
      K=NP(J)
      KK=JC*K+II+JJ
      K1=KK+KSPAN
      K2=0
  928 K2=K2+1
      AT(K2)=A(K1)
      BT(K2)=B(K1)
      K1=K1-INC
      IF(K1 .NE. KK) GO TO 928
  932 K1=KK+KSPAN
      K2=K1-JC*(K+NP(K))
      K=-NP(K)
  936 A(K1)=A(K2)
      B(K1)=B(K2)
      K1=K1-INC
      K2=K2-INC
      IF(K1 .NE. KK) GO TO 936
      KK=K2
      IF(K .NE. J) GO TO 932
      K1=KK+KSPAN
      K2=0
  940 K2=K2+1
      A(K1)=AT(K2)
      B(K1)=BT(K2)
      K1=K1-INC
      IF(K1 .NE. KK) GO TO 940
      IF(JJ .NE. 0) GO TO 926
      IF(J .NE. 1) GO TO 924
  950 J=K3+1
      NT=NT-KSPNN
      II=NT-INC+1
      IF(NT .GE. 0) GO TO 924
      RETURN
C  ERROR FINISH, INSUFFICIENT ARRAY STORAGE
  998 ISN=0
C     PRINT 999
      STOP
  999 FORMAT(44HOARRAY BOUNDS EXCEEDED WITHIN SUBROUTINE FFT)
      END
```

## Appendix II

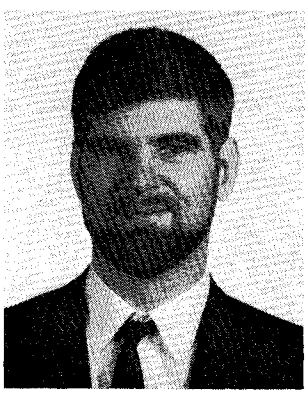
### FORTRAN Subroutine REALTR, for Completing the Fourier Transform of 2*n* Real Values

```
      SUBROUTINE REALTR(A,B,N,ISN)
C  IF ISN=1, THIS SUBROUTINE COMPLETES THE FOURIER TRANSFORM
C    OF 2*N REAL DATA VALUES, WHERE THE ORIGINAL DATA VALUES ARE
C    STORED ALTERNATELY IN ARRAYS A AND B, AND ARE FIRST
C    TRANSFORMED BY A COMPLEX FOURIER TRANSFORM OF DIMENSION N.
C    THE COSINE COEFFICIENTS ARE IN A(1),A(2),...A(N+1) AND
C    THE SINE COEFFICIENTS ARE IN B(1),B(2),...B(N+1).
C    A TYPICAL CALLING SEQUENCE IS
C      CALL FFT(A,B,N,N,N,1)
C      CALL REALTR(A,B,N,1)
C    THE RESULTS SHOULD BE MULTIPLIED BY 0.5/N TO GIVE THE
C    USUAL SCALING OF COEFFICIENTS.
C  IF ISN=-1, THE INVERSE TRANSFORMATION IS DONE, THE FIRST STEP
C    IN EVALUATING A REAL FOURIER SERIES.
C    A TYPICAL CALLING SEQUENCE IS
C      CALL REALTR(A,B,N,-1)
C      CALL FFT(A,B,N,N,N,-1)
C    THE RESULTS SHOULD BE MULTIPLIED BY 0.5 TO GIVE THE USUAL
C    SCALING, AND THE TIME DOMAIN RESULTS ALTERNATE IN ARRAYS A
C    AND B, I.E. A(1),B(1),A(2),B(2),...A(N),B(N).
C  THE DATA MAY ALTERNATIVELY BE STORED IN A SINGLE COMPLEX
C    ARRAY A, THEN THE MAGNITUDE OF ISN CHANGED TO TWO TO
C    GIVE THE CORRECT INDEXING INCREMENT AND A(2) USED TO
C    PASS THE INITIAL ADDRESS FOR THE SEQUENCE OF IMAGINARY
C    VALUES, E.G.
C      CALL FFT(A,A(2),N,N,N,2)
C      CALL REALTR(A,A(2),N,2)
C    IN THIS CASE, THE COSINE AND SINE COEFFICIENTS ALTERNATE IN A.
C  BY R. C. SINGLETON, STANFORD RESEARCH INSTITUTE, OCT. 1968
      DIMENSION A(1),B(1)
      REAL IM
      INC=IABS(ISN)
      NK=N*INC+2
      NH=NK/2
      SD=2.0*ATAN(1.0)/FLOAT(N)
      CD=2.0*SIN(SD)**2
      SD=SIN(SD+SD)
      SN=0.0
      IF(ISN .LT. 0) GO TO 30
      CN=1.0
      A(NK-1)=A(1)
      B(NK-1)=B(1)
   10 DO 20 J=1,NH,INC
      K=NK-J
      AA=A(J)+A(K)
      AB=A(J)-A(K)
      BA=B(J)+B(K)
      BB=B(J)-B(K)
      RE=CN*BA+SN*AB
      IM=SN*BA-CN*AB
      B(K)=IM-BB
      B(J)=IM+BB
      A(K)=AA-RE
      A(J)=AA+RE
      AA=CN-(CD*CN+SD*SN)
      SN=(SD*CN-CD*SN)+SN
C  THE FOLLOWING THREE STATEMENTS COMPENSATE FOR TRUNCATION
C    ERROR.  IF ROUNDED ARITHMETIC IS USED, SUBSTITUTE
C  20 CN=AA
      CN=0.5/(AA**2+SN**2)+0.5
      SN=CN*SN
   20 CN=CN*AA
      RETURN
   30 CN=-1.0
      SD=-SD
      GO TO 10
      END
```

## Appendix III

### Rader's Radix-5 Method, for Possible Substitution in Subroutine FFT in Appendix I

```
C  TRANSFORM FOR FACTOR OF 5 (OPTIONAL CODE),
C    USING METHOD DUE TO C. M. RADER
  510 C2=0.25*SQRT(5.0)
      S2=2.0*C72*S72
  520 K1=KK+KSPAN
      K2=K1+KSPAN
      K3=K2+KSPAN
      K4=K3+KSPAN
      AKP=A(K1)+A(K4)
      AKM=A(K1)-A(K4)
      BKP=B(K1)+B(K4)
      BKM=B(K1)-B(K4)
      AJP=A(K2)+A(K3)
      AJM=A(K2)-A(K3)
      BJP=B(K2)+B(K3)
      BJM=B(K2)-B(K3)
      AK=AKP+AJP
      AJP=(AKP-AJP)*C2
      BK=BKP+BJP
      BJP=(BKP-BJP)*C2
      AKP=A(KK)-0.25*AK
      A(KK)=A(KK)+AK
      BKP=B(KK)-0.25*BK
      B(KK)=B(KK)+BK
      AK=AKP+AJP
      AJP=AKP-AJP
      BK=BKP+BJP
      BJP=BKP-BJP
      AKP=AKM*S72+AJM*S2
      AKM=AKM*S2-AJM*S72
      BKP=BKM*S72+BJM*S2
      BKM=BKM*S2-BJM*S72
      A(K1)=AK-BKP
      A(K4)=AK+BKP
      B(K1)=BK+AKP
      B(K4)=BK-AKP
      A(K2)=AJP-BKM
      A(K3)=AJP+BKM
      B(K2)=BJP+AKM
      B(K3)=BJP-AKM
      KK=K4+KSPAN
      IF(KK .LT. NN) GO TO 520
      KK=KK-NN
      IF(KK .LE. KSPAN) GO TO 520
      GO TO 700
```



**Richard C. Singleton** (S'48 – A'52 – M'58 – SM'60) was born in Schenectady, N. Y., on February 21, 1928. He received the B.S. and M.S. degrees in electrical engineering in 1950 from the Massachusetts Institute of Technology, Cambridge. He received the M.B.A. degree in business administration in 1952, and the Ph.D. degree in mathematical statistics in 1960, from Stanford University, Stanford, Calif.

While at M.I.T., he worked one year as a co-op student at Philco Corporation, Philadelphia, Pa. Since January, 1952, he has been employed by the Stanford Research Institute, Menlo Park, Calif. While on leave of absence, from 1958 to 1960, he was employed by the Applied Mathematics and Statistics Laboratories at Stanford University; there he did research on the mathematic theory of inventory control, and completed work for his doctorate. At the Stanford Research Institute he was initially engaged as an Operations Research Analyst, working on inventory control problems, airline passenger reservation system design, and management control studies. He has recently been working on problems of statistical inference, threshold switching functions, time series analysis, and coding theory. He is an Associate Editor of *Information Sciences.*

Dr. Singleton is a member of the Operations Research Society of America, the Institute of Mathematical Statistics, the Research Society of America, Sigma Xi, and Eta Kappa Nu.